THIS IS THE SEQUENTIAL BOOK VERSION

A Systematic Approach to Sequential Algorithms

Vijay K. Garg
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084

To my family

Contents

List of Figures

Chapter 1

Introduction

The goal of the book is to present a unified treatment of a wide variety of algorithms. Linear programming, or integer programming, serves as a tool for providing insights into a large class of problems. The shortest path problem, the max-flow problem, the stable marriage problem, and the weighted bipartite matching problem can all be modeled and analyzed using linear programming techniques. Linear programming formulation provides additional insights into the problem such as notions of dual problems, certificates for optimality, and lower and upper bounds on the objective function. In this book, we present another general technique called *lattice-linear predicate detection* that can solve many problems. We use this method to solve the generalization of many fundamental problems in combinatorial optimization, including the stable marriage problem [GS62], the shortest path problem [Dij59], and the assignment problem [Mun57]. Due to the importance and applications of these problems, each one has been the subject of numerous books and thousands of papers. The classical algorithms to solve these problems are the Gale-Shapley algorithm [GS62] for the stable marriage problem, Dijkstra's algorithm [Dij59] for the shortest path problem, and Kuhn's Hungarian method [Mun57] to solve the assignment problem (or equivalently, Demange, Gale, Sotomayor auction-based algorithm [DGS86] for market clearing prices). Could there be a single efficient algorithm that solves all of these problems?

The book presents a technique that solves not only these problems but more *general* versions of each of the above problems. We seek the optimal solution for these problems that satisfy additional constraints modeled using a *lattice-linear* predicate [CG98]. When there are no additional constraints (i.e., when the set of constraints is empty), our approach reverts to addressing the classical versions of these problems.

Our technique requires the underlying search space to be viewed as a distributive lattice [Bir67, DP90]. Common to all these seemingly disparate combinatorial optimization problems is the structure of the *feasible* solution space. The set of all stable marriages, the set of all feasible rooted trees for the shortest path problem, and the set of all market clearing prices are all closed under the meet operation of the lattice. If the order is appropriately defined, then finding the optimal solution (the man-optimal stable marriage, the shortest path cost vector, the minimum market clearing price vector) is equivalent to finding the infimum of all feasible solutions in the lattice.

We note here that it is well-known that the set of stable marriage and the set of market clearing price vectors form distributive lattices. The set of stable marriages forms a distributive lattice is given in [Knu97] where the result is attributed to Conway. The set of market clearing price vectors forms a distributive lattice is given in [SS71]. However, the algorithms to find the man-optimal stable marriage and the minimum market clearing price vectors are not derived from the lattice property. In our method, once the

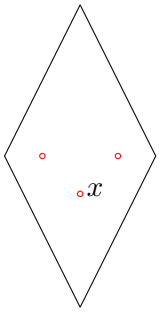


Figure 1.1: Search space with feasible solutions. The feasible solutions are shown in red. The solution x is the least feasible solution.

lattice-linearity of the feasible solution space is established, the algorithm to find the optimal solution falls out as a consequence. The reference [Gar20] derives the Gale-Shapley's algorithm, Dijkstra's algorithm and Demange-Gale-Sotomayor's algorithm from a single algorithm by exploiting the lattice property.

The lattice-linear predicate detection method to solve the combinatorial optimization problem is as follows. The first step is to define a lattice of vectors, L, such that each vector is *assigned* a point in the search space. For the stable marriage problem, the vector corresponds to the assignment of men to women (or equivalently, the choice number for each man). For the shortest path problem, the vector assigns a cost to each node. For the market clearing price problem, the vector assigns a price to each item. The comparison operation ($\leq$) is defined on the set of vectors such that the least vector, if feasible, is the extremal solution of interest. For example, in the stable marriage problem if each man orders women according to his preferences and every man is assigned the first woman in the list, then this solution is the man-optimal solution whenever the assignment is a matching and has no blocking pair. Similarly, in the shortest path problem and the minimum market clearing price problem, the zero vector would be optimal if it were feasible.

The second step in our method is to define a boolean predicate B that models feasibility of the vector. For the stable marriage problem, an assignment is feasible iff it is a matching and there is no blocking pair. For the shortest path problem, an assignment is feasible iff there exists a rooted spanning tree at the source vertex such that the cost of each vertex is greater than the cost of traversing the path in the rooted tree. For the minimum market clearing price problem, a price vector is feasible iff it is a market clearing price vector.

The third step is to show that the feasibility predicate is a lattice-linear predicate [CG98]. The lattice-linearity property allows one to search for a feasible solution efficiently. If any point in the search space is not feasible, it allows one to make progress towards the optimal feasible solution without any need for exploring multiple paths in the lattice. Moreover, multiple processes can make progress towards the feasible solution independently. In a finite distributive lattice, it is clear that the maximum number of such advancement steps before one finds the optimal solution or reaches the top element of the lattice is equal to the height of the lattice. Once this step is done, we get the following outcomes.

First, by applying the lattice-linear predicate detection algorithm to unconstrained problems, we get the Gale-Shapley algorithm for the stable marriage problem, Dijkstra's algorithm for the shortest path problem and Demange, Gale, Sotomayor's algorithm for the minimum market clearing price. In fact, the lattice-linear predicate detection method yields a parallel version of these algorithms and by restricting

these to their sequential counterparts, we get these classical sequential algorithms.

Second, we get solutions for the constrained version of each of these problems, whenever the constraints are lattice-linear. We solve the *Constrained stable marriage Problem* where in addition to men's preferences and women's preferences, there may be a set of lattice-linear constraints. For example, we may require that Peter's regret [GI89] should be less than that of Paul, where the *regret* of a man in a matching is the choice number he is assigned. We note here that some special cases of the constrained stable marriage problems have been studied. Dias et al [DdFdFS03, CM16] study the stable marriage problem with restricted pairs. A restricted pair is either a *forced* pair which is required to be in the matching, or a *forbidden* pair which must not be in the matching. Both of these constraints are *lattice-linear* and therefore can be modeled in our system. The constrained shortest path problem asks for a rooted tree at the source node with the smallest cost at each vertex that satisfies additional constraints of the form "the cost of reaching node x is at least the cost of reaching node y", "the cost of reaching x must be equal to the cost of reaching y", and "the cost of reaching x must be within δ of the cost of reaching y". For the market clearing price problem, we consider constraints on the clearing prices of the form that item i must be priced at least as much as item j, or the difference in prices for item i and j must not exceed δ.

Third, by applying a constructive version of Birkhoff's theorem on finite distributive lattices [Bir67, DP90], we give an algorithm that outputs a succinct representation of all feasible solutions. In particular, the join-irreducible elements [DP90] of the feasible sublattice can be determined efficiently (in polynomial time). For the constrained stable marriage problem, we get a concise representation of all stable marriages that satisfy given constraints. Thus, our method yields a more general version of rotation posets [GI89] to represent all *constrained* stable marriages. Analogously, we get a concise representation of all constrained integral market clearing price vectors.

In the remaining chapter, we provide an introduction to algorithms, emphasizing their definition, efficiency analysis, and mathematical underpinnings. We explore what constitutes an algorithm and formalize efficiency with asymptotic notation.

1.1 What is an Algorithm?

An algorithm is a finite, unambiguous sequence of instructions that solves a problem or computes a function, transforming an input into an output in a finite number of steps. It must exhibit *finiteness* (terminates), *definiteness* (each step is precisely specified), and *effectiveness* (each operation is basic and executable). Algorithms underpin computer science, solving problems from simple arithmetic to complex optimization.

We start with one of earliest algorithms due to Euclid. Suppose we are required to find the greatest common divisor (GCD) of two natural numbers a and b (integers greater than or equal to 1). If they are both equal, then we have the answer. Suppose that a is greater than b, then we claim that it is safe to reduce a to $a - b$ (prove it!). Similarly, if b is greater than a, then we reduce b to $b - a$. The algorithm terminates with all numbers identical and equal to the gcd.

For $a = 48$, $b = 18$, in the first step, we get a equals 30 and b equals 18. Since a is still greater than b, a becomes 12 and b is still 18. Now b is greater than a. So, in the next step b becomes $b - a$ which equals 6. Finally, a becomes 6. Now, both a and b are equal to 6 and the algorithm terminates. It is easy to see that the algorithm always terminates. All values are initially non-zero positive integers. They stay non-zero and integral and always decrease; therefore, the algorithm must terminate.

In our algorithm, we have used subtraction to decrease the numbers. We leave it as an exercise for the reader to use the *mod* function to speed up the above algorithm.

Algorithm GCD: GCD(a, b)

Input : Two positive integers a, b greater than or equal to 1
Output: The greatest common divisor of a and b

```
1 while a ≠ b do
2 |   if (a > b) then  a ← a − b;
3 |   if (b > a) then  b ← b − a;
4 end
5 return a
```

Suppose that we want to compute the factorial of a non-negative integer n. Algorithm Factorial shows a method that uses *recursion*. A recursive approach requires specification of the *base* case, when n equals 0. For a higher value of n, we use the value of Factorial$(n-1)$ to compute Factorial(n).

Algorithm Factorial: Factorial(n)

Input : A non-negative integer n
Output: $n!$

```
1 if n = 0 then  return 1 ;
2 return n · Factorial(n − 1)
```

As a final simple example of an algorithm, suppose that we are required to find the largest element in an array A. Algorithm MaxElement uses a *for* loop to find this element.

Algorithm MaxElement: MaxElement(A)

Input : Array A of size n
Output: Maximum element in A

```
1 max ← A[0] ;
2 for i = 1 to n − 1 do
3 |   if A[i] > max then  max ← A[i] ;
4 end
5 return max
```

1.2 Asymptotic Notation (Big O, Big Omega, Big Theta)

Asymptotic notation quantifies runtime growth, abstracting constants and lower-order terms to focus on scalability. It is the cornerstone of algorithmic comparison.

- $T(n) = O(f(n))$ if $\exists c, n_0 > 0$ such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$. For example, suppose that $T(n) = 2n + 5$. Since $2n + 5 \leq 3n$ for $n \geq 5$, we conclude that $T(n)$ equals $O(n)$.
- We say that $T(n) = \Omega(f(n))$ if $\exists c, n_0 > 0$ such that $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$. For example, suppose that $T(n) = n^2 - n$. Since $n^2 - n \geq 0.5n^2$ for $n \geq 2$, we conclude that $T(n)$ equals $\Omega(n^2)$.

- We say that $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$. For example, suppose that $T(n) = 4n^2 + 3n$, $3n^2 \leq 4n^2 + 3n \leq 5n^2$ for $n \geq 3$, we conclude that $T(n)$ equals $\Theta(n^2)$.

We first give an example, where $T(n)$ is a linear growing function of n. Let $T(n) = 7n + 10$. Then, it is easy to see that $T(n)$ equals $O(n)$: $7n + 10 \leq 8n$ for $n \geq 10$. Also, $T(n)$ equals $\Omega(n)$: $7n + 10 \geq 7n$. Thus, $T(n)$ equals $\Theta(n)$.

As another example, suppose $T(n) = 2n^3+5n^2+n$. Then, $T(n)$ equals $O(n^3)$ because $2n^3+5n^2+n \leq 3n^3$ for $n \geq 5$. Similarly, $T(n)$ equals $\Omega(n^3)$ because $2n^3 + 5n^2 + n \geq 2n^3$. Hence, $T(n)$ equals $\Theta(n^3)$.

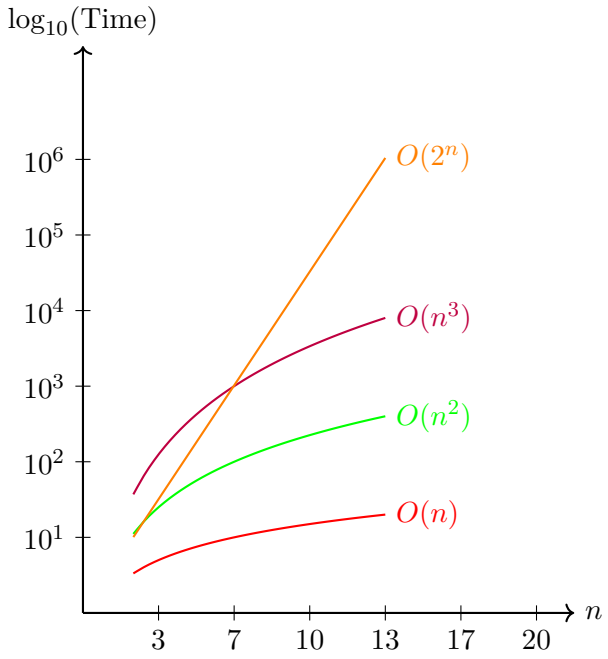


Figure 1.2: Comparison of Growth Rates for $n = 1$ to 20: Linear, Quadratic, Cubic, and Exponential (Log Scale)

1.3 Analyzing Algorithm Efficiency

Efficiency analysis evaluates time and space complexity across best, worst, and average cases, providing a holistic view of performance. We start with the Algorithm LinearSearch.

Algorithm LinearSearch: LinearSearch(A, key)

Input : Array A of size n, key to find
Output: Index of key or -1 if not found

```
1 for i = 0 to n − 1 do
2 |   if A[i] = key then return i ;
3 end
4 return -1
```

The time complexity of the algorithm is as follows.

- *Best*: $O(1)$, key at $A[0]$.
- *Worst*: $O(n)$, key absent.
- *Average*: $\frac{n+1}{2} = O(n)$, assuming uniform likelihood.

We now consider the example of a *nested loop* shown in Algorithm NestedSum. The time complexity

Algorithm NestedSum: NestedSum(n)

```
   Input  : Integer n
   Output: Sum of products
1  total ← 0 ;
2  for i = 0 to n − 1 do
3  |  for j = 0 to n − 1 do
4  |  |  total ← total + i · j ;
5  |  end
6  end
7  return total
```

of the algorithm is $O(n^2)$. The space complexity of the algorithm is $O(1)$. The time and space complexity will always refer to the *worst case* unless specified otherwise.

Let us consider another example in which our goal is to find the number of pairs in an array A that sum to the provided target sum. The reader should verify that the time complexity of the algorithm is $O(n^2)$.

Algorithm PairSum: PairSum(A, target)

```
   Input  : Array A of size n, target sum
   Output: Number of pairs summing to target
1  count ← 0 ;
2  for i = 0 to n − 1 do
3  |  for j = i + 1 to n − 1 do
4  |  |  if A[i] + A[j] = target then  count ← count + 1 ;
5  |  end
6  end
7  return count
```

We now give an example of an algorithm BinarySearch that speeds up Algorithm LinearSearch. We see the recurrence $T(n) = T(n/2) + 1$. Later in this book, we show that $T(n) = O(\log n)$.

As another example, we describe the problem called Towers of Hanoi. The Towers of Hanoi problem involves three pegs and n disks of increasing size, initially stacked on one peg in decreasing size order. The goal is to move the entire stack to another peg, obeying two rules: only one disk can be moved at a time, and a larger disk cannot be placed on a smaller disk. The solution requires $2^n - 1$ moves, achieved recursively by moving $n - 1$ disks to a spare peg, transferring the largest disk, then moving the $n - 1$

Algorithm BinarySearch: BinarySearch(A, key, low, high)

Input : Array A, key, range $[low, high]$
Output: Index of key or -1
1 **if** $low > high$ **then return** *-1* ;
2 $mid \leftarrow \lfloor (low + high)/2 \rfloor$;
3 **if** $A[mid] = key$ **then return** mid ;
4 **else if** $A[mid] > key$ **then return** $BinarySearch(A, key, low, mid - 1)$;
5 **else return** $BinarySearch(A, key, mid + 1, high)$;

disks onto the largest. Algorithm Hanoi shows a recursive solution to this problem. Here the recurrence is: $T(n) = 2T(n-1) + 1$, $T(1) = 1$. On solving this recurrence, we get $T(n) = 2^n - 1$.

Algorithm Hanoi: Hanoi(n, source, aux, target)

Input : Number of disks n, pegs
1 **if** $n = 1$ **then** Move disk from *source* to *target* ;
2 **else**
3 | Hanoi$(n-1, source, target, aux)$;
4 | Move disk from *source* to *target* ;
5 | Hanoi$(n-1, aux, source, target)$;
6 **end**

Finally, we consider the problem of sorting an array A using a method called *MergeSort*. The algorithm uses recursion to sort an array by using a method called divide and conquer. In this example, we have the recurrence $T(n) = 2T(n/2) + n$. We later show that, the recurrence can be solved to determine that $T(n) = O(n \log n)$.

Algorithm MergeSort: MergeSort(A, low, high)

Input : Array A, range $[low, high]$
1 **if** $low < high$ **then**
2 | $mid \leftarrow \lfloor (low + high)/2 \rfloor$;
3 | MergeSort(A, low, mid) ;
4 | MergeSort$(A, mid + 1, high)$;
5 | Merge$(A, low, mid, high)$;
6 **end**

1.4 Common Data Structures: Lists, Stacks, Queues, Binary Trees

This section briefly describes four fundamental data structures: linked lists, stacks, simple queues, and binary search trees, focusing on their main methods and time complexities.

Linked Lists

A linked list is a linear sequence of nodes, where each node contains data and a reference to the next node.

- **Insert** (at head/tail): $O(1)$
- **Delete** (given node): $O(1)$ with pointer; $O(n)$ to find
- **Search**: $O(n)$

Stacks

A stack is a Last-In-First-Out (LIFO) structure, typically implemented with an array or linked list.

- **Push** (add to top): $O(1)$
- **Pop** (remove from top): $O(1)$
- **Top** (access top): $O(1)$

Simple Queues

A simple queue is a First-In-First-Out (FIFO) structure, often using an array or linked list.

- **Enqueue** (add to rear): $O(1)$
- **Dequeue** (remove from front): $O(1)$ with linked list; $O(n)$ with array
- **Front** (access front): $O(1)$

Binary Search Trees

A binary search tree (BST) is a binary tree where each node's left subtree has smaller values, and the right subtree has larger values.

- **Insert**: $O(h)$ (height h; $O(\log n)$ if balanced)
- **Delete**: $O(h)$ (height h; $O(\log n)$ if balanced)
- **Search**: $O(h)$ (height h; $O(\log n)$ if balanced)

1.5 Heaps

A heap is a complete binary tree satisfying the heap property: in a min-heap, each parent's value is less than or equal to its children's values. We focus on min-heaps here, implemented as arrays.

Structure and Representation

Figures 1.3 and 1.4 show a min-heap with values 2, 4, 3, 7, 5.

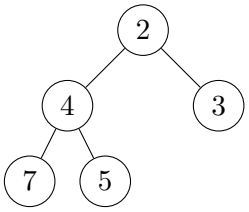


Figure 1.3: Min-heap as a binary tree

2	4	3	7	5
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$

Figure 1.4: Min-heap as an array

Heapify

The Heapify algorithm ensures the min-heap property at a given node by comparing it with its children and swapping with the smallest if necessary, then recursively applying the process downward. It takes an array A, an index i, and heap size n, running in $O(\log n)$ time due to the tree's height.

Algorithm Heapify: Heapify (Min-Heap)

Input: Array A, index i, heap size n

```
1 smallest ← i
2 left ← 2i + 1
3 right ← 2i + 2
4 if left < n and A[left] < A[smallest] then smallest ← left ;
5 if right < n and A[right] < A[smallest] then smallest ← right ;
6 if smallest ≠ i then
7 |   Swap A[i] and A[smallest]
8 |   Heapify(A, smallest, n)
9 end
```

Insert

The Insert algorithm adds a new value to the heap by placing it at the end of the array and sifting it up to its correct position, comparing it with its parent until the min-heap property is restored. It runs in $O(\log n)$ time, as it traverses up the tree's height.

Extract-Min

The Extract-Min algorithm removes the minimum element (root) by replacing it with the last element, reducing the heap size, and then calling Heapify to restore the min-heap property. It operates in $O(\log n)$

Algorithm HeapInsert: Heap-Insert

Input: Array A, value key, heap size n

```
1 n ← n + 1
2 A[n − 1] ← ∞
3 i ← n − 1
4 while i > 0 and A[⌊(i − 1)/2⌋] > key do
5 |   A[i] ← A[⌊(i − 1)/2⌋]
6 |   i ← ⌊(i − 1)/2⌋
7 end
8 A[i] ← key
```

time due to the heapification step.

Algorithm HeapExtractMin: Heap-Extract-Min

Input: Array A, heap size n

```
1 if n < 1 then return error ;
2 min ← A[0]
3 A[0] ← A[n − 1]
4 n ← n − 1
5 Heapify(A, 0, n)
6 return min
```

Build-Heap

The Build-Heap algorithm constructs a min-heap from an unsorted array by iteratively applying Heapify from the last non-leaf node up to the root. It runs in $O(n)$ time, as the cumulative effect of heapifying all subtrees is linear in the number of nodes.

Algorithm BuildHeap: Build-Heap (Min-Heap)

Input: Array A, size n

```
1 for i ← ⌊n/2⌋ − 1 downto 0 do
2 |   Heapify(A, i, n)
3 end
```

The time complexity of various operations on a heap can be summarized as follows.

- **Heapify**: $O(\log n)$
- **Insert**: $O(\log n)$
- **Extract-Min**: $O(\log n)$
- **Build-Heap**: $O(n)$

1.6 Organization of the Book

Chapter 2 presents the stable marriage problem. Chapter 3 presents the general Lattice Linear Predicate algorithm. Chapter 2 shows how the stable marriage problem can be solved using the LLP algorithm. In particular, the chapter gives a more general algorithm than the Gale-Shapley algorithm for the stable marriage. Chapter 8 discusses various sorting algorithms. A sorting algorithm can be viewed as searching for the appropriate permutation of the given array. Chapter 10 discusses various dynamic programming problems. These include the longest increasing subsequence problem, the optimal binary search tree problem and the knapsack problem. All these problems are shown to be solvable using the LLP algorithm. Chapter 4 discusses basic graph algorithms. These include various traversal techniques and the topological sort of a graph. Chapter 6 discusses various algorithms for computing the shortest path in a graph. Chapter 7 describes algorithms for computing the minimum spanning tree in a weighted undirected graph. By considering different lattices on the set of edges, one can derive classical Kruskal and Prim's algorithms. Chapter 14 discusses the housing allocation problem which is a one-sided version of the stable marriage problem. Chapter 15 discusses the assignment problem.

1.7 Problems

1. Please prove or disprove each of the following:

   (a) $3^n = O(2^n)$

   (b) If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$

   (c) If $f(n) = O(g(n))$, then $f(n)^3 = O(g(n)^3)$

2. Prove the following statements.

   (a) Prove that for any constant $d > 0$, $f(n) = O(d \cdot g(n))$ iff $f(n) = O(g(n))$.

   (b) Prove that if $f(n) = O(h_1(n))$ and $g(n) = O(h_2(n))$ then $f(n) + g(n) = O(\max\{h_1(n), h_2(n)\})$.

   (c) Prove that all logarithms are asymptotically equivalent. That is, prove that $\log_a(n) = O(\log_b(n))$ for any positive a and b.

3. Suppose that you have a min Heap of size n.
   (a) Give the pseudo-code to find and delete the smallest element in the Heap.
   What is the time complexity of your algorithm in terms of n ?
   (b) Give the pseudo-code to find and delete the k^{th} smallest element in the heap. What is the time complexity of your algorithm in terms of n and k ? Assume that you have a method available to you for a Heap that can insert any element in the heap in $O(\log n)$ time.

4. Suppose that we have an array A of n distinct integers in two forms: a min Heap H and a max heap K. We are given the following functions on the min-heap:
   $H.getMin()$ // return the least value in the heap H
   $H.deleteMin()$ // delete the least value from the heap H
   Similarly, we have analogous methods for the max-heap.

   Give pseudo code of an efficient program to return a value $A[i]$ in the array A such that there exists j where $A[i] + A[j]$ equals m. If there are no two such entries, then the program should return null.

   Give the tight asymptotic time complexity of your method.

1.8 Bibliographic Remarks

Heaps were introduced by J.W.J. Williams in 1964 as part of the heapsort algorithm, which leverages the heap structure for efficient sorting in $O(n \log n)$ time. The linear-time Build-Heap construction was later analyzed by Floyd, showing its $O(n)$ complexity. Heaps have since become foundational in priority queue implementations, with applications in scheduling, graph algorithms (e.g., Dijkstra's), and data compression (e.g., Huffman coding).

Chapter 2

The Stable Marriage Problem

2.1 Introduction

The Stable Matching Problem (SMP) introduced by Gale and Shaply [GS62] has wide applications in economics, distributed computing, resource allocation and many other fields. In the standard SMP, there are n men and n women each with their totally ordered preference list. The goal is to find a matching between men and women such that there is no instability, i.e., there is no pair of a woman and a man such that they are not married to each other but prefer each other over their partners.

We consider stable marriage instances with m men numbered $1, 2, \ldots, m$ and w women numbered $1, 2, \ldots w$. We assume that the number of women w is at least m; otherwise, the roles of men and women can be switched. The variables $mpref$ and $wpref$ specify the men preferences and the women preferences, respectively. Thus, $mpref[i][k] = j$ iff woman j is k^{th} preference for man i. Fig. 2.1 shows an instance of the stable matching problem.

$mpref$				$wpref$			
m_1	w_2	w_3	w_1	w_1	m_1	m_3	m_2
m_2	w_1	w_2	w_3	w_2	m_2	m_1	m_3
m_3	w_3	w_1	w_2	w_3	m_1	m_2	m_3

Figure 2.1: Stable Matching Problem with men preference list ($mpref$) and women preference list ($wpref$).

In this chapter, we first give the Gale-Shapley (GS) algorithm in Section 2.3. Section 2.4 presents Algorithm α that takes any proposal vector (and therefore any matching) to a stable proposal vector in $O(m^2 + w)$ time such that the resulting proposal vector has the least distance from the initial proposal vector of all stable proposal vectors that are greater than initial proposal vector. This algorithm generalizes the GS algorithm which assumes the initial proposal vector to be the top choices.

2.2 Proposal Vector Lattice

We use the notion of a *proposal vector* for our algorithms. A (man) proposal vector, G, is of dimension m, the number of men. We view any vector G as follows: $(G[i] = k)$ if man i has proposed to his k^{th}

preference, i.e. the woman given by $mpref[i][k]$. If $mpref[i][k]$ equals j, then $G[i]$ equals k corresponds to man i proposing to woman j. For convenience, let $\rho(G, i)$ denote the woman $mpref[i][G[i]]$. The vector $(1, 1, \ldots, 1)$ corresponds the proposal vector in which every man has proposed to his top choice. Similarly, $(w, w, \ldots, w)$ corresponds to the vector in which every man has proposed to his last choice. Our algorithms can also handle the case when the lists are incomplete, i.e., a man prefers staying alone to being matched to some women. However, for simplicity, we assume complete lists. It is clear that the set of all proposal vectors forms a distributive lattice under the natural less than order in which the meet and join are given by the component-wise minimum and the component-wise maximum, respectively. This lattice has w^m elements.

Given any proposal vector, G, there is a unique matching defined as follows: man i and $\rho(G, i)$ are matched in G if the proposal by man i is the best for that woman in G. A man p is unmatched in G if his proposal is not the best proposal for that woman in G. A woman q is unmatched in G if she does not receive any proposal in G; otherwise, she is matched with the best proposal for her in G.

A proposal vector G represents a *man-saturating matching* iff no woman receives more than one proposal in G. Formally, G is a man-saturating matching if $\forall i, j : i \neq j : \rho(G, i) \neq \rho(G, j)$. When the number of men equals the number women, a man-saturating matching is a perfect matching (all men and women are matched). When the number of men is less than the number of women, then G is a man-saturating matching if every man is matched (but some women are unmatched). We say that a matching M_1 (or a marriage) is less than another matching M_2 if the proposal vector for M_1 is less than that of M_2. Thus, the man-optimal marriage is the *least* stable matching in the proposal lattice and woman-optimal marriage is the *greatest* stable matching.

A proposal vector G may have one or more blocking pairs. A pair of man and woman (p, q) is a *blocking pair* in G iff $\rho(G, p)$ is not q, man p prefers q to $\rho(G, p)$, and woman q prefers p to any proposal she receives in G. Observe that this definition works even when woman q is unmatched, i.e. she has not received any proposals in G. In this case, woman q prefers p to staying alone, and p prefers q to $\rho(G, p)$.

A proposal vector G is a stable marriage (or a stable proposal vector) iff it is a man-saturating matching and there are no blocking pairs in G.

2.3 Gale-Shapley Algorithm

In this section, we first present an algorithm due to Gale and Shapley for this problem (also known as the deferred-acceptance algorithm). In this algorithm, a free man proposes to women in his decreasing order of preferences, i.e., he first proposes to his top choice. Only when a man is rejected by his top choice, he would move to his next top choice. We maintain the list of all men who are not engaged in the variable $mList$. Initially all men are free (not engaged) and are in this list.

When any woman z receives a proposal from a man i, she always accepts it if she is not engaged. In this case, they both get engaged and the variable $partner[z]$ is set to i. If the woman z is engaged then she compares this proposal to her existing partner. If she prefers this proposal to her existing partner, then she breaks the engagement with her existing partner and makes i as her partner. The previous partner joins $mList$, the list of free men. If the woman z prefers her existing partner, then the proposal by man i is rejected and the man i goes back to $mList$. Algorithm Gale-Shapley shows the pseudo-code for these steps.

Algorithm Gale-Shapley: Finding the man-optimal marriage

```
1  input: mpref, rank
2  output: man-optimal stable marriage
3  var
4      mList: list of 1..n // list of men that are free, initially includes all men
5      partner: array[1..n] of 0..n initially partner[i] = 0 for all i // Current fiance for woman i
6      G: array[1..n] of 0..n initially G[i] = 0 for all i// Number of proposals made by man i
7  mList := [1..n] // Initialize all men as free
8  while mList is nonempty do
9  |   i := mList.removeFirst()
10 |   G[i] := G[i] + 1 // Move to the next available top choice for man i
11 |   z := mpref[i][G[i]] // Woman corresponding to that choice
12 |   if partner[z] = 0 then partner[z] := i;
13 |   else if rank[z][i] < rank[z][partner[z]] then
14 |   |   mList.add(partner[z])
15 |   |   partner[z] := i
16 |   end
17 |   else  mList.add(i) ;
18 end
19 return G
```

It is easy to verify the following properties of the algorithm.

Lemma 2.1 *1. As the algorithm progresses, the partner for a man can only worsen and the partner for a woman can only improve.*

2. Once a woman is engaged, she stays engaged.

3. If the number of men is less than or equal to the number of women, then the algorithm is guaranteed to terminate.

2.4 Algorithm α: Upward Traversal

Algorithm α generalizes Gale-Shapley algorithm in two fundamental ways.

- *Arbitrary Initial proposal vector*: Gale-Shapley algorithm always finds the man-optimal stable marriage. Suppose that we are interested in finding a stable marriage such that the men are allowed to propose such that the proposal vector is at least I. For example, if $I = (3, 1, 2, 1)$, then the first man cannot propose to his two top choices and the third man cannot propose to his top choice. Algorithm α works even when the initial proposal vector is arbitrary instead of the top choice for each man, i.e., $I = (1, 1, \ldots, 1)$. Observe that standard Gale-Shapley algorithm does not work as is when the starting proposal vector is arbitrary. Simple Gale-Shapley algorithm would require men to make proposals and women to accept best proposals they have received so far. If the starting proposal vector is a perfect matching but not stable, then each woman gets a unique proposal. All women would accept the only proposal received, but the resulting marriage would may not be stable.

- *Unequal number of men and women*: We consider the case when the number of women exceeds the number of men. If the number of men is greater than we can simply switch the roles in our algorithm.

  If we started with the top choices of all men, then Gale-Shapley algorithm would still return a man-optimal stable matching with the excess women unmatched. However, if we start from an arbitrary proposal vector, we can end up with all women getting unique proposals but there may exist an unmatched woman who is preferred by some man over his current match. To tackle this problem, we first do a simple check on the initial proposal vector as given by the following Lemma. Let $numw(I)$ denote the total number of unique women that have been proposed in all vectors that are less than or equal to I.

  Lemma 2.2 *Given any stable marriage instance with m men and the initial proposal vector I there is no stable marriage for any proposal vector $G \geq I$ whenever $numw(I)$ exceeds m.*

  Proof: Consider any proposal vector $G \geq I$. Since the total number of men is m, there is at least one woman q who has been proposed in the past of G who does not have any proposal in G. Suppose that proposal was made by man p. Then, man p prefers q to $\rho(G, p)$ and q prefers p to staying alone.

  ■

  Hence, in our algorithm we only consider I such that the total number of women proposed until I is at most m.

Even when the number of men and women are equal, the proposal vector may be a perfect matching but not stable. To address this problem, we first define $forbidden(G, i)$ as the predicate that there exists another man j such that either (1) both i and j have proposed to the same woman in G and that woman prefers j, or (2) $(j, \rho(G, i))$ is a blocking pair in G.

We first show that

Lemma 2.3 *Let G be any proposal vector such that $numw(G) \leq m$. There exists a man i such that $forbidden(G, i)$ iff G is not a stable marriage.*

Proof: First suppose that there exists i such that $forbidden(G, i)$. This mean that there exists a man j such that j has proposed to the same woman and that woman prefers j or $(j, \rho(G, i))$ is a blocking pair in G. If both i and j have proposed to the same woman in G, then it is clearly not a matching. If $(j, \rho(G, i))$ is a blocking pair then G is not stable.

Conversely, assume that G is not a stable marriage. This means that either G is not a perfect matching or there is a blocking pair for G. If it is not a perfect matching, then there must be some woman who has been proposed by multiple men. Any man i who is not the most favored in the set of proposals satisfies $forbidden(G, i)$. If G is a perfect matching but not a stable marriage, then there must be a blocking pair (p, q). If q has been proposed in G by man i, then $(p, \rho(G, i))$ is a blocking pair. If q has not been proposed in G then we know at least $m + 1$ women that are in $numw(G)$ which violates our requirement on G.

■

Algorithm UpwardTraversal exploits the $forbidden(G, i)$ function to search for the stable marriage in the proposal lattice. The basic idea is that if a man i is forbidden in the current proposal vector G, then he must go down his preference list until he finds a woman who is either unmatched or prefers him to her

current match. The while loop at line (1) iterates until none of the man is forbidden in G. If the while loop terminates then G is a stable marriage on account of Lemma 2.3. At line (2), man i advances on his preference list until his proposal is the most preferred proposal to the woman among all proposals that are made to her in any proposal vector less than or equal to G. If there is no such proposal, then there does not exist any $G \geq I$ such that G is stable and in line (3), the algorithm returns null. Otherwise, the man makes that proposal at line (4).

Algorithm UpwardTraversal: An Algorithm that returns the man-optimal marriage greater than or equal to the given proposal vector I.

```
1  input: A stable marriage instance, initial proposal vector I
2  output: smallest stable marriage greater than or equal to I (if one exists)
3  forbidden(G, i) holds if there exists another man j such that either (1) both i and j have proposed
     to the same woman in G and that woman prefers j, or (2) (j, ρ(G, i)) is a blocking pair for G.
4  if numw(I) > m then return null; // no stable matching exists
5        else G := I;
6  while there exists a man i such that forbidden(G, i)
7        find the next woman q in the list of man i s.t. i has the most preferred proposal to q until G,
8        if no such choice after G[i] or the number of women proposed including q exceeds m then
9              return null; // "no stable matching exists"
10       else G[i] := choice that corresponds to woman q;
11 endwhile;
12 return G;
```

2.5 Problems

1. Show that the Gale-Shapley algorithm always finds a stable marriage.

2. Give an instance of the stable marriage problem in which there is only one stable marriage: the last choice for all men.

3. Show that the set of stable marriages is closed not only under meet, but also the join operation. Thus, the set of stable marriages form a sublattice of the proposal lattice.

4. Give an algorithm in which all men start with their least favored choice and improve their choices in the search of the stable marriage.

5. Given an assignment vector, how will you check whether it forms a stable marriage?

2.6 Bibliographic Remarks

The stable matching problem has been studied extensively with multiple books and survey articles devoted on the topic [GI89, Knu97, RS92, IM08, Man13].

Chapter 3

Lattice Linear Predicate Detection

3.1 Introduction

In this chapter we discuss a general method called Lattice-Linear Predicate (LLP) algorithm that can be used to solve a wide variety of problems including the stable marriage problem, the shortest path problem, the assignment problem, the minimum spanning tree problem and the housing allocation problem.

Section 3.2 defines lattice-linear predicates formally. These predicates are defined on a distributive lattice. The problem is framed as searching for an element in the lattice that satisfies the predicate. Generally, we are interested in the least element in the lattice that satisfies the predicate. Section 3.3 gives the notation that we use for programs based on lattice-linear predicates. It specifies the lattice that we are working on, the starting element in the lattice, the top element of the lattice and the predicate *forbidden* which allows us to advance in the lattice. Section 3.4 lists some desirable properties of the LLP algorithm. The algorithm is naturally *nondeterministic*. There are multiple ways that the algorithm can advance in the lattice. The algorithm is *online*. It does not inspect any element higher than the current element of the lattice that is under consideration. Finally, the algorithm returns an answer that is optimal for all components.

3.2 Lattice-Linear Predicates

Let L be the lattice of all n-dimensional vectors of reals greater than or equal to zero vector and less than or equal to a given vector T where the order on the vectors is defined by the component-wise natural $\leq$. The minimum element of this lattice is the zero vector. The lattice is used to model the search space of the combinatorial optimization problem. For simplicity, we are considering the lattice of vectors of non-negative reals; later we show that our results are applicable to any distributive lattice. The combinatorial optimization problem is modeled as finding the minimum element in L that satisfies a boolean *predicate* B, where B models *feasible* (or acceptable solutions). We are interested in parallel algorithms to solve the combinatorial optimization problem with n processes. We will assume that the systems maintains as its state the current candidate vector $G \in L$ in the search lattice, where $G[i]$ is maintained at process i. We call G, the global state, and $G[i]$, the state of process i.

Finding an element in lattice that satisfies the given predicate B, is called the *predicate detection* problem. Finding the *minimum* element that satisfies B (whenever it exists) is the combinatorial optimization problem. We now define *lattice-linearity* which enables efficient computation of this minimum element. A key concept in deriving an efficient predicate detection algorithm is that of a *forbidden* state. Given a predicate B, and a vector $G \in L$, a state $G[i]$ is *forbidden* (or equivalently, the index i is forbidden) if for any vector $H \in L$, where $G \leq H$, if $H[i]$ equals $G[i]$, then B is false for H. Formally,

Definition 3.1 (Forbidden State) *Given any distributive lattice L of n-dimensional vectors of $\mathbf{R}_{\geq 0}$, and a predicate B, we define $forbidden(G, i, B) \equiv \forall H \in L : G \leq H : (G[i] = H[i]) \Rightarrow \neg B(H)$.*

We define a predicate B to be *lattice-linear* with respect to a lattice L if for any global state G, B is false in G implies that G contains a *forbidden state*. Formally,

Definition 3.2 (lattice-linear Predicate) *A boolean predicate B is* lattice-linear *with respect to a lattice L iff $\forall G \in L : \neg B(G) \Rightarrow (\exists i : forbidden(G, i, B))$.*

We now give some examples of lattice-linear predicates.

1. **Job Scheduling Problem**: Our first example relates to scheduling of n jobs. Each job j requires time t_j for completion and has a set of prerequisite jobs, denoted by $pre(j)$, such that it can be started only after all its prerequisite jobs have been completed. Our goal is to find the minimum completion time for each job. We let our lattice L be the set of all possible completion times. A completion vector $G \in L$ is feasible iff $B_{jobs}(G)$ holds where $B_{jobs}(G) \equiv \forall j : (G[j] \geq t_j) \wedge (\forall i \in pre(j) : G[j] \geq G[i] + t_j)$. B_{jobs} is lattice-linear because if it is false, then there exists j such that either $G[j] < t_j$ or $\exists i \in pre(j) : G[j] < G[i] + t_j$. We claim that $forbidden(G, i, B_{jobs})$. Indeed, any vector $H \geq G$ cannot be feasible with $G[j]$ equal to $H[j]$. The minimum of all vectors that satisfy feasibility corresponds to the minimum completion time.

2. **Continuous Optimization Problem**: We are required to find minimum nonnegative x and y such that $B \equiv (x \geq y^2/4 + 5) \wedge (y \geq x - 4)$. We view this problem as finding minimum (x, y) pair such that B holds. It is easy to verify that B is lattice-linear. If the first conjunct is false, then x is forbidden. Unless x is increased the predicate cannot become true, even if other variables (y for this example) increase. If the second conjunct is false, then y is forbidden. In this case, $x = 6$ and $y = 2$ is the pointwise minimum solution. For some predicates, there may not be any solution. For example, when $B \equiv (x \geq 2y^2 + 5) \wedge (y \geq x - 4)$, there is no nonnegative (x, y) pair such that B holds (verify this!). The predicate B is still lattice-linear but by advancing along x and y we go beyond any bounded (x, y).

3. **A Non Lattice-Linear Predicate** As an example of a predicate that is not lattice-linear, consider the predicate $B \equiv \sum_j G[j] \geq 1$ defined on the space of two dimensional vectors. Consider the vector G equal to $(0, 0)$. The vector G does not satisfy B. For B to be lattice-linear either the first index or the second index should be forbidden. However, none of the indices are forbidden in $(0, 0)$. The index 0 is not forbidden because the vector $H = (0, 1)$ is greater than G, has $H[0]$ equal to $G[0]$ but it still satisfies B. The index 1 is also not forbidden because $H = (1, 0)$ is greater than G, has $H[1]$ equal to $G[1]$ but it satisfies B.

The following Lemma is useful in proving lattice-linearity of predicates.

Lemma 3.3 *Let B be any boolean predicate defined on a lattice L of vectors.*
(a) Let $f : L \rightarrow \mathbf{R}_{\geq 0}$ be any monotone function defined on the lattice L of vectors of $\mathbf{R}_{\geq 0}$. Consider the predicate $B \equiv G[i] \geq f(G)$ for some fixed i. Then, B is lattice-linear.
(b) Let L_B be the subset of the lattice L of the elements that satisfy B. Then, B is lattice-linear iff L_B is closed under meets.
(c) If B_1 and B_2 are lattice-linear then $B_1 \wedge B_2$ is also lattice-linear.

For the job scheduling example, we can define B_j as $G[j] \geq max(t_j, max\{G[i] + t_j \mid i \in pre(j)\})$. Since $f_j(G) = max(t_j, max\{G[i] + t_j \mid i \in pre(j)\})$ is a monotone function, it follows from Lemma 3.3(a) that B_j is lattice-linear. The predicate $B_{jobs} \equiv \forall j : B_j$ is lattice-linear due to Lemma 3.3(c). Also note that the problem of finding the minimum vector that satisfies B_{jobs} is well-defined due to Lemma 3.3(b).

3.3 Notation

We first go over the notation used in description of our parallel algorithms. Fig. 3.1 shows parallel algorithms for the job-scheduling and the shortest path problems. We have a single variable G in all the examples shown in Fig. 3.1.

All other variables are derived directly or indirectly from G. There are three sections of the program.

The **init** section is used to initialize the state of the program. All the parts of the program are applicable to all values of j. For example, the *init* section of the job scheduling program in Fig. 3.1 specifies that $G[j]$ is initially $t[j]$. Every thread j would initialize $G[j]$.

The **always** section defines additional variables which are derived from G. The actual implementation of these variables are left to the system. They can be viewed as macros.

The third section gives the desirable predicate either by using the **forbidden** predicate or **ensure** predicate. The *forbidden* predicate has an associated *advance* clause that specifies how $G[j]$ must be advanced whenever the forbidden predicate is true. For many problems, it is more convenient to use the complement of the forbidden predicate. The *ensure* section specifies the desirable predicates of the form $(G[j] \geq expr)$ or $(G[j] \leq expr)$. The statement *ensure* $G[j] \geq expr$ simply means that whenever thread j finds $G[j]$ to be less than $expr$; it can advance $G[j]$ to $expr$. Since $expr$ may refer to G, just by setting $G[j]$ equal to $expr$, there is no guarantee that $G[j]$ continues to be equal to $expr$ — the value of $expr$ may change because of changes in other components. We use *ensure* statement whenever $expr$ is a monotonic function of G and therefore the predicate is lattice-linear.

3.4 Properties of the LLP Algorithm

The LLP algorithm has many useful properties. We list them here so that the reader can apply them to various problems studied in this book.

These properties are applicable to all the problems for which the LLP algorithm is used.

1. **Nondeterminism in Evaluation of Forbidden Predicate**: Given a global state G, there may be multiple indices j for which $G[j]$ is forbidden. The LLP algorithm is correct irrespective of the order in which these indices are updated. The efficiency of the algorithms may differ depending upon the order in which these indices are updated, but the correctness is independent of the order. For example, in the stable marriage problem, the final answer returned is independent of the order in which men propose. In the shortest path problem, the final answer returned is independent of the

P_j: Code for thread j
// common declaration for all the programs below
shared var G: array[$1..n$] of $0..maxint$;
job-scheduling:
    input: $t[j] : int$, $pre(j)$: list of $1..n$;
    init: $G[j] := t[j]$;
    forbidden: $G[j] < \max\{G[i] + t[j] \mid i \in pre(j)\}$;
    advance: $G[j] := \max\{G[i] + t[j] \mid i \in pre(j)\}$;

job-scheduling:
    input: $t[j] : int$, $pre(j)$: list of $1..n$;
    init: $G[j] := t[j]$;
    ensure: $G[j] \geq \max\{G[i] + t[j] \mid i \in pre(j)\}$;

shortest path from node s: Parallel Bellman-Ford
    input: $pre(j)$: list of $1..n$; $w[i, j]$: int for all $i \in pre(j)$
    init: if $(j = s)$ then $G[j] := 0$ else $G[j] :=$ maxint;
    ensure: $G[j] \leq \min\{G[i] + w[i, j] \mid i \in pre(j)\}$

Figure 3.1: LLP Parallel Program for (a) job scheduling problem using forbidden predicate (b) job scheduling problem using ensure clause and (c) the shortest path problems

order in which the nodes update their estimates. It is important to note that the term *answer* is with respect to the LLP algorithm. For the shortest path problem, the lattice is with respect to the cost and not the actual paths. There may be multiple shortest paths to a vertex and the order of evaluation of forbidden indices may result in the paths that are returned be different. However, they would all have the same cost.

2. **Parallel Evaluation of Forbidden Predicate**: Suppose that G is shared among different threads such that thread j is responsible for evaluating $forbidden(G, j)$. While this thread is evaluating this predicate other threads may have advanced on other indices, i.e., thread j may have old information of $G[i]$ for $i \neq j$. However, this would still keep the algorithm correct. For example, in the stable marriage problem, men can propose to women in parallel. In the shortest path algorithm, multiple vertices can update the estimate of their lower bounds and their parents in parallel.

3. **No Lookahead Required for evaluation of Forbidden Predicate**: The LLP algorithm determines whether an index j is forbidden depending upon only the current global state G. This means that these algorithms are applicable in online settings where the future part of the lattice is revealed only when a forbidden index needs to advance. For example, in the stable marriage problem, when we are computing the man-optimal stable marriage, a man may not reveal his preference list. Only when he is rejected (his state is forbidden), he needs to advance on his choices and therefore reveal the next woman on his list. The same reasoning applies to the housing allocation problem. For some problems, such as the shortest path problem, the weights on the edges is required to evaluate forbidden predicate. Therefore, the graph must be known and static for the LLP algorithm.

4. **The Optimal Feasible Global State is Optimal for Individual Nodes**: Suppose that for the stable marriage problem, one of the men, say m_1 is interested in finding the stable marriage in which

he has the topmost choice possible for him. This man does not care how other men are paired. If we take the solution derived from the LLP algorithm, then the woman m_1 is paired with would be identical to the woman m_1 is paired with in any stable marriage that is optimal from just the perspective of m_1. More generally, let G be the global state that is feasible and optimal with respect to index i, i.e., for all feasible H, $G[i] \leq H[i]$. Let G_{llp} be the global state computed by the LLP algorithm, then $G[i] = G_{llp}[i]$. This follows from the meet-closure property of feasible states. If $G[i] < G_{llp}[i]$, then the global state given by $G \sqcap G_{llp}$ is also feasible and strictly smaller than G_{llp} contradicting that G_{llp} is the least global state that satisfies the feasibility predicate.

3.5 An LLP Algorithm for the Stable Matching Problem

We now derive the algorithm for the stable matching problem using Lattice-Linear Predicates. We let $G[i]$ be the choice number that man i has proposed to. Initially, $G[i]$ is 1 for all men. For convenience, let $\rho(G, i)$ denote the woman $mpref[i][G[i]]$.

Definition 3.4 *An assignment G is feasible for the stable marriage problem if (1) it corresponds to a perfect matching (all men are paired with different women) and (2) it has no blocking pairs.*

We show that the predicate "G is a stable marriage" is a lattice-linear predicate.

Lemma 3.5 *The predicate that a vector G corresponds to a stable marriage is lattice-linear.*

Proof: Let z be $\rho(G, j)$, the woman that corresponds to choice $G[j]$ for man j. We define j to be forbidden in G if there exists a man i such that z prefers man i to man j and either man i has also been assigned z in G or he prefers z to his current choice, i.e., man i and woman z would form a blocking pair in G. Formally, $forbidden(G, j)$ is defined as $(\exists i : \exists k \leq G[i] : (z = mpref[i][k]) \wedge (rank[z][i] < rank[z][j]))$.

It is easy to see that G is not a stable marriage iff $\exists j : forbidden(G, j)$. If G is not a perfect matching then there must be at least one woman who is assigned to two men. In that case, the less preferred man is forbidden. If G is a perfect matching but has a blocking pair, then the partner of the woman in the blocking pair is forbidden. Conversely, $forbidden(G, j)$ implies that either G is not a perfect matching or has a blocking pair.

We only need to show that if $forbidden(G, j)$ holds, then there is no proposal vector H such that $(H \geq G)$ and $(G[j] = H[j])$ and H is a stable marriage.

Consider any H such that $(H \geq G)$ and $(G[j] = H[j])$. We show that H is not a stable marriage. Since $G[j]$ is equal to $H[j]$, $\rho(G, j)$ is equal to $\rho(H, j)$. Let i be such that $\exists k \leq G[i] : (z = mpref[i][k]) \wedge (rank[z][i] < rank[z][j]))$. Since $G \leq H$, $G[i] \leq H[i]$, we get that $\exists k \leq H[i] : (z = mpref[i][k]) \wedge (rank[z][i] < rank[z][j]))$. Hence, $forbidden(H, i)$ also holds.

■

Lemma 3.5 immediately gives us the Algorithm LLP-ManOptimalStableMarriage. The **always** section defines variables which are derived from G. These variables can be viewed as macros. For example, for any thread $z = mpref[j][G[j]]$. This means that whenever $G[j]$ changes, so does z.

If man j is forbidden, it is clear that any vector in which man j is matched with z and man i is matched with his current or a worse choice can never be a stable marriage. Thus, it is safe for man j to advance to the next choice.

Algorithm LLP-ManOptimalStableMarriage: A Parallel Algorithm for Stable Matching

```
1 P_j: Code for thread j
2 input: mpref[i,k]: int for all i,k; rank[k][i]: int for all k,i;
3 init: G[j] := 1; // works for any initialization
4 always: z = mpref[j][G[j]];
5 forbidden: (∃i : ∃k ≤ G[i] : (z = mpref[i][k]) ∧ (rank[z][i] < rank[z][j]))
6     advance: G[j] := G[j] + 1;
```

Algorithm LLP-ManOptimalStableMarriage works for any initialization of G vector. If we know that G is initialized to $[1, 1, \ldots, 1]$, then we can simplify the forbidden condition to

$$(\exists i : (z = mpref[i][G[i]]) \wedge (rank[z][i] < rank[z][j])).$$

Observe that if we assume sequential implementation and if we implement a variable $partner[z]$ for any woman z, we can further simplify the condition to

$$(partner[z] \neq null) \wedge (rank[z][partner[z]] < rank[z][j]).$$

This is precisely the condition used in Gale-Shapley algorithm which is sequential and starts with the initial vector $[1, 1, \ldots, 1]$.

We now present an algorithm to find stable marriages that satisfy additional constraints. The following lemma proves lattice-linearity of many such constraints.

Lemma 3.6 *The following constraints are lattice-linear.*

1. *The regret of man i is at most that of the regret of man j.*
2. *Man i cannot be married to woman j.*
3. *The regret of man i is equal to that of man j.*

Proof: Let B be the predicate that G is a stable marriage and it satisfies the corresponding additional constraint.

1. Suppose that G is a stable marriage but it does not satisfy B. This means that regret of man i is more than the regret of man j. In this case, we have $forbidden(G, j)$, because unless $G[j]$ is advanced, the predicate cannot become true.
2. If $\rho(G, i) = j$, then $forbidden(G, i)$ holds.
3. This condition is a conjunction of two lattice-linear conditions of type in part (1).

■

We now enumerate advantages of the LLP algorithm.

1. It gives us a more general algorithm. Instead of starting from the initial vector, we can start from any vector and find a stable marriage greater than or equal to that vector, if one exists.

2. We can use the LLP algorithm for finding a stable marriage that satisfies additional constraint such as the regret of man i is at most that of the regret of man j.

3. We can automatically deduce that the intersection of two stable marriages is also a stable marriage.

4. The LLP algorithm is useful in the context of parallel computing. If two men are forbidden, then both of them can advance in parallel.

5. The LLP algorithm is also useful in the context of distributed computing. The man i may not have the most recent information about man j. Even then, the algorithm will terminate with the stable marriage that is the least in the proposal vector lattice.

3.6 Additional LLP Algorithms

For computing GCD of an array of natural numbers A, we consider the lattice of natural numbers less than or equal to the largest value in A. The lattice is ordered decreasingly by componentwise comparison of vectors. The algorithm terminates when all components are equal. Otherwise, the component j is forbidden if it is greater than some component i. In this case, we replace $G[j]$ by $G[i]$ if $G[i]$ divides $G[j]$. Otherwise, we replace $G[j]$ by $G[j]$ *mod* $G[i]$. When the algorithm terminates, all components of G vector qre equal to the greatest common divisor of A.

The Factorial algorithm for LLP considers the lattice of 2-dimensional vectors where the first dimension will finally hold the result. The lattice is ordered in the increasing order by the first component and the decreasing order by the second component. The algorithm initializes $G[2]$ by n and $G[1]$ by 1. Whenever $G[2]$ is greater than 1, we multiply $G[1]$ by $G[2]$ and reduce $G[2]$ by 1. It is clear that in $n-1$ steps, $G[2]$ will become 1 and $G[1]$ will hold $factorial(n)$.

The MaxElement algorithm initializes G with $A[1]$. If for any index j, $A[j]$ is greater than G, we replace G by $A[j]$. For a sequential implementation, we need not traverse all indices j in any particular order.

The LinearSearch algorithm initializes G with -1. The goal is to replace G with the index j of the array A if $A[j]$ equals key. This is a nondeterministic program. If there are multiple j such that $A[j]$ equals key, the final answer could be any one of them.

The NestedSum algorithm is another nondeterministic algorithm. The variable $done[i,j]$ is 0 iff the entry $i * j$ has not been added to the variable G. Whenever all entries inthe array $done$ become 1, the algorithm terminates.

The PairSum algorithm counts the number of tuples (i,j) where $i < j$ and $A[i]$ plus $A[j]$ equals $target$.

Finally, the BinarySearch algorithm returns the index j in $A[j]$ such that $A[j]$ equals key. It assumes that A is sorted. The algorithm continues until the range $[low \ldots high]$ is nonempty. It finds the new range in A for searching based on the comparison of $A[mid]$ to $target$.

3.7 Problems

1. Let G be a n-dimensional vector of positive numbers. Let $pred$ be a binary acyclic relation on $[n]$. Show that the predicate $B \equiv \forall (i,j) \in pred : G[j] \geq G[i] + 1$ is a lattice-linear predicate.

2. Let $(X, \leq)$ be a poset. A subset $Y \subseteq X$ is an order ideal if it satisfies

$$\forall u, v \in X : (v \in Y) \wedge (u \leq v) \Rightarrow (u \in Y)$$

GCD of an array A:
G: array[$1..n$] of int initially $\forall i : G[i] = A[i]$;
forbidden(j): $\exists i : G[j] > G[i]$
advance(j): if $G[i]$ divides $G[j]$ then $G[j] := G[i]$;
else $G[j] := G[j]\ mod\ G[i]$;

Factorial
input: n: non-negative integer;
G: array[$1..2$] of int initially $(G[1] = 1) \wedge (G[2] = n)$;
forbidden: $G[2] > 1$
advance: $G[1] := G[1] * G[2]; G[2] := G[2] - 1$;

MaxElement
input: A: array of integers;
G: int initially $G = A[1]$;
forbidden(j): $A[j] > G$
advance: $G := A[j]$;

LinearSearch
input: A: array of integers; key: integer;
G: int initially $G = -1$;
forbidden(j): $(G == -1) \wedge (A[j] == key)$
advance: $G := j$;

NestedSum
input: A: array of integers;
G: int initially $G = 0$;
$done$: array[$0..n-1$, $0..n-1$] of int initially $\forall i, j : done[i, j] = 0$;
forbidden(i,j): $(done[i, j] == 0)$
advance: $G := G + i * j; done[i, j] := 1$;

PairSum
input: A: array of integers; $target$: integer;
G: int initially $G = 0$;
$done$: array[$0..n-1$, $0..n-1$] of int initially $\forall i, j : done[i, j] = 0$;
forbidden(i,j): $(i < j) \wedge (done[i, j] == 0) \wedge (A[i] + A[j] == target)$
advance: $G := G + 1; done[i, j] := 1$;

BinarySearch
input: A: array of integers;
G: int initially $G = -1$;
low : int initially $low = 0$;
$high$: int initially $high = n - 1$;
forbidden: $(low \leq high)$
advance: $mid = \lceil low + high \rceil$;
if $(A[mid] = target)$ then $low := mid; high := mid - 1; G = mid$;
else if $(A[mid] < target)$ then $low := mid + 1$; else $high := mid - 1$;

Figure 3.2: LLP Programs for (a) GCD (b) Factorial (c) MaxElement (d) Linear Search (e) Nested Sum (f) PairSum and (g) Binary Search

Show that the predicate $B(Y) \equiv$ "Y is an order ideal of $(X, \leq)$" is lattice-linear in the boolean lattice of all subsets of X.

3. Show that lattice-linearity is not closed under disjunction.

4. Show that lattice-linearity is not closed under negation.

3.8 Bibliographic Remarks

The lattice-linearity property is described in [CG98]. Its application to stable matching is first shown in [Gar17]. Its application to design of parallel algorithms is in [Gar20].

Chapter 4

Basic Graph Algorithms

4.1 Introduction

Graph theory, a cornerstone of algorithmic design, underpins numerous modern applications across diverse domains. In *social network analysis*, graphs model relationships where vertices represent individuals and edges denote interactions, enabling platforms like X to recommend connections or detect communities using algorithms such as breadth-first search or clustering. In *transportation and logistics*, directed graphs optimize routing: companies like Amazon use shortest-path algorithms (e.g., Dijkstra's) on road networks to minimize delivery times, while airlines employ bipartite matching to schedule crews efficiently. *Computer networks* rely on graphs to represent topologies, with spanning tree protocols ensuring loop-free communication and centrality measures identifying critical nodes for cybersecurity. In *bioinformatics*, graphs map protein interactions or gene regulatory networks, where traversal algorithms identify functional pathways, aiding drug discovery. Emerging fields like *machine learning* leverage graph neural networks (GNNs) to process structured data—such as molecular graphs for chemistry or citation networks in academia—outperforming traditional models in tasks like node classification. These applications highlight graphs' versatility, driving innovation in optimization, prediction, and system design.

Graphs come in numerous varieties. They may be *directed* or *undirected*. They may be *simple* or with loops and parallel edges. They may be *weighted* or *unweighted*. They may be *acyclic* or not. For example, when modeling a transportation networks, the nodes may represent important milestones in a city. A directed edge may represent an one-way street and an undirected edge may represent a street that can be traversed in either direction. The weights may represent the distance between the nodes.

In this chapter we cover some basic traversal algorithms in a simple directed graph. Section 4.4 gives an LLP algorithm to traverse a graph. Section 4.3 gives an LLP algorithm to construct a breadth-first-search (BFS) tree from a graph given a source vertex.

4.2 Graph Representations

We present an undirected graph and its directed counterpart, each with vertices v_0, v_1, v_2, v_3, shown with adjacency matrix and adjacency list (as linked lists) representations.

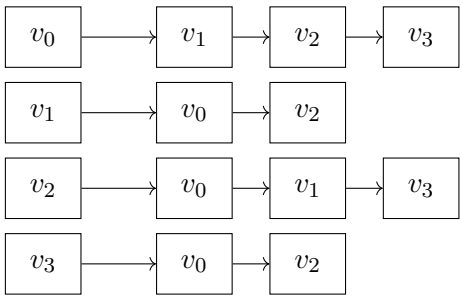


Figure 4.2: Adjacency representation of an undirected graph

Undirected Graph

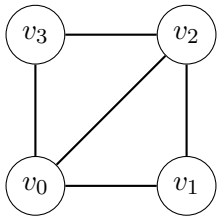


Figure 4.1: An undirected graph

The graph in Fig. 4.1 can be represented using adjacency matrix as follows.

$$\begin{bmatrix} & v_0 & v_1 & v_2 & v_3 \\ v_0 & 0 & 1 & 1 & 1 \\ v_1 & 1 & 0 & 1 & 0 \\ v_2 & 1 & 1 & 0 & 1 \\ v_3 & 1 & 0 & 1 & 0 \end{bmatrix}$$

An entry of 1 indicates an edge; the matrix is symmetric. Such a representation allows answering of any question of the form: is there an edge between v_i and v_j in constant time. However, it takes $O(n^2)$ space irrespective of the number of edges in the graph. When the graph is sparse the adjacency list representation is less wasteful. Fig. 4.2 shows the adjacency list representation of the same graph.

We now consider the directed version of the graph in Fig. 4.3.

As before, we can have the adjacency matrix and adjacency list representation shown below.

$$\begin{bmatrix} & v_0 & v_1 & v_2 & v_3 \\ v_0 & 0 & 1 & 1 & 0 \\ v_1 & 0 & 0 & 1 & 0 \\ v_2 & 0 & 0 & 0 & 1 \\ v_3 & 1 & 0 & 0 & 0 \end{bmatrix}$$

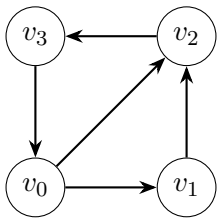


Figure 4.3: A Directed Graph

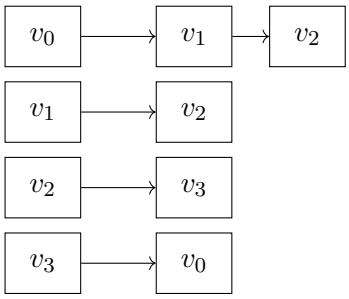


Figure 4.4: Adjacency representation of a directed graph

4.3 Sequential BFS and DFS Traversals in a Graph

In this section, we cover two of the most common ways of traversing a directed graph. The traversal is required for many reasons. We may be interested in searching for a node, finding the number of components in a graph, or checking for some user-specified property.

Breadth-First Search Tree in a Directed Graph

Suppose that we are given one of the nodes v_0 in the graph. Our task is to find all the nodes that are reachable from v_0. Algorithm BFS-Traversal-Sequential can be used for this purpose. It will generate a tree rooted at v_0 such that every node is at the minimum distance from v_0. We maintain a variable *parent* that gives the parent of any node x in the graph. If x is reachable from v_0, then by following the *parent* pointer from x to v_0, we will get a shortest path from v_0 to x. A node v_i has its parent set to v_j if v_j is along a path from v_0 to v_i and has distance one less than that of v_i.

Depth-First Search Tree in a Directed Graph

Another way to traverse a graph is the depth-first search method. While the breadth-first search traverses the graph one layer at a time, the depth-first search continues to take a path as far as it can take and then it backtracks to traverse the remaining graph (in the same fashion!).

Algorithm BFS-Traversal-Sequential: Algorithm BFS to find the Breadth-First-Search Tree in a Directed Graph

```
1   var G: vector of int initially ∀i : G[i] = maxint;
2       parent: vector of int initially ∀i : parent[i] = −1; // null parent
3       S: queue of indices;
4   S.add(s);
5   G[s] := 0;
6   while ¬S.empty() do
7       j := S.removeFirst(); // remove an index from S
8       forall (k ∈ dep(j)):
9           if (G[k] > G[j] + 1)
10              G[k] := G[j] + 1;
11              parent[k] := j;
12              append k to S;
13      endforall;
14  endwhile;
15  return G; // the optimal solution
```

Algorithm DFS-Traversal: Sequential Algorithm DFS to find the Depth-First-Search Tree in a Directed Graph

```
1  var
2      G[0..n − 1]: int, initially 0
3      parent[0..n − 1]: int, initially -1
4      S: stack, initially empty
5      discovered[0..n − 1]: int
6      finished[0..n − 1]: int
7      tick: int, initially 1
8  S.push(0)
9  while ¬S.empty() do
10     j := S.pop()
11     if G[j] = 0 then
12         G[j] := 1
13         discovered[j] := tick
14         tick := tick + 1
15         for k ∈ edges(j) do
16             if G[k] = 0 then
17                 parent[k] := j
18                 S.push(k)
19             end
20         end
21         finished[j] := tick
22         tick := tick + 1
23     end
24 end
```

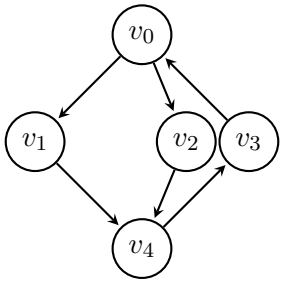


Figure 4.5: An Example of DFS-Traversal

Node	discovered	finished
v_0	1	10
v_1	2	7
v_4	3	6
v_3	4	5
v_2	8	9

Figure 4.6: The values of *discovered* and $finished$ for vertices with DFS on the graph in Fig. 4.5

Algorithm DFS-Traversal visits all the vertices reachable from the vertex s in a depth-first search manner. The variable G is used the mark all the vertices that are reachable from s. The variable S is used as a stack to store the vertices while traversing the graph. The variable *discovered* is used to store the time (tick) when each vertex in the graph is explored. The variable $finished$ is used to store the tick when the algorithm is finished exploring the vertex. The variable *parent* is used to store the depth-first-tree. Anytime a vertex k is visited for the first time from a vertex j, the $parent[k]$ is recorded as j.

The variables *discovered* and $finished$ are used only for recording the *tick* when a vertex is started to be explored and finished for exploration.

Fig. 4.5 shows the DFS traversal on a graph. Suppose we start the DFS traversal from the node v_0. Then, node v_1 is visited next. After v_1, we visit the node v_4. From node v_4, we visit the node v_3. When v_3 is explored, we get to the node v_0 which has already been visited. We backtrack to node v_4 which does not have any other outgoing edge. From v_4, we backtrack to v_1 and then to v_0. Now, from v_0, we can visit the node v_2. From v_2, we can go to node v_4 which has already been visited. Hence, we backtrack to node v_0. At this point, all the outgoing edges from v_0 have been traversed and the DFS traversal is done. The variables for DFS traversal for this graph are shown in Fig. 4.6.

4.4 LLP Algorithms for Traversal in a Graph

Given a graph (V, E) and a source vertex v_0, our goal is to find all the vertices that are reachable from v_0. We can view this problem as searching for a subset of vertices of $W \subseteq V$ such that $v_0 \in W$ and $\forall x, y : (x \in W) \wedge (x, y) \in E$ implies that $y \in W$. We model W using a boolean array G such that $v_i \in W$ iff $G[i]$ is true.

The following LLP algorithm computes the least G (or the smallest W) that satisfies

$$B_{traverse} \equiv G[0] \wedge (\forall (v_i, v_j) \in E : G[j] \geq G[i])$$

The predicate B requires v_0 to be reachable from v_0 and for all edges (v_i, v_j) if v_i is reachable then so is v_j. It can be verified that $B_{traverse}$ is a lattice-linear predicate. It is a conjunctive predicate and the first conjunct is a local predicate. We only need to show that the second conjunct is also lattice-linear. If the second conjunct is false, then there exists $(v_i, v_j) \in E$ such that $G[i]$ equals 1 and $G[j]$ equals 0. In this case, unless $G[j]$ is also set to 1, the predicate can never become true. Therefore, we get the following LLP algorithm.

Algorithm LLP-Traversal: Reachability Set

```
1  P_j: Process for vertex j
2  input: edges(j): list of adjacent vertices
3  init: G[j] := 1 if j = 0, else 0
4  forbidden(j): ∃i : j ∈ edges(i) ∧ G[i] = 1 ∧ G[j] = 0
5      advance(j): G[j] := 1
```

LLP Algorithm is non-deterministic and one could use different strategies to compute forbidden indices. For example, one can maintain a set S that contains *frontier* vertices – set of vertices that are reachable but their outgoing edges have not been explored yet. Algorithm Traversal-Sequential gives a sequential implementation of the LLP algorithm with S as the set of frontier vertices. By using different strategies for removing an index from S, we can traverse the graph in different order. If S is maintained as a queue, the removal corresponds to removing from the head of the queue, and the addition corresponds to appending at the end of the queue, we traverse the graph in a *breadth-first* manner. If S is maintained as a stack such that removal corresponds to *pop*() and the addition corresponds to *push*() method on the stack, then we traverse the graph in a depth-first search manner.

Algorithm Traversal-Sequential: Finding the reachable set of vertices from v_0 in a Directed Graph

```
1  var
2      G: vector of int initially ∀i : G[i] = 0
3      S: set of indices
4  S.add(0) // add 0 as the initial forbidden index
5  G[0] := 1
6  while ¬S.empty() do
7  |   j := S.removeAny(G) // remove any index from S
8  |   forall (k ∈ dep(j)) do
9  |   |   if (G[k] = 0) then
10 |   |   |   G[k] := 1 // advance on k
11 |   |   |   S.add(k) // update frontier
12 |   |   end
13 |   end
14 end
15 return G ;                                   // the optimal solution
```

In the previous section, we saw an algorithm to compute the set of reachable vertices from a given vertex. Now suppose that we are interested in computing the breadth-first-search tree rooted at any given

node v_0. We first define the search lattice L to be the set of distance vectors where $G[i]$ is the distance of vertex i from v_0. Initially, $G[0]$ is zero for the vertex v_0 and ∞ for all other vertices. We define G to be feasible if

$$B_{bfs}(G) \equiv \forall j \neq 0 : \forall (i,j) \in E : G[j] \leq G[i] + 1.$$

Our goal is to maximize G subject to B_{bfs}.

A vertex j is defined to be forbidden if it has a predecessor i such that $G[j] > G[i] + 1$. If none of the vertices is forbidden, we get that $\forall j \neq 0 : \forall (i,j) \in E : G[j] \leq G[i] + 1$. Whenever a vertex j is forbidden, we advance the index j by setting $G[j]$ to $G[i] + 1$. This is equivalent to ensuring that $G[j]$ is at most $G[i] + 1$ for any $i \in pre(j)$.

Algorithm LLP-Traversal-BFS: Finding the reachable set of vertices using BFS.

```
1 Process P_j
2 input: pre(j): list of 1..n;
3 init(j): if (j = 0) G[j] := 0 else G[j] := ∞;
4 ensure(j): G[j] ≤ min{G[i] + 1 | i ∈ pre(j)}
```

The algorithm terminates when there is no forbidden vertex. Any vertex k such that $G[k]$ is infinite is not reachable from the vertex v_0.

We observe that Algorithm LLP-Traversal-BFS does not specify the order in which the algorithm (sequential or parallel) should check if the ensure condition is not met. For example, when the total number of threads is much smaller than the number of processes, then it may be worthwhile giving a priority to the indices that are checked for forbidden predicate. We use the clause **priority** with the LLP algorithm to put an order in which the indices should be checked for the *forbidden* or *ensure* clause. Algorithm LLP-Traversal-BFS-priority shows an algorithm in which the priority of forbidden indices is given by $G[j]$.

Algorithm LLP-Traversal-BFS-priority: Finding the reachable set of vertices using BFS.

```
1 Process P_j
2 input: pre(j): list of 1..n;
3 init(j): if (j = 0) G[j] := 0 else G[j] := ∞;
4 ensure(j): G[j] ≤ min{G[i] + 1 | i ∈ pre(j)}
5 priority: G[j]
```

Algorithm LLP-Traversal-BFS is based on using the predecessor information for each vertex. Alternatively, we can use the adjacency list for each vertex. The set S keeps the set of reachable indices. Initially, only v_0 is the reachable vertex. We explore all the outgoing edges from any vertex $v_j \in S$. The set T stores all the vertices that become reachable when vertices in S are explored. If any vertex k adjacent to v_j gets its $G[k]$ reduced because of the edge (v_j, v_k), it is added to the set T. This exploration is continued until there are no unexplored vertices.

4.5 Layering of a Directed Acyclic Graph

Suppose that we are given a directed graph with no cycles. Such a graph can be "layered" as follows. Every vertex i in the graph is assigned a number $G[i]$ such that if there is a directed edge from i to j, then the

number assigned to i is strictly less than that of j, i.e., for all edges $(i, j) \in E$, $G[i]$ is less than $G[j]$. An application of this concept is the prerequisite structure of courses in a University and the integer assigned to the course corresponds to the earliest semester in which the course can be taken by a student.

In this application, our lattice consists of vectors of natural numbers. We are determining the least vector, G, that satisfies the predicate

$$B_{layer} \equiv \forall (i, j) \in E : G[i] < G[j].$$

The predicate is lattice-linear because if it is false then there exist $(i, j) \in E$ such that $G[i] \geq G[j]$. In this case, the index j is forbidden and unless $G[j]$ is advanced B_{layer} can never become true. For efficiency, we define a predicate

$$fixed(j) \equiv \forall (i, j) \in E : G[i] < G[j].$$

Now, the predicate B_{layer} can be rewritten as

$$B_{layer} \equiv \forall j \in [n] : fixed(j).$$

We will use $fixed[j]$ as a variable in our program. Algorithm shown in Fig. LLP-Layering gives the layering of a directed acyclic graph. We use the boolean array $fixed$ to mark the vertices whose level number in G are final. Initially, all vertices that do not have any predecessors are $fixed$ and labeled with 0. A vertex is forbidden if it is not fixed and all its predecessors are fixed. Whenever a vertex j is forbidden, we mark its level as 1 more than any of its predecessors. In this algorithm, $G[j]$ gives the length of the longest path from any initial vertex to j.

Algorithm LLP-Layering: Layering of a Directed Acyclic Graph.

```
1 input: pre(j): list of 1..n;
2 init(j): G[j] := 0;
3       if pre(j) = {} then fixed[j] := true else fixed[j] := false;
4 forbidden(j): ¬fixed[j] ∧ ∀i ∈ pre(j) : fixed[i]
5       advance(j): G[j] := max_{i∈pre(j)} G[i] + 1; fixed[j] := true;
```

The algorithm takes parallel time equal to the length of the longest path in the directed acyclic graph. Its work complexity is $O(n + m)$.

In many applications, we are interested in coming up with a *total order* on all vertices such that for all edges (i, j), $G[i]$ is strictly less than $G[j]$. This operation is called *topological sort* of a directed acyclic graph. The reader is invited to modify LLP-Layering for such applications.

4.6 Problems

1. Algorithm LLP-Layering gives a level number $G[i]$ for each vertex i such that for any edge (i, j), we have that $G[i]$ is strictly less than $G[j]$. Modify the algorithm to generate a total order on all vertices.

2. Algorithm LLP-Layering gives us the least integral labels satisfying that for all edges (i, j), $G[i]$ is strictly less than $G[j]$. Such a property is possible only for acyclic graphs. Modify the algorithm to output "error" whenever there is a cycle in the input graph.

4.7 Bibliographic Remarks

The reader is referred to [CLRS01] for basic graph algorithms.

Chapter 5

Greedy Algorithms

"Greed is good." — Gordon Gekko, "Wall Street"

5.1 Introduction

Many problems can be solved as making n choices such that at the end of these choices we have the solution of the problem. Greedy algorithms start with with the solution for the problem of trivial size, generally of size 0 or 1. It then keeps increasing the size of the problem until we get back the original problem after n steps. For many applications, this strategy results in a solution that is a global optimal or close to a global optimal. Thus, many algorithms that we have seen earlier can be called greedy algorithms. Prim's Minimum Spanning Tree (MST) is as follows. Given a connected graph, we need to find a spanning tree of that graph which is minimal in the total edge weight. The greedy approach is as follows. Start with an arbitrary node and always add the smallest edge that connects any node in the tree to a node outside the tree. Similarly, Kruskal's Minimum Spanning Tree is as follows. Always pick the smallest edge that does not cause a cycle in the MST being constructed. Similarly, Dijkstra's Shortest Path Algorithm to find the shortest path from the source to all vertices in the given graph is also greedy. It always picks the next vertex with the shortest known distance.

5.2 Interval Scheduling Problem

We start with a standard simple greedy algorithm. Suppose that we have m intervals with the start times s_i and finish times f_i for jobs $i = 1..m$, where $s_i < f_i$. We assume that activity i happens during the open interval $[s_i, f_i)$. Two intervals i and j are *compatible* if $f_i \leq s_j$ or $f_j \leq s_i$. Our goal is to select a maximum-sized subset of mutually compatible intervals. We assume that the intervals are sorted by their finish times. We assume that jobs with the same finish times are sorted according to their start times. Furthermore, no two intervals have identical start and finish times.

Let us begin with a sequential algorithm for this problem shown in Fig. IntervalSequential. Let G be the set of mutually compatible intervals represented as a boolean array such that $G[i]$ is 1 iff the activity i is in the subset of mutually compatible intervals that can be selected. It is clear that the first activity (with the least finishing time) is always in G. (Why?) We now traverse over all intervals as follows. We use the variable *cur* to denote the current activity under consideration and the variable *last* as the last activity that was included in G. We initialize *last* to 0. The current activity is included in G iff its start

time is greater than or equal to the finish time of the last activity. If it is, our current activity becomes the last activity and we explore the next activity.

Algorithm IntervalSequential: Sequential Interval Scheduling

Data: s: array[0..$n-1$] of int // Start times
Data: f: array[0..$n-1$] of int // Finish times
Output: G: array[0..$n-1$] of 0..1 initially 0 // Selected intervals

```
1 G[0] ← 1;                                   // First interval always selected
2 last ← 0;                                   // Index of last selected interval
3 for i ← 1 to n − 1 do
4 |   if s[i] ≥ f[last] then
5 |   |   G[i] ← 1;                           // Interval i is compatible
6 |   |   last ← i
7 |   end
8 end
```

The sequential algorithm takes $O(n)$ time (assuming that the intervals are provided in a sorted manner based on the finish times) and it is optimal.

Theorem 5.1 *For the Activity Selection Problem, selecting activities in increasing order of their finish times yields an optimal solution, i.e., the maximum number of mutually compatible activities.*

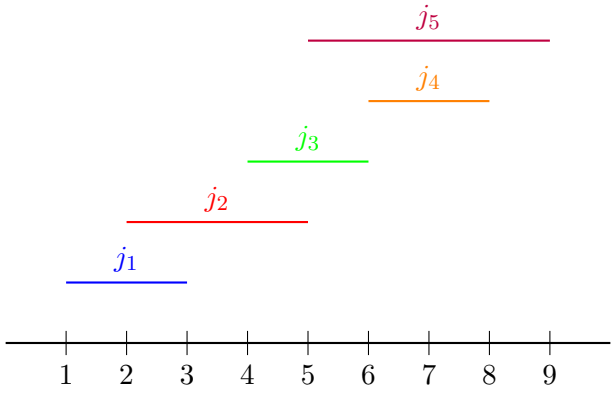
Proof: We prove the theorem using the **exchange argument**. Let G be the set of activities chosen by the greedy algorithm, which selects the activity with the earliest finish time first. Let O be an optimal solution with the maximum number of activities. The greedy algorithm selects the activity with the earliest finish time, say A_1. Suppose O selects a different first activity A_k. Since activities are sorted by finish time, $f_1 \leq f_k$. Replacing A_k with A_1 in O still allows the remaining selections in O to proceed without reducing the total count. By repeatedly applying this argument for the next selected activity, we replace each activity in O with the corresponding activity from G without decreasing the number of selected activities. After at most n exchanges, O is transformed into G. Since O was an optimal solution, and G has the same size, G is also optimal.

■

Suppose, we are given five jobs, each with a start time and an end time. The goal is to select the maximum number of non-overlapping jobs.

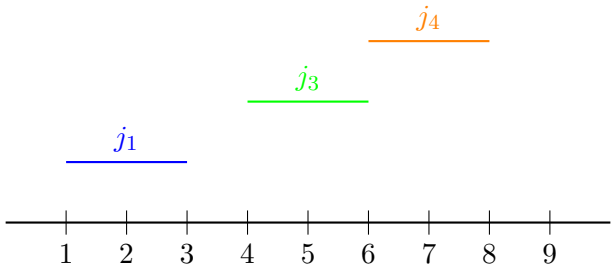
Job	Start Time	End Time
j_1	1	3
j_2	2	5
j_3	4	6
j_4	6	8
j_5	5	9

Table 5.1: List of jobs with their start and end times.



An optimal schedule selects the maximum number of non-overlapping jobs. The optimal solution is:

$$\{j_1, j_3, j_4\}$$



5.3 Interval Partition Problem

Consider a university which is offering multiple courses for students. Every course i has a fixed slot $[s_i, f_i)$ where s_i indicates the start time for the lecture in course i and the f_i indicates the finish time for that lecture. There are n courses in total. The university wants to use as few class rooms as possible for the lecture. Our task is to assign a room to each course such that if two lectures overlap, then they must be assigned different rooms. Since each lecture is being modeled as an interval, this problem is called the *interval partition problem*.

For this problem, we first look at the lower bound on the number of rooms required. Suppose that there are m overlapping lectures. Then it is clear, that any room assignment must use at least m rooms.

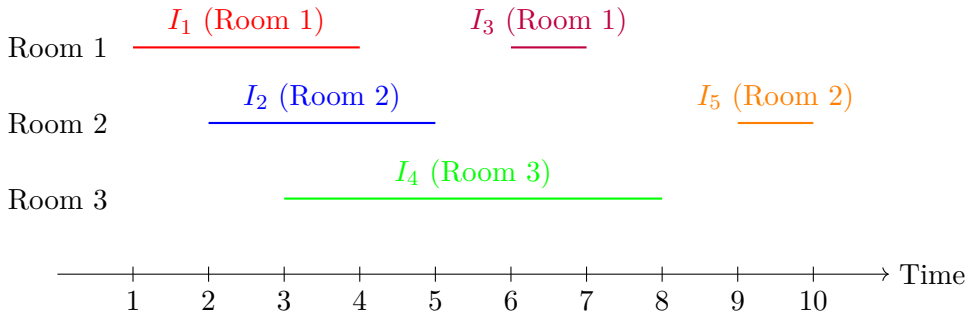
We now show a simple greedy algorithm that achieves the lower bound on the number of rooms. We sort the courses based on the start times of the lectures. We assign courses to rooms one at a time. The first course is assigned room 1. Now suppose that k courses have been assigned their rooms. We assign the course $k+1$ as follows: assign it to the least numbered room that is available at the start time of the lecture. It is clear that a course is assigned a *new* room only if there are overlapping lectures for all small numbered rooms. Thus, our greedy algorithm achieves the minimum number of rooms assigned.

Given the intervals:

$$I_1 = [1, 4], \quad I_2 = [2, 5], \quad I_3 = [6, 7], \quad I_4 = [3, 8], \quad I_5 = [9, 10]$$

We sort them based on the start times.

$$(1,4),(2,5),(3,8),(6,7),(9,10)$$



The minimum number of rooms required is equal to the maximum number of overlapping intervals, which is **3**.

Theorem 5.2 *For the Interval Partitioning Problem, assigning intervals to the first available room in order of their start times produces an optimal solution, i.e., it minimizes the number of rooms required.*

Proof: The greedy algorithm sorts the intervals by increasing start time and assigns each interval to the first available room. If no room is available, it assigns the interval to a new room. Let G be the number of rooms used by this algorithm, and let O be the minimum number of rooms required in an optimal solution. At any time t, let $D(t)$ be the number of overlapping intervals. Since at least $D(t)$ rooms are necessary at that time, the minimum number of rooms required is at least $\max D(t)$. The greedy algorithm assigns a new room only when all current rooms are occupied, ensuring that it never exceeds $\max D(t)$ rooms. Thus, $G \leq O$ and, since O must also use at least $\max D(t)$ rooms, we conclude that $G = O$. Therefore, the greedy algorithm produces an optimal solution.

■

Now, let us determine the time complexity of our algorithm. The sorting of intervals based on the start times can be done in $O(n \log n)$ time. Suppose that we keep the *available* rooms in a heap ordered by the room number. Whenever we need a room for a lecture, we simply remove the minimum available in the heap. This heap operation takes $O(\log n)$ time. Thus, we take $O(n \log n)$ time for the entire algorithm.

5.4 Minimizing Maximum Lateness of Jobs

Suppose that we have n jobs such that each job j takes time t_j to execute and should ideally be completed before its deadline d_j. Our task is to schedule these jobs on a single processor. Suppose that the job j is started at time s_j. Then, the task finishes at time $s_j + t_j$. If $s_j + t_j$ is less than or equal to the deadline d_j, then this job is not late. Otherwise, this job has the *lateness* equal to $s_j + t_j - d_j$. Our task is to schedule these jobs such that the maximum lateness is minimized.

Let us start with a sequential algorithm. Should we consider these jobs in the order of their processing times, their *slack* times $(d_j - t_j)$), or their deadlines d_j? It turns out that for this problem, we should consider jobs in the order of their deadlines.

Algorithm MinMaxLateness: A Sequential Program for the Minimizing Maximum Lateness problem

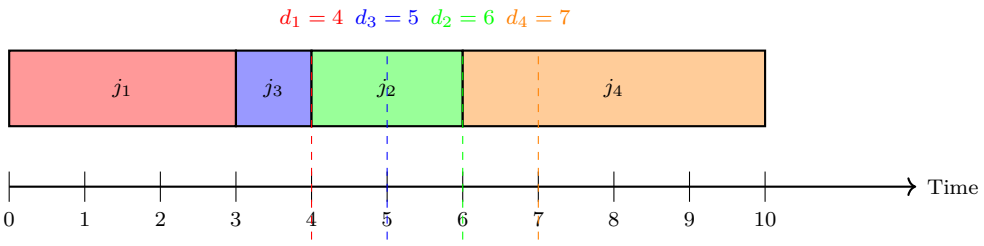
```
1 var G: array[1...n] of int initially 0;
2       const t: array[1...n] of int; // completion times
3       const d: array[1...n] of int ; // deadlines
4 int last := 0;
5 int lateness := 0;
6 for int i := 1 to n do
7       G[i] := last;
8       last := G[i] + t[i];
9       if (last > d[i]) then lateness := max{lateness, (last − d[i])};
```

Algorithm MinMaxLateness is quite simple. It processes jobs in the order of increasing deadlines. The variable $G[i]$ denotes the starting time of job i. The algorithm takes $O(n)$ time when the input is provided as sorted based on the deadlines.

Consider the following example.

Job j_i	**Processing Time** t_i	**Deadline** d_i
j_1	3	4
j_2	2	6
j_3	1	5
j_4	4	7

The schedule Using Earliest Deadline First (EDF) is shown next.



The job j_2 finishes at time 6 but had a deadline of 4. The job j_4 finishes at time 10 but had a deadline of 7. Using earliest deadline first, this schedule minimizes *maximum lateness*, which is 3 units.

Theorem 5.3 *For the problem of scheduling n jobs with known processing times and deadlines on a single machine to minimize the maximum lateness, the Earliest Deadline First (EDF) algorithm produces an optimal solution.*

Proof:

The EDF algorithm schedules jobs in increasing order of their deadlines. Let G be the schedule produced by the EDF algorithm, and let O be an optimal schedule that minimizes maximum lateness. Suppose for contradiction that O is different from G and has a different order of jobs while achieving a smaller maximum lateness.

Consider the first position in which O and G differ. Let job j be scheduled earlier in O than job i, while G places job i before job j. Since EDF sorts jobs by increasing deadlines, we have $d_i \leq d_j$. Since both schedules are contiguous, swapping i and j in O does not delay the completion time of any job before i, nor does it improve the lateness of any job. Instead, it ensures that the job with an earlier deadline finishes sooner, which prevents an increase in lateness. By iteratively applying this argument, we can transform O into G without increasing the maximum lateness. Since O was assumed to be optimal, it follows that G is also optimal. Therefore, EDF produces an optimal schedule.

■

5.5 Huffman Tree

The goal of the Huffman tree problem is to come up with an efficient encoding of a set of symbols. We are given n symbols. For each symbol i, the value $p[i]$ gives the probability that it appears in the given text such that

$$\sum_i p[i] = 1$$

Our goal is to design a *prefix* code such that the given text of symbols can be translated to a string such that the text of symbols can be recovered from the string. We will create a binary tree such that leaves correspond to the symbols and the path from the root to the leaf corresponds to the *code* for the symbol. Note that only the leaves of the tree correspond to the symbol. Also note that leaves may be at different levels and therefore this method returns a variable length code for symbols rather than a fixed length code.

Given n symbols with their probabilities of occurrence, the Huffman coding algorithm constructs an optimal prefix code using the following greedy approach:

1. Insert all symbols into a min-priority queue sorted by probability.

2. While there is more than one node in the queue:

   - Remove the two nodes with the smallest probabilities.
   - Merge them into a new node whose probability is the sum of the two.
   - Insert the new node back into the priority queue.

3. The final remaining node is the root of the Huffman tree.

4. Assign binary codes by traversing the tree: assign ‘0’ to the left branch and ‘1’ to the right branch.

Algorithm 1: Huffman Coding

Input: Set of symbols $S = \{s_1, s_2, \ldots, s_n\}$ and frequencies $f(s_1), f(s_2), \ldots, f(s_n)$
Output: Huffman tree T

$Q \leftarrow$ priority queue (min-heap) of nodes, initialized with S
for *each* $s_i \in S$ **do**
  Create a leaf node n_i with frequency $f(s_i)$ and symbol s_i
  Insert n_i into Q
end
for $i = 1$ ***to*** $n - 1$ **do**
  $x \leftarrow$ extract-min(Q) ; `// Remove node with smallest frequency`
  $y \leftarrow$ extract-min(Q) ; `// Remove next smallest`
  $z \leftarrow$ new internal node with children x and y
  $f(z) \leftarrow f(x) + f(y)$; `// Sum frequencies`
  Insert z into Q
end
$T \leftarrow$ extract-min(Q) ; `// Final tree (root)`
return T

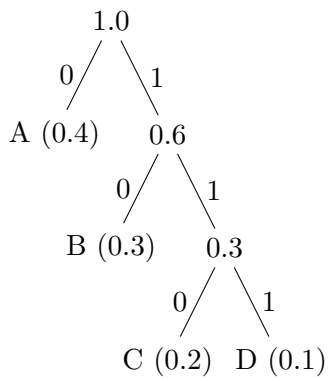
Consider the following symbols and their probabilities:

Symbol	Probability
A	0.4
B	0.3
C	0.2
D	0.1

- Merge (C, D) to create node (CD) with probability 0.3.
- Merge (CD) with B to create node (BCD) with probability 0.6.
- Merge (BCD) with A to create the root node with probability 1.0.

The resulting Huffman codes are shown below.

Symbol	Huffman Code
A	0
B	10
C	110
D	111



The time and space complexity of the Huffman coding algorithm depend on the use of a min-heap for the priority queue Q.

- **Time Complexity:**
  - *Initialization*: Creating n leaf nodes and inserting them into Q takes $O(n \log n)$, as each insertion into a min-heap of size up to n is $O(\log n)$.
  - *Main Loop*: The loop runs $n - 1$ times. Each iteration performs two extract-min operations (each $O(\log n)$) and one insertion ($O(\log n)$), totaling $O(3 \log n)$ per iteration. Thus, the loop takes $O((n-1) \cdot 3 \log n) = O(n \log n)$.
  - *Final Extraction*: Extracting the root is $O(\log n)$, negligible compared to the loop.
  - *Total*: $O(n \log n) + O(n \log n) = O(n \log n)$.
- **Space Complexity:**
  - *Priority Queue*: Q stores at most n nodes initially, reducing to 1, requiring $O(n)$ space.
  - *Tree Nodes*: The Huffman tree has n leaf nodes and $n-1$ internal nodes, totaling $2n-1$ nodes, each storing a frequency and pointers (or symbol for leaves), using $O(n)$ space.
  - *Total*: $O(n) + O(n) = O(n)$.

Thus, the algorithm runs in $O(n \log n)$ time and uses $O(n)$ space, where n is the number of symbols.

5.6 LLP: Interval Scheduling Algorithm

We now explore LLP versions of all the problems discussed in the chapter. We maintain a linked list of all the jobs under considerations. These jobs are in the order of their finish times. The algorithm will keep a boolean $G[j]$ for each of the job j. The variable $G[j]$ is initially false and it is set to true only if it is guaranteed to be in the maximal set of jobs that are chosen by the algorithm. We call a job j *fixed* if $G[j]$ is true. We now make the following observations:

1. If any job j has its start time greater than the finish time of the previous job, then that job is always included in the final set. Thus, job j can be fixed.
2. For any job, if the previous job in the linked list is fixed and the start time of the current job is less than or equal to the finish time of the previous job, then this job can be deleted.

It follows from the above two rules that eventually all jobs will either be fixed or deleted. The algorithm terminates when all jobs are fixed. The set of all fixed jobs is a maximal set of jobs that can be completed.

The LLP algorithm for the activity selection problem is shown in Fig. IntervalLLP.

Algorithm IntervalLLP: LLP Program for the Interval Scheduling problem

```
1  // Assume that all jobs are in a doubly linked list
2  // The variable prev denotes the previous node in the linked list
3  input:
4       s[j] : int, f[j]: int;
5       G[j]: boolean;
6  init: if (j = 1) then G[j] := true; else G[j]:= false;
7  forbidden: ¬G[j] ∧ f[prev] < s[j]
8       advance: G[j] := true;
9  forbidden: (j > 1) ∧ G[prev] ∧ s[j] < f[prev]
10      advance: delete this node from the linked list;
```

For example, consider the following four jobs: $job_1 = [1, 4), job_2 = [2, 5), job_3 = [4, 5), job_4 = [5, 7)$. The first job gets fixed because the first job is always fixed. The fourth job also gets fixed because its starting point is bigger than the finishing time of the previous job. Since the first job is fixed and the second job starts before the first job is finished, it is deleted from the list. Now, the third job is also fixed because it starts after the first job is finished.

The sequential algorithm takes $O(n)$ time.

5.7 LLP: Interval Partition Algorithm

We now explore an LLP version of the algorithm for Interval Partition. Let $G[i]$ denote the room assigned to course i (where courses are sorted based on the start times). We initialize $G[i]$ to 1 for all i. This room assignment is feasible iff there are no overlapping intervals. We now define feasibility of G for any set of intervals. An assignment G is feasible if for any pair of intervals i, j whenever they overlap, they are in different rooms. Formally, let $B(k)$ be defined as

$$\forall i, j : (i \leq k) \wedge (j \leq k) \wedge s_j \leq f_i \Rightarrow room(i) \neq room(j)$$

$B(k)$ denotes that the room assignment is feasible for courses $1 \ldots k$. Then, G is feasible if it satisfies $B(n)$. Observe that $B(0)$ holds trivially.

The predicate $B(n)$ may not be true for the initial assignment when there are overlapping intervals. Also, note that if we had assigned a separate room for each interval, then the predicate is trivially true. Our interest is in finding the least assignment to the rooms that makes the predicate true. We consider the lattice of n dimensional vectors of natural numbers. The bottom element of this lattice is the vector with all ones. We use $fixed(i)$ to denote that the course i has been assigned a room that will not be changed. Let $pre(i)$ denote all courses numbered less than i that start prior to course i or at the same time as course i and finish later than

We say that P_k is forbidden if all courses in $pre(k)$ are fixed and $\neg fixed(k)$. Whenever P_k is forbidden, it can be advanced as follows. Let r be the least room that has not been assigned to any overlapping course prior to k. Then, $room(k)$ is set to r and $fixed(k)$ is set to true.

Algorithm IntervalPartition: An LLP Program for the Interval Partition Problem

```
1  P_j: Code for thread j
2  input:
3       s[j] : int, f[j]: int;
4       const pre[j]: set of 1..n initially pre[j] = {i | (i < j) ∧ (s[i] ≤ s[j]) ∧ (s[j] < f[i])};
5  var:
6       G[j]: int initially 1;
7       fixed[j]: boolean initially false;
8  forbidden: (∀i ∈ pre(j) : fixed(i)) ∧ ¬fixed(j)
9       advance: G[j] := min_{k∈pre(j)}{r | r ≠ G[k]};
10           fixed[j] := true;
```

As before, we assume that the intervals are provided to us sorted by their start times. Each index j can keep track of the number of intervals in $pre[j]$ that have been fixed. Whenever all the prerequisite intervals are fixed, the index j is forbidden. We have to compute the minimum room that is available when all prerequisite intervals are fixed.

5.8 LLP: Minimizing Maximum Lateness of Jobs

We now consider the LLP version of this algorithm. Variable $G[j]$ denotes the completion time for job j.

Algorithm MinLateLLP: LLP Program for the Minimum Lateness problem

```
1  // Assume that all jobs are in a doubly linked list
2  // The variable prev denotes the previous node in the linked list
3  input:
4       t[j]: int; // time required for job j
5       d[j]: int; // deadline for job j
6       G[j]: int init 0;
7  forbidden: ¬fixed[j] ∧ ((j = 1) ∨ (fixed[j − 1])
8       advance: fixed[j] = true; if (j > 1) then G[j] := G[j − 1] + t[j]; else G[j] := t[j]
```

5.9 LLP: Huffman Coding

We now give an LLP algorithm for Huffman coding.

5.10 Summary

The greedy strategy for *Interval Scheduling* prioritizes activities with the earliest finish time to maximize the number of non-overlapping intervals, ensuring that more activities can be accommodated. In *Interval Partitioning*, assigning each interval to the earliest available room ensures minimal room usage by efficiently

Algorithm LLP-HuffmanTree1: Finding Huffman Code

1 **input**: p:array of real;// frequency of each symbol
2 **init**: $G[i,j] = 0 \; \forall i \neq j$;
3 $\quad G[i,i] = p[i]$;
4 **always**: $s(i,j) = \sum_{k=i}^{k=j} p[i]$
5 **ensure**(i,j):
6 $\quad G[i,j] \geq \min_{i \leq k < j} G[i,k-1] + s(i,j) + G[k+1,j]$

Problem	Strategy	Time Complexity
Interval Scheduling	Earliest finish time	$O(n \log n)$
Interval Partitioning	Earliest available room	$O(n \log n)$
Minimizing Lateness	Earliest deadline	$O(n \log n)$
Huffman Tree	Merge smallest probabilities	$O(n \log n)$

Table 5.2: Summary of greedy algorithms.

tracking overlapping intervals. For *Minimizing Maximum Lateness*, scheduling jobs by their earliest deadline prevents unnecessary delays, minimizing the worst-case lateness across all jobs. Finally, in *Huffman Tree Construction*, merging the least probable symbols first ensures that frequent symbols receive shorter codes, leading to an optimal encoding that minimizes the total cost of transmission.

Table 5.2 summarizes all four problems considered in this chapter. In time complexity, we have included time to sort the array as required by the strategy.

5.11 Problems

1. A network processes requests that require a fixed bandwidth for a given time interval. Find the minimum number of servers required to handle all requests without exceeding bandwidth limits.

2. A railway station needs to schedule maintenance for incoming trains before their next departure. Each train has a maintenance duration and a deadline. Find the optimal order of maintenance to minimize the worst delay.

3. Given n intervals and an integer k, find the maximum subset of intervals such that at most k intervals overlap at any time.

4. Each task has a required start time and finish time, but there must be a mandatory idle gap of at least g units between consecutive tasks in the same room. Find the minimum number of rooms required.

5. Each task has a start and finish time, but also requires a *specific type of machine*. Different machine types may be limited in number. Find the minimum number of machines needed.

5.12 Bibliographic Remarks

This chapter on greedy algorithms covers several classic problems—Interval Scheduling, Interval Partitioning, Minimizing Lateness, and Huffman Tree construction: each with well-established foundations in the literature. The Interval Scheduling Problem and Interval Partitioning Problem are covered in [CLRS09], and [KT06]. For the Minimizing Lateness problem, scheduling jobs by earliest deadline (EDF) to minimize maximum lateness is a well-known greedy solution, as shown in Algorithm MinMaxLateness. This algorithm's optimality, proven via an exchange argument, is a standard result in scheduling theory. [LRK76] provides one of the earliest rigorous treatments of this problem, establishing the EDF rule's effectiveness for single-machine scheduling with deadlines. The Huffman Tree construction, which builds an optimal prefix code by greedily merging the least probable symbols, is a cornerstone of data compression. Introduced by [Huf52], this algorithm's greedy nature and optimality for variable-length coding are foundational to information theory. Our sequential description mirrors the classic priority-queue implementation in $O(n \log n)$ time, as detailed in [CLRS09].

Chapter 6

The Shortest Path Problem

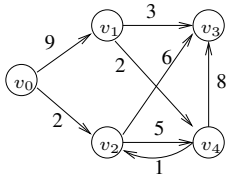
6.1 Introduction

The single source shortest path (SSSP) problem has wide applications in transportation, networking and many other fields. The problem takes as input a weighted directed graph with n vertices and e edges. We are required to find $cost[x]$, the minimum cost of a path from the *source* vertex v_0 to all other vertices x where the cost of a path is defined as the sum of edge weights along that path. Dijkstra's algorithm (or one of its variants) is the most popular single source shortest path algorithm used in practice.

6.2 Dijkstra's Algorithm

We consider a directed weighted graph (V, E, w) where V is the set of vertices, E is the set of directed edges and w is a map from the set of edges to positive reals (see Fig. 22.2 for a running example). To avoid trivialities, we assume that the graph is loop-free and every vertex x, except the source vertex v_0, has at least one incoming edge.

Dijkstra's algorithm maintains $dist[i]$, which is a cost to reach v_i from v_0. Every vertex x in the graph has initially $dist[x]$ equal to ∞. Whenever a vertex is discovered for the first time, its $dist[x]$ becomes less than ∞. We use the predicate $discovered(x) \equiv dist[x] < \infty$. The variable $dist$ decreases for a vertex whenever a shorter path is found due to edge relaxation.



(a)

Figure 6.1: A Weighted Directed Graph

Algorithm Dijkstra: Finding the shortest costs from v_0 .

```
1  var dist: array[0...n-1] of integer initially ∀i : dist[i] = ∞;
2      fixed: array[0...n-1] of boolean initially ∀i : fixed[i] = false;
3      H: binary heap of (j, d) initially empty;// j is the vertex and d is the cost
4  dist[0] := 0;
5  H.insert((0,dist[0]));
6  while ¬H.empty() do
7      (j, d) := H.removeMin();
8      if (fixed[j]) continue;
9      fixed[j] := true;
10     forall k: ¬fixed(k) ∧ (j, k) ∈ E
11         if (dist[k] > dist[j] + w[j, k]) then
12             dist[k] := dist[j] + w[j, k];
13             H.add (k, dist[k]);
14 endwhile;
```

In addition to the variable *dist*, a boolean array *fixed* is maintained. Thus, every discovered vertex is either *fixed* or *non-fixed*. The invariant maintained by the algorithm is that if a vertex x is *fixed* then $dist[x]$ gives the final shortest cost from vertex v_0 to x. If x is *non-fixed*, then $dist[x]$ is the cost of the shortest path to x that goes only through fixed vertices.

A heap H keeps all vertices that have been discovered but are non-fixed along with their distance estimates *dist*. We view the heap as consisting of tuples of the form $(j, dist[j])$ where the heap property is with respect to *dist* values. The algorithm has one main *while* loop that removes the vertex with the minimum distance from the heap with the method H.removeMin(), say v_j, and marks it as fixed. It then *explores* the vertex v_j by relaxing all its adjacent edges going to non-fixed vertices v_k. The value of $dist[k]$ is updated to the minimum of $dist[k]$ and $dist[j] + w[j, k]$. This step is called *edge relaxation*. If v_k is not in the heap, then it is inserted, else if $dist[k]$ has decreased then the label associated with vertex k is inserted in the heap. We abstract this step as the method H.add$(k, dist[k])$. Since the vertex may already be on the heap, the insertion may cause a vertex to be present in a heap with different distances. Hence, when we remove a vertex from the heap, if it is already fixed, we simply go to the next vertex in the heap. The algorithm terminates when the heap is empty. At this point there are no discovered non-fixed vertices and *dist* reflects the cost of the shortest path to all discovered vertices. If a vertex j is not discovered then $dist[j]$ is infinity reflecting that v_j is unreachable from v_0.

Observe that every vertex goes through the following states. Every vertex x is initially *undiscovered* (i.e., $dist[x] = \infty$). If x is reachable from the source vertex, then it is eventually *discovered* (i.e., $dist[x] < \infty$). A discovered vertex is initially *non-fixed*, and is therefore in the heap H. Whenever a vertex is removed from the heap it is a *fixed* vertex. A fixed vertex may either be *unexplored* or *explored*. Initially, a fixed vertex is unexplored. It is considered explored when all its outgoing edges have been relaxed.

The following lemma simply summarizes the well-known properties of Dijkstra's algorithm. We leave them as an exercise.

Lemma 6.1 *The outer loop in Dijkstra's algorithm satisfies the following invariants.*
(a) For all vertices x: $fixed[x] \Rightarrow (dist[x] = cost[x])$.
(b) For all vertices x: $dist[x]$ is equal to cost of the shortest path from v_0 to x such that all vertices in the

path before x are fixed.

For complexity analysis, let n be the number of vertices and m be the number of edges. We analyze the version of Dijkstra's algorithm in which instead of adjusting the key in the heap for a vertex, we simply insert the vertex in the heap. As a result the heap may have a vertex multiple times with different keys. When a vertex is removed, we check if it has already been fixed. If it is fixed, then we do not need to explore it and we can remove the next vertex in the heap. Since a vertex can be inserted in the heap only when an edge is relaxed, we get that there are at most m insertions from the heap. We can also conclude that there are at most m deletions from the heap. Since an insertion or a deletion from a heap takes $O(\log m) = O(\log n)$ time, we get the overall time complexity of $O(m \log n)$.

Here is a walkthrough of Dijkstra's algorithm on the graph, starting from vertex v_0.

1. Set v_0 as the source vertex and assign it a distance of 0. Assign all other vertices a tentative distance of infinity. We insert the source vertex v_0 in the heap H.

2. Since the heap is not empty, we remove the vertex at the top of the heap. It is v_0 with a distance of 0. We mark that vertex as *fixed*. We examine its neighbors, v_1 and v_2. For $v_0 \to v_1$, the existing distance to v_1 is infinity. The distance 0 (distance to v_0) + 9 (edge weight from v_0 to v_1) is less than infinity, so we update the distance to v_1 to 9 and add it to the heap. For $v_0 \to v_2$, the existing distance to v_2 is infinity. The distance $0 + 2$ is less than infinity, so update the distance to v_2 to 2 and add v_2 to the heap.

3. We remove the vertex at the top of the heap. The smallest distance among the unvisited vertices is 2 (for v_2), so set v_2 as the current vertex. We examine its neighbors, v_3 and v_4. For $v_2 \to v_3$, the existing distance to v_3 is infinity. Since 2 (distance to v_2) + 6 (edge weight from v_2 to v_3) is less than infinity, we update the distance to v_3 to 8. For $v_2 \to v_4$, the existing distance to v_4 is infinity. Since $2 + 5$ is less than infinity, we update the distance to v_4 to 7.

4. Continuing in this manner until the heap becomes empty, we get the final distances as: v_0: 0, v_1: 9, v_2: 2, v_3: 8, v_4: 7.

6.3 Bellman-Ford's Algorithm

In this section we give an algorithm that computes the shortest paths from any given vertex even when edge weights are negative. Why could we not apply Dijkstra's algorithm for this problem? Consider the case where the node that can be reached from the source node on one edge with a minimum cost w has a cost of x. Let this node be v. Dijkstra's algorithm assumes that this is the shortest path to node v, because any other path would already have incurred the cost of w by going to some other vertex. However, in the presence of negative weights, there may be a longer path to v by incurring more cost initially but then traversing an edge with negative cost.

When there are negative cost edges, we have to be careful that there are no negative cost cycles. In presence of negative cost cycles, one may decrease the cost of going from s to some vertex v in an unbounded manner by going through the cycle. For now, we will assume that the graph may have negative cost edges but no negative cost cycles.

In 1950's, Bellman and (independently) Ford gave an algorithm for graph with negative weights. The algorithm uses dynamic programming. To set up a recurrence relation, we need to come up with an

appropriate variable with an index k such that its base case when k is small is easy and when k is large, we reach our goal. We let $d^k[v]$ be the cost of reaching v from the source vertex s with a path of at most k edges. It is clear that the base case of k equal to 0 is quite simple. The value $d^0[v]$ equals 0 when v equals s and ∞ otherwise. When k equals $n-1$, we claim that $d^k[v]$ is exactly equal to the shortest path from s to v. Why could there be not a path with n edges or more with a lower cost? If there is a path with n edges, then at least some vertex w appears more than once. The path from w to itself forms a cycle. From our assumption, there are no negative cost cycles. If the cycle is not negative, then we get a smaller path with equal or lower cost by removing the cycle.

The recurrence relation is as follows: $d^0[v] = 0$ is v equals s and ∞, otherwise.

$$d^k[v] = \min\{d^{k-1}[v], \min_{(u,v)\in E} d^{k-1}[v] + w(u,v)\}$$

We let $p[v]$ denote the predecessor of node v in the shortest path. Initially, $p[v]$ is null for all vertices.

Fig. Bellman-Ford Algorithm shows the procedure used when the edge weights in a directed graph may be negative.

Algorithm Bellman-Ford Algorithm: Finding the shortest costs from v_0 when edge weights may be negative .

Input: Graph $G = (V, E)$, source vertex s, edge weights $w(u, v)$
Output: Distance array $d[]$, Predecessor array $p[]$

```
1  foreach vertex v ∈ V do
2  |  d[v] ← ∞;
3  |  p[v] ← null;
4  end
5  d[s] ← 0;
6  for k = 1 to |V| − 1 do
7  |  foreach edge (u, v) ∈ E do
8  |  |  if d[u] + w(u, v) < d[v] then
9  |  |  |  d[v] ← d[u] + w(u, v);
10 |  |  |  p[v] ← u;
11 |  |  end
12 |  end
13 end
14 foreach edge (u, v) ∈ E do
15 |  if d[u] + w(u, v) < d[v] then
16 |  |  return "Negative cycle detected";
17 |  end
18 end
19 return d[], p[];
```

The first *for* loop initializes the value of d and p for the base case corresponding to k equal to zero. We now compute d^k for k equal to $1 \ldots n-1$. Instead of computing the recurrence explicitly for each vertex, we *relax* each edge to check if $d^k[v]$ can be reduced via some edge (u, v). Finally, we check that if there is a negative cost cycle in the graph. If there is a negative cost cycle reachable from the source vertex, then there would be an edge (u, v) in the graph such that $d[v]$ can be reduced further by using that edge even after k has reached the value $n-1$.

The time complexity of Bellman-Ford is $O(|V| \cdot |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges, making it less efficient than Dijkstra's algorithm for graphs with non-negative weights but more versatile due to its handling of negative weights.

In our algorithm, we have iterated over k from 1 to $n-1$. What if the values of d does not change in some iteration of k? The reader is invited to change the algorithm that checks for this condition and terminates it sooner.

6.4 All Pairs Shortest Path Algorithm

Suppose that $A[i,j]$ represents the weight of the direct edge from i to j denoting the cost of going from i to j using at most one edge. We assume that all diagonal entries are zero and that there is no negative weight cycle. Our goal is to find a matrix $G[i,j]$ such that $G[i,j]$ is the least cost of going from i to j by using any number of edges. We apply the idea of dynamic programming again. However, now we define $d^k(i,j)$ as the cost of going from vertex i to vertex j by using intermedaite vertices from $1..k$. When k is zero, it is clear that we cannot use any intermediate vertex. Thus,

$$d^0[i,j] = 0 \; if \; (i=j), w(i,j) \; otherwise.$$

Observe that if there is no edge from i to j when $i \neq j$, $w(i,j)$ is ∞. Now, consider the case when $k \geq 1$. Then, we get the following recurrence:

$$d^k[i,j] = \min\{d^{k-1}[i,j], d^{k-1}[i,k] + d^{k-1}[k,j]\}$$

Thus, we get the algorithm Floyd-Warshall.

Algorithm Floyd-Warshall: All pairs shortest path Algorithm

Input : A weighted graph represented by an $n \times n$ cost matrix $G[i,j]$, where $G[i,j]$ is the cost from vertex i to j (possibly ∞ if no edge exists).

Output: Updated matrix $G[i,j]$ containing shortest path costs between all pairs of vertices.

```
1 for k = 1 to n do
2 |  for i = 1 to n do
3 |  |  for j = 1 to n do
4 |  |  |  if G[i,k] + G[k,j] < G[i,j] then
5 |  |  |  |  G[i,j] ← G[i,k] + G[k,j]
6 |  |  |  end
7 |  |  end
8 |  end
9 end
```

The reader is invited to make two changes to the algorithm. First, modify the algorithm to return an error message when there is a negative cost cycle in the graph. Second, add variable $p[u][v]$ to the algorithm to return the predecessor of v in the path from u to v.

It is clear that the time complexity of Floyd-Warshall's algorithm is $O(n^3)$.

6.5 Bibliographic Remarks

The single source shortest path problem has a rich history. For the history of Dijkstra's algorithm, the reader is referred to the book by [Eri19]. One popular research direction is to improve the worst case complexity of Dijkstra's algorithm by using different data structures. For example, by using Fibonacci heaps for the min-priority queue, Fredman and Tarjan [FT87] gave an algorithm that takes $O(e + n \log n)$. There are many algorithms that run faster when weights are small integers bounded by some constant γ. For example, Ahuja et al [AMOT90] gave an algorithm that uses Van Emde Boas tree as the priority queue to give an algorithm that takes $O(e \log \log \gamma)$ time. Thorup [Tho00] gave an implementation that takes $O(n + e \log \log n)$ under special constraints on the weights. Raman [Ram97] gave an algorithm with $O(e + n\sqrt{\log n \log \log n})$ time. The LLP algorithm for the shortest path is taken from [AKG20]. Bellman and Ford's algorithm is from [Bel58] and [For56].

Chapter 7

The Minimum Spanning Tree Problem

7.1 Introduction

Suppose that we have an undirected weighted graph on n vertices with m edges. Our goal is to find the minimum spanning tree (MST). We assume that all edge weights are distinct. It is known that if the graph is connected and all edge weights are distinct then there is a unique minimum spanning tree. If the graph is not connected, then there is a unique minimum spanning forest. For simplicity of exposition, we will assume that the underlying graph is connected.

Section 7.2 describes the notion of a *fragment* of a minimum spanning tree. This notion is useful in understanding all the spanning tree algorithms which extend a fragment (or, fragments) appropriately.ru

The problem of finding the minimum spanning tree is a canonical problem for which a greedy approach works. In this chapter, we first present a greedy sequential algorithm called Kruskal's algorithm in Section 7.3. Kruskal's algorithm chooses the least cost edge that is feasible as its next edge. It is sequential in nature because it picks one edge at a time. We then show a parallel version of Kruskal's algorithm in which every processor is responsible for either choosing or rejecting one edge.

We then present another greedy sequential algorithm: Prim's algorithm in Section 7.4. Prim's algorithm is greedy in a different way. It chooses the next vertex that can be added to the existing tree with the least cost.

In Section 7.5, we discuss Boruvka's algorithm which chooses multiple edges at a time. Boruvka's algorithm maintains multiple *fragments* and adds edges to all of them in every iteration.

7.2 Fragments

The notion of a *fragment* is crucial in understanding MST algorithms. A fragment is simply a subtree of the MST. Consider the graph in Fig. 7.1. The minimum spanning tree in this graph corresponds to the edges $\{2, 3, 4, 7\}$. The subtree formed by edges 3 and 4 is a fragment with three vertices $\{a, b, c\}$ and two edges $\{(a, c), (b, c)\}$.

A crucial property of the MST is as follows.

Lemma 7.1 *Let F be a fragment. Let e be the edge with minimum weight that is outgoing from F. Then, $F \cup \{e\}$ is also a fragment.*

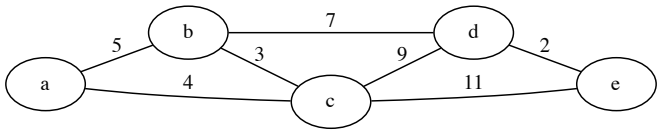


Figure 7.1: An undirected weighted graph

Proof: In the minimum spanning tree T, there must be at least one edge going out of the fragment F. Let that edge be f. If we add e to T and remove f, we get another tree T' with lower weight than T, a contradiction because we assumed that T is the minimum spanning tree.

■

In the fragment formed by edges with weights 3 and 4, there are three outgoing edges — edges with weight 7, 9 and 11. The edge 5 is not outgoing since it connect vertices that are part of the fragment. According to Lemma 7.1, the edge with weight 7 can be added to the edges with weight 3 and 4 to grow the fragment.

7.3 Kruskal's Algorithm

Kruskal's algorithm is a canonical greedy algorithm for the minimum spanning tree problem. It chooses edges one at a time in a greedy fashion. Let T be the set of edges chosen by the algorithm at any stage. The algorithm maintains the invariant that T does not have any cycle. Initially T is empty and therefore trivially satisfies the invariant. The algorithm considers the edges in the increasing order of weights. For this reason it is sometimes also known as the *shortest-edge-next* algorithm. Suppose that the algorithm has chosen edges in T so far. To grow T, it finds the least weight edge that does not form a cycle with existing edges in T. If any edge e forms a cycle with T, then it is rejected and the algorithm considers the least weight edge of the remaining edges.

In Algorithm Kruskal, the variable T keeps the set of all edges that are chosen and the variable *Rejected* keeps the set of all edges that are rejected. Consider the graph shown in Fig. 7.1. Kruskal's algorithm will first choose the edge with weight 2 since it has the least weight. It then chooses the edges with weight 3 and 4 because these edges do not form any cycle. The next edge has weight 5. However, this edge needs to be rejected because it forms a cycle with the already chosen edges with weights 3 and 4. The algorithm then chooses the edge with weight 7 and terminates.

The algorithm is dominated by the cost of sorting the edges: $O(m \log n)$. Checking whether the edge e forms a cycle with T can be done efficiently using find-union data structure.

The Find-Union data structure maintains a collection of disjoint sets, supporting two primary operations:

- **Find**: Determines which set a particular element belongs to by returning its representative (or "root"). With path compression, this operation is nearly constant time, $\mathcal{O}(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function.

Algorithm Kruskal: Computing Minimum Spanning Tree

```
1  Input: Undirected Weighted Graph: (V, E, w).
2  Output: Minimum Weight Spanning Tree
3  var
4       T, Rejected: set of edges initially {};
5  while (|T| < n − 1) do
6       if T ∪ Rejected = E then return null; // no spanning tree in the graph
7       e := the least weight edge that is not in T ∪ Rejected
8       if e forms a cycle in T then
9            Rejected := Rejected ∪ {e}
10      else T := T ∪ {e}
11 endwhile
12 return T
```

- **Union**: Merges two sets into one by linking their roots, typically using rank or size heuristics to keep trees balanced, also achieving $\mathcal{O}(\alpha(n))$ amortized time.

Initially, each element is in its own set. As Kruskal's algorithm processes edges, Union merges sets of vertices connected by selected edges, while Find checks if adding an edge would form a cycle (i.e., if its vertices are already in the same set).

Here's the basic implementation of find-union data structure.

Algorithm Find: Find the root of x's set with path compression

```
1  Input: Vertex x
2  Output: Root of the set containing x
3  var
4     parent: array of int
5  if parent[x] ≠ x then  parent[x] := Find(parent[x]) // path compression
6  ;
7  return parent[x]
```

Algorithm Union: Union by rank of sets containing x and y

```
1  Input: Vertices x, y
2  Output: Unified set containing x and y
3  var
4      parent: array of int
5      rank: array of int
6  rootX := Find(x)
7  rootY := Find(y)
8  if rootX = rootY then return;
   // already in same set, no union needed
9  if rank[rootX] < rank[rootY] then
10 |   parent[rootX] := rootY
11 end
12 else if rank[rootX] > rank[rootY] then
13 |   parent[rootY] := rootX
14 end
15 else
16 |   parent[rootY] := rootX
17 |   rank[rootX] := rank[rootX] + 1 // increment rank if equal
18 end
```

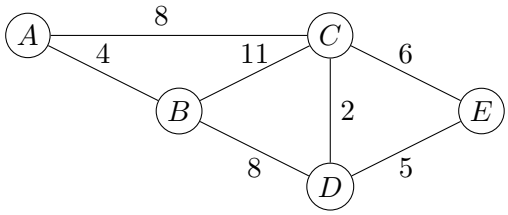
Kruskal's algorithm sorts all edges by weight, then iteratively adds the smallest edge to the MST if it doesn't form a cycle. The Find-Union structure ensures cycle detection:

1. Sort edges in non-decreasing order of weight.
2. For each edge (u, v), if Find$(u) \neq$ Find(v), include the edge and perform Union(u, v).
3. Repeat until the MST has $n - 1$ edges, where n is the number of vertices.

Consider a graph with 5 vertices (A, B, C, D, E) and the following edges with weights:

- $A - B : 4$, $A - C : 8$, $B - C : 11$, $B - D : 8$, $C - D : 2$, $C - E : 6$, $D - E : 10$

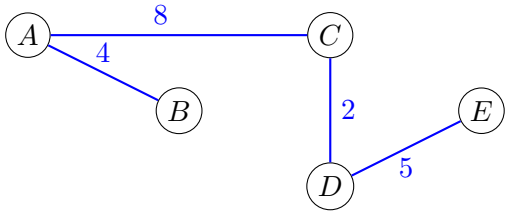
We apply Kruskal's algorithm, using Find-Union to build the MST for the following graph.



1. **Sort Edges**: $C - D(2)$, $A - B(4)$, $D - E(5)$, $C - E(6)$, $A - C(8)$, $B - D(8)$, $B - C(11)$.
2. **Initialize**: Each vertex in its own set: $\{A\}, \{B\}, \{C\}, \{D\}, \{E\}$.
3. **Edge** $C - D(2)$: Find$(C) = C$, Find$(D) = D$. Union: $\{A\}, \{B\}, \{C, D\}, \{E\}$.
4. **Edge** $A - B(4)$: Find$(A) = A$, Find$(B) = B$. Union: $\{A, B\}, \{C, D\}, \{E\}$.

5. **Edge** $D - E(5)$: Find$(D) = C$, Find$(E) = E$. Union: $\{A, B\}, \{C, D, E\}$.
6. **Edge** $C - E(6)$: Find$(C) = C$, Find$(E) = C$. Same set, skip.
7. **Edge** $A - C(8)$: Find$(A) = A$, Find$(C) = C$. Union: $\{A, B, C, D, E\}$. MST complete.
8. **Reject Others**: $B - D(8)$, $B - C(11)$ form cycles.

The MST includes edges $C - D(2)$, $A - B(4)$, $D - E(5)$, $A - C(8)$, with total weight 19.



The Find-Union data structure enables Kruskal's algorithm to efficiently construct an MST by managing vertex sets and avoiding cycles. With optimizations like path compression and union by rank, it achieves near-linear time complexity

7.4 Prim's Algorithm

Prim's algorithm is also a greedy algorithm. It builds the minimum spanning tree by increasing the size of a single *fragment* by adding the minimum weight outgoing edge of the fragment. It simply exploits the Lemma 7.1 to increase the size of fragment until it becomes the MST. At any stage, Prim's algorithm has a fragment F. It finds the minimum outgoing edge from that fragment e. This edge can be viewed as the edge from the fragment to its nearest neighbor. Therefore, this algorithm is sometimes also known as the *nearest-neighbor-next* algorithm. To find the nearest neighbor, every vertex v maintains a label d which corresponds to the cost of adding v to the fragment. At every iteration, the algorithm chooses the vertex v with the minimum d value and adds it to the fragment. The array $fixed$ keeps track of the vertices in the fragment. Whenever, a new vertex v is *fixed* and added to the fragment, the d values for any adjacent vertex v' is updated as follows. We check whether the weight of the edge (v, v') is lower than the previous value of $d[v']$. If this is true, then $d[v']$ is updated to $w[v, v']$. We also use *parent* pointer with each node which keeps track of the node v that is responsible for the d value of v'.

Consider the graph shown in Fig. 7.1 again. For Prim's algorithm, we start from a fixed node. Suppose we start from the vertex a. Then, the nearest neighbor is c with the cost of 4. The next nearest neighbor to the fragment with vertices $\{a, c\}$ is the vertex b. The cost of adding b is 3. At this point, we have vertices $\{a, b, c\}$ in the fragment. The cost to add vertex d is 7 and to add the vertex e is 11. We add the vertex d to our fragment with the cost 7. Finally, e is added with the cost 2. Hence, the edges are added to the tree in the order $4, 3, 7, 2$. Note that the set of edges chosen are identical to Kruskal's algorithm even though the order in which they are chosen is different. This is not surprising because there is a unique minimum spanning tree when all edge weights are unique (Problem 1).

The step of finding the vertex v with minimum d value can be done either by simply traversing the array d or by maintaining d in a heap. If we simply traverse the array d to find the minimum, the work complexity of the above algorithm is $O(n^2 + e)$. In every iteration of the while loop, we perform $O(n)$ work for finding the minimum and there are $O(n)$ iterations of the loop. The work for processing edges over all iterations is $O(e)$ because every edge is processed at most once. If we use a heap to store d values

Algorithm Prim: Finding a Minimum Spanning Tree (MST)

```
1  Input: Undirected Weighted Graph: (V, E, w).
2  Output: Minimum Weight Spanning Tree
3  var
4       d: array[0..n − 1] of real initially ∞; //d[v] is the cost to add vertex v
5       parent: array[0..n − 1] of int initially −1; // parent[v] is the node that corresponds to d[v] cost
6       fixed : array[0..n − 1] of boolean initially false; //vertices whose d value is fixed
7       T: set of edges initially {};
8  d[0] := 0;
9  while (|T| < n − 1) do
10      v := arg min_i{d[i] | ¬fixed[i]} ;
11      if d[v] = ∞ then return null;// no spanning tree in the graph
12      if parent[v] ≠ −1 then add (v, parent[v]) to T
13      fixed[v] := true;
14      forall (v, v′) ∈ E:
15          if w[v, v′] < d[v′] then
16              d[v′] := w[v, v′];
17              parent[v′] := v;
18      end // forall
19 end  //while
```

of all the vertices that are not fixed, we require $O(m)$ insertions on the heap resulting in $O(m \log n)$ work complexity. This approach results in better work complexity when the graph is sparse and m equals $O(n)$.

7.5 Boruvka's Algorithm: Sequential Implementation

In Prim's algorithm, we started with a trivial fragment including just the vertex v_0. We kept increasing the size of the fragment till it became a spanning tree. In Boruvka's algorithm, we may have more than one fragment. We increase the size of all fragments by adding the minimum outgoing edge for each fragment.

Algorithm BoruvkaSeq presents the sequential Boruvka algorithm for finding the MST. We use T to denote the set of tree edges. Initially, T is empty. When we determine the components in (V, T), we get that there are n components as each vertex is a component by itself when T is empty. The algorithm finds the minimum weight outgoing edge for each component as follows. At any iteration, we use BFS to find the least numbered vertex that any vertex is connected to in the graph (V, T). This vertex serves as the identifier for the component of the node i, and we use the variable $cid[i]$ to store it. Once we have determined the component identity of all nodes, we move to the next step of determining the minimum weight outgoing edge for each component. We traverse all edges and for each edge that connects two different components we check whether it is cheaper than previously known outgoing edge for the component on either side. Once we have determined all minimum weight edges for every component, we add these to T and start the next iteration.

For example, consider the graph in Fig. 7.1. Initially, T is empty and there are 5 components. We we compute mwe for each component, we get the edges $4, 3, 3, 2, 2$ as the minimum weight edges of a, b, c, d, e, respectively. Once, these edges are added we have two components: $\{a, b, c\}$ and $\{d, e\}$. We then find

Algorithm BoruvkaSeq: Finding MST

```
1  Input: Undirected connected Weighted Graph: (V, E, w).
2  Output: Minimum Weight Spanning Tree
3  var
4      T: { set of edges } initially {};
5      cid: array[1..n] of 0..n initially all 0;
6      mwe: array[1..n] of edge initially all null;
7      dist: array[1..n] of 0..n initially all ∞;
8  while (|T| < n − 1) do
9      visited: array[1..n] of boolean initially all false;
10     for i := 1 to n do
11         if (¬visited[i])
12             // do a BFS in the graph (V, T) from vertex i setting cid of every visited vertex to i
13             BFS(i);
14     for (i, j) ∈ E such that (cid[i] ≠ cid[j]) do
15             if w[i, j] < dist[cid[i]]
16                 dist[cid[i]] = w[i, j]
17                 mwe[cid[i]] = (i, j)
18             if w[i, j] < dist[cid[j]]
19                 dist[cid[j]] = w[i, j]
20                 mwe[cid[j]] = (i, j)
21     forall i do:
22         T := T ∪ mwe[cid[i]];
23 endwhile
24 return T
```

mwe of these two components as the edge 7. On adding this edge, we have chosen $(n-1)$ edges and the algorithm terminates with the edges $\{2, 3, 4, 7\}$.

After every iteration, the number of connected components in (V, T) reduces by at least a factor of two. This is because every component gets attached to some other component. The worst case is when every least weight edge chosen by any component is also chosen as the least weight edge by the component on the other side of the edge. Hence, the algorithm takes at most $O(\log n)$ iterations of the while loop. It is easy to see that the least weight edge outgoing from each component is found in $O(m)$ work. Thus, the algorithm takes $O(m \log n)$ work.

7.6 Problems

1. Show that there is a unique minimum spanning tree in any weighted undirected graph when all edge weights are distinct.

7.7 Bibliographic Remarks

Sequential algorithms for minimum spanning tree can be found in [CLRS01]. Parallel Boruvka algorithm is available in [SMDD19].

Chapter 8

Sorting Algorithms

8.1 Introduction

In this chapter we consider some basic ideas in designing algorithms which are crucially based on the notion of *order* on a set of elements of size n. We will assume that the order is total, i.e., for every two elements x and y, either x is less than or equal to y, or y is less than or equal to x in that order.

Before we discuss sorting algorithms, let us consider the problem of searching an element in a sorted array. Suppose that we are given a sorted array A of size n and an element x. We are interested in finding if there exists an i such that $A[i]$ equals x. On a single processor, we can accomplish this using binary search in $O(\log n)$ time. The idea is to compare x with the middle element of the array. If the middle element equals x, we are done; otherwise, the range of indices of A which can possibly have x is divided by a factor of 2.

This chapter is organized as follows. Section 8.2 discusses sorting algorithms based on swapping consecutive entries that are out of order. This is a general class of algorithms that are very natural. However, these algorithms take $O(n^2)$ sequential time in the worst case. Section 8.3 discusses *MergeSort*. Section 8.4 discusses *QuickSort*. Since MergeSort and QuickSort are based on divide-and-conquer paradigm, they are revisited in Chapter 9. All these sorting algorithms are based on the comparison of two entries in the array. Finally, Section 8.5 describes a sorting algorithm that is not based on the comparison of two entries.

8.2 Sorting Algorithms based on Swapping Consecutive Entries

Let A be an array of distinct integers. Our goal is to sort the array. At face value, this problem does not appear to be searching for a satisfying element in a lattice. However, with a little effort, it can be viewed from that angle. Observe that the problem of sorting the array is same as finding an permutation π such that on applying that permutation to the array, we get a sorted array. For example, if the input array is $[45, 12, 15]$, then we know that the permutation $[3, 1, 2]$ will sort the array once the entry i in array A goes to $\pi(i)$. There are many different ways to represent a permutation. Let us define the *inversion* number of any entry i as the number of entries less than i that appear to the right of i in the permutation. Thus, the permutation $[3, 1, 2]$ can be equivalently written using the concept of inversions as $[2, 0, 0]$ because there are two numbers less than 3 that appear to the right of 3, zero numbers less than 1 that appear to the

right of 2 and zero numbers less than 2 that appear to the right of 2. The reader can verify that there is a 1-1 correspondence between the set of inversions and the set of permutations.

We now consider the set of all inversions. Note that the last entry on an inversion table is always zero because there cannot be any inversion after the last entry in the array. The first entry can have $0 \ldots n-1$ inversions, the second entry can have $0 \ldots n-2$ inversions, and so on. Thus, the total number of inversion vectors possible are $n!$. We define an order between two inversion vectors base on component-wise order. The set of all inversions forms a finite distributive lattice under this order. The bottom element is the zero vector which corresponds to the identity permutation. When the zero inversion vector is applied to any array, we get back the same array. Our goal is to find the inversion vector which on its application gives us the sorted array. G is initialized to the zero inversion vector. Let us formalize the definition of a feasible inversion vector. Suppose that we are at an inversion vector G such that it is less than the unique inversion vector that sorts A. When G is applied to A, the array is not sorted. This means that there exists an index i such that still has nonzero inversions, i.e, there exists $j > i$ such that $A[j] < A[i]$. It is clear that any inversion vector in which the number of inversions for i is not increased cannot result in the sorted array. To check for forbidden indices, instead of checking for all inversions, we will only check for *immediate* inversion in our first algorithm. If $A[i] > A[i+1]$ on applying G, then clearly i is forbidden in G. Swapping $A[i]$ and $A[i+1]$ will advance G vector by incrementing $G[i]$.

Instead of first finding the inversion vector and then applying it, we will continue to apply inversions as we traverse the lattice searching for the optimal inversion vector. In fact, we will not even maintain the inversion vector; we will only keep the effect of applying the inversion vector to the original array.

Algorithm LLP-Sort1: High-Level LLP Sorting via Adjacent Swaps

```
1 P_j: Code for thread j
2 var
3     A[1..n]: array of int ;                          // Input array
4     G[1..n]: array of 0..n − 1 initially 0 ;         // Inversion vector (abstract)
5 forbidden(j): A[j] > A[j + 1] ;                      // G[j] misses an inversion
6     advance(j): swap(A[j], A[j + 1]) ;               // Increment G[j]
```

Algorithm LLP-Sort1 is a non-deterministic algorithm since multiple j may be forbidden at any point. One can create many instances of deterministic algorithms with different order of evaluating forbidden indices.

Algorithm Bubble-Sort checks forbidden indices from 1 to $n-1$ in round robin order until no forbidden index is found. We leave it as an exercise to show that the repeat loop is executed at most n times giving us the sequential time complexity of $O(n^2)$.

Another schedule is to increase the size of the sorted array one at a time. Suppose that $A[1..i-1]$ is sorted. This means that the inversion vector is zero vector for $A[1..i-1]$. We now consider the array $A[1..i]$. Since there is at most one entry added at the end, the inversion number for any entry can increase by at most one. We simply have to check for inversion with respect to the new entry.

The time complexity is clearly $O(n^2)$ due to nested *for* loops. We leave it for the reader to show how the second for loop can be cut short.

We now prove that any comparison-based sorting algorithm that restricts itself to comparing consecutive entries (adjacent elements) in an array of size n requires at least $\Omega(n^2)$ comparisons in the worst case. This applies to algorithms like Bubble Sort or Insertion Sort, where comparisons are limited to pairs of elements

Algorithm Bubble-Sort: Bubble Sort

```
1  var
2      A[1..n]: array of int ;                                  // Input array
3      found: boolean
4  repeat
5      found ← false
6      for j = 1 to n − 1 do
7          if A[j] > A[j + 1] then
8              found ← true
9              swap(A[j], A[j + 1])
10         end
11     end
12 until ¬found
```

Algorithm Insertion-Sort: Insertion Sort

```
1  var
2      A[1..n]: array of int ;                                  // Input array
3  for i = 1 to n do
4      for j = i − 1 downto 1 do
5          if A[j] > A[j + 1] then
6              swap(A[j], A[j + 1])
7          end
8      end
9  end
```

at positions i and $i+1$.

Theorem 8.1 *Consider an array $A = [a_1, a_2, \ldots, a_n]$ of n distinct elements. A sorting algorithm using only consecutive comparisons requires at least $\Omega(n^2)$ comparisons.*

Proof: An adjacent comparison between a_i and a_{i+1} can identify or resolve an *inversion*—a pair where $a_i > a_{i+1}$ in the current permutation, which must be swapped to achieve sorted order ($a_1 < a_2 < \cdots < a_n$). Each comparison either: (1) confirms $a_i < a_{i+1}$ (no swap needed), or (2) detects $a_i > a_{i+1}$, potentially leading to a swap (e.g., in Bubble Sort).

The maximum number of inversions in a permutation of n elements is achieved by the reverse order $[n, n-1, \ldots, 2, 1]$, with:

$$\text{Number of inversions} = \binom{n}{2} = \frac{n(n-1)}{2}.$$

Each adjacent comparison can resolve at most one inversion (e.g., swapping a_i and a_{i+1}), so a permutation with $\frac{n(n-1)}{2}$ inversions requires at least that many comparisons to sort fully.

■

8.3 Merge Sort

Merge Sort is the classic example of divide-and-conquer method of designing algorithms. To sort an array, A, assume that the left-half and the right-half of the arrays are sorted. Then we are simply left with the task of merging these two sorted halves. How do we sort the left and the right halves? By using recursion with the base case when the halves have only single elements and are already sorted. So, it is sufficient to consider the following problem: we have two sorted arrays B and C, each of size n. We would like to merge them into another array D such that D is sorted.

Algorithm MergeSort-Seq: Sequential Merge Sort

```
1  MergeSort(A, low, high)
2  if low < high then
3  |  mid ← ⌊(low + high)/2⌋
4  |  B ← MergeSort(A, low, mid)
5  |  C ← MergeSort(A, mid, high)
6  |  return MergeTwo(B, C)
7  end
8  else
9  |  return A[low]
10 end
```

It is easy to design a sequential algorithm that merges them two arrays B and C into D. We simply keep two indices i and j in arrays B and C, respectively. At any step of the algorithm, we compare $B[i]$ and $C[j]$. If $B[i]$ is smaller than $C[j]$, then we copy $B[i]$ into the next available slot in D and advance index i. If $C[j]$ is smaller than $B[i]$, then we copy $C[j]$ into the next available slot in D and advance index j. This algorithm takes $O(n)$ time.

Algorithm MergeTwo: Merging Two Sorted Arrays

```
1  var
2      B[1..m], C[1..n]: arrays of int ;                    // Sorted inputs
3      D[1..m + n]: array of int ;                          // Output
4  i, j, k ← 1, 1, 1
5  while i ≤ m and j ≤ n do
6  |   if B[i] < C[j] then
7  |   |   D[k] ← B[i]
8  |   |   i ← i + 1
9  |   end
10 |   else
11 |   |   D[k] ← C[j]
12 |   |   j ← j + 1
13 |   end
14 |   k ← k + 1
15 end
16 while i ≤ m do
17 |   D[k] ← B[i]; i ← i + 1; k ← k + 1
18 end
19 while j ≤ n do
20 |   D[k] ← C[j]; j ← j + 1; k ← k + 1
21 end
```

Sequentially, $MergeTwo$ takes $O(m+n)$ time. For n total elements, Merge Sort's recurrence is $T(n) = 2T(n/2) + O(n)$, yielding $O(n \log n)$ time.

8.4 Quicksort

Quicksort (due to C.A.R. Hoare) is one of the fastest sorting algorithms on sequential computers. It has $O(n^2)$ worst case time complexity but requires $O(n \log n)$ time on average. The algorithm is based on first partitioning the array into two parts based on a *pivot*. Once we have the property that the lower half of the array is less than or equal to pivot and the upper half of the array has elements greater than pivot, then we can recurse on each half. Since sorting on each half is independent, they can be sorted in parallel.

There are multiple methods to choose a pivot to partition the array A. Choosing an element at random is an easy method that will result in approximately equal sized partitions on average. This gives us the average case sequential time complexity of $O(n \log n)$. In the worst case, however, the recursion may reduce the range by only 1, resulting in the sequential time compexity of $O(n^2)$.

Algorithm QuickSort-Seq: Sequential QuickSort

```
1 QuickSort(A, low, high)
2 if low < high then
3 |   pivot ← choosePivot()
4 |   (p, q) ← Partition(A, pivot, low, high)
5 |   QuickSort(A, low, p)
6 |   QuickSort(A, q, high)
7 end
```

We describe a slight variant of the Quicksort algorithm in which we partition the array into three parts. Algorithm Three-Way-Partition takes array A, low and $high$ as input parameters. The part of the array this method sorts is given by

$$\{A[i] \mid low \leq i < high\}$$

Note that when low equals $high$, the range is empty.

The method *partition* returns two indices p and q. All elements in the range $[lo \ldots p)$ are strictly less than the pivot, in the range $[p \ldots q)$ are equal to the pivot and in the range $[q \ldots high)$ are greater than the pivot. We only need to recurse on the first and the third part. Depending upon the pivot, any (or both!) of the first and the third parts may be empty.

Once we have a pivot, how do we partition the array into three parts: the first partition with all elements less than the pivot, the second one with all elements equal to the pivot and the the third partition with elements strictly greater than the pivot. Algorithm Three-Way-Partition is due to Dijkstra who called this problem as the Dutch National Flag problem. The *while* loop maintains the invariant that entries $[low \ldots p)$ are less than pivot, $[p \ldots q)$ are equal to pivot and $[k \ldots high)$ are greater than pivot. The algorithm initializes p and q to low, therefore the first two ranges are empty initially and trivially satisfies the invariants. It also initializes k to $high$ making the range $[k \ldots high)$ empty and thereby ensuring that the invariant holds initially. The range $[q..k)$ corresponds to the initial input and initially contains entries that may be less than, equal to, or greater than the pivot. In each iteration of the *while* loop, this range is reduced by one by either increasing q or decreasing k. Depending upon the comparison between $A[q]$ and the pivot, the entry $A[q]$ is placed in the appropriate range maintaining the invariants.

Algorithm Three-Way-Partition: Three-Way Partition (Dutch National Flag)

```
1  Partition(A, pivot, low, high) returns (int, int)
2  p, q ← low
3  k ← high
4  while q < k do
5  |  if A[q] < pivot then
6  |  |  swap(A[p], A[q])
7  |  |  p ← p + 1
8  |  |  q ← q + 1
9  |  end
10 |  else if A[q] > pivot then
11 |  |  k ← k − 1
12 |  |  swap(A[q], A[k])
13 |  end
14 |  else
15 |  |  q ← q + 1
16 |  end
17 end
18 return (p, q)
```

It is easy to verify that the algorithm takes $O(n)$ time when the range has n elements. A parallel version for *QuickSort* based on divide and conquer is discussed in Chapter 9.

We prove that any comparison-based sorting algorithm for an array of n distinct elements requires at least $\Omega(n \log n)$ comparisons in the worst case. This result applies to algorithms using unrestricted pairwise comparisons (e.g., Merge Sort), contrasting with the $\Omega(n^2)$ bound for consecutive-only comparisons. We use a decision-tree model.

Theorem 8.2 *Given an array $A = [a_1, a_2, \ldots, a_n]$ of n distinct elements, a comparison-based sorting algorithm uses binary comparisons ($a_i < a_j$) to sort A into $a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$, where π is a permutation. The minimum number of comparisons needed to sort any input permutation in the worst case is $\Omega(n \log n)$.*

Proof: We model the sorting algorithm as a decision tree, where internal nodes represent comparisons, and leaves correspond to permutations. The worst-case number of comparisons is the tree's height.

For n distinct elements, there are $n!$ possible permutations. The algorithm must distinguish all $n!$ permutations, so the decision tree requires at least $n!$ leaves.

In a comparison-based algorithm, each node compares two elements (a_i vs. a_j), branching on $<$ or $>$. With distinct elements, each node has two children (binary tree). After k comparisons, the maximum number of leaves is 2^k.

To cover all permutations:

$$2^k \geq n!.$$

Taking the base-2 logarithm:

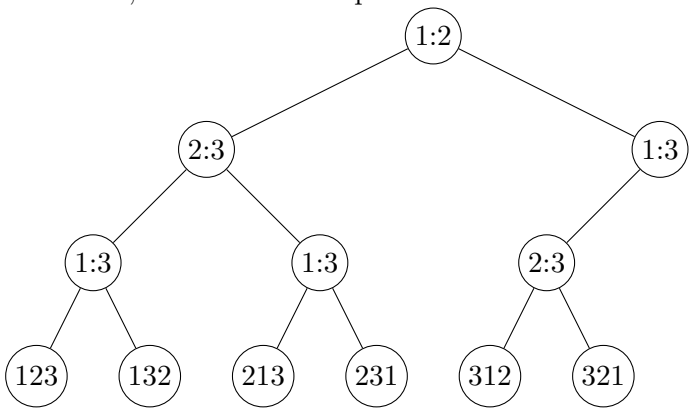
$$k \geq \log_2(n!).$$

The value $\log_2(n!)$ represents the minimum comparisons needed. For large n, it is well-known that:

$$\log_2(n!) = \Omega(n \log n),$$

since $n!$ grows factorially, and its logarithm reflects the information required to order n elements. This can be seen intuitively: sorting halves the problem repeatedly (e.g., in Merge Sort), requiring $\log n$ levels, each with $O(n)$ comparisons.

For $n = 3$, there are $3! = 6$ permutations. A decision tree might start with $a_1 < a_2$:



Since $k \geq \log_2(n!)$, and $\log_2(n!) = \Omega(n \log n)$:

$$k = \Omega(n \log n).$$

This is the minimum number of comparisons in the worst case, as the tree must be tall enough to reach $n!$ leaves.

■

8.5 Radix Sort

All our sorting algorithms, so far, have been based on comparison. Any comparison-based sorting algorithm must do at least $O(n \log n)$ work. We now show an algorithm that works when keys for sorting have a fixed number of digits (in any radix r, generally a power of 2). In our examples below, we simply use digits to the base 10, as they are easy for us humans. Suppose that we need to sort the following list. $[85, 72, 94, 45, 13, 12, 61, 81]$. Here, our keys have two digits. The idea of the radix sort is to sort numbers, one digit at a time. When sorting on a single digit, we use a simple linear time sort by exploiting that there are only r possibilities for each digit. On getting any number, we can simply add it to the pile associated with that value. The number of passes we will make on the array is equal to the number of digits.

It may appear that it is easier to sort starting from the most significant digit (msd) first. If we employed that strategy, we would get $[13, 12, 45, 61, 72, 85, 81, 94]$ after the first pass. The second pass would consist of sorting all subarrays with the same msd and we would get the array $[12, 13, 45, 61, 72, 81, 85, 94]$. The problem with this approach is that we are forced to maintain different subarrays - one for each digit after the first pass. With every subsequent pass, it gets even more cumbersome. Hence, somewhat counterintuitively, we will employ the least significant digit first strategy. After the first pass, $[61, 81, 72, 12, 13, 94, 85, 45]$. We do not need to remember any sublists that are created during the first pass. Now, we sort on the second least significant digit. We need to ensure that if two numbers have the same digit, then their order is

preserved. In other words, we require our sorting algorithms at each pass to be *stable*. After the second pass, we get the sorted array $[12, 13, 45, 61, 72, 81, 85, 94]$. Since 12 appeared before 13 after the first pass, the order is preserved after the second pass.

Thus, the sequential algorithm is simply stated as Algorithm RadixSort.

Algorithm RadixSort: Radix Sort

```
1 RadixSort(A[1..n])
2 for i = 1 to k do
3 |  StableSort(A, digit i)
4 end
```

The time complexity of this algorithm is $O(kn)$ assuming that the stable sort is accomplished in $O(n)$ time. For $r = 10$, sorting $[85, 72, 94, 45, 13, 12, 61, 81]$ yields $O(kn)$ time (here, $k = 2$, $O(n)$ per pass).

8.6 Summary

The following table lists all the algorithms for comparison-based sorting discussed in this chapter.

Problem	Algorithm	Time
Sorting	Bubble Sort	$O(n^2)$
Sorting	Insertion Sort	$O(n^2)$
Sorting	Merge Sort	$O(n \log n)$
Sorting	QuickSort	$O(n^2)$ worst, $O(n \log n)$ avg
Sorting	Radix Sort	$O(kn)$

8.7 Problems

1. Implement the binary search algorithm discussed in Section 8.1.
2. Show that any algorithm that is based on comparison of consecutive entries must take $O(n^2)$ comparisons in the worst case.
3. Give an algorithm to merge k sorted arrays of size n into a single sorted array.
4. We have partitioned the array into three parts in the QuickSort algorithm. Give a version of the QuickSort algorithm in which the array is partitioned only in two parts: entries that are less than *pivot* and the entries that are greater than or equal to the pivot.
5. Give a randomized version of Quicksort in which the pivot is chosen at random. Give the expected running time of your algorithm.

8.8 Bibliographic Remarks

Quicksort is a divide-and-conquer algorithm that was invented by Tony Hoare in 1962 [Hoa61] It is one of the most widely used sorting algorithms and has a time complexity of $O(n \log n)$ in the average case. Merge sort is another divide-and-conquer algorithm that was first described by John von Neumann in 1945. Radix sort is a non-comparison-based sorting algorithm that was first described by Herman Hollerith in 1887.

Chapter 9

Divide and Conquer

9.1 Introduction

A useful strategy for solving a problem is to divide the problem into sub-problems, solve the sub-problems and then merge the solutions to the sub-problems to get the solution of the original problem. When the sub-problem is of trivial size, then it can simply be solved by case analysis. We give many examples of this approach.

9.2 Mergesort: Revisited

To sort an array of size n, we divide it into two subarrays of size $n/2$ and sort each of the parts. The main trick is to determine how to merge these two sorted arrays.

Algorithm MergeSort: MergeSort

Input: Array A, indices low, $high$
Output: Sorted array $A[low..high]$

```
1  MergeSort(A, low, high)
2  if low < high then
3  |  mid ← ⌊(low + high)/2⌋;
4  |  B ← MergeSort(A, low, mid);
5  |  C ← MergeSort(A, mid + 1, high);
6  |  return MergeTwo(B, C);
7  end
8  else
9  |  return A;
10 end
```

Now, we merge the two sorted subarrays to get the sorted version of the original array. Let us first look at the sequential time complexity. We have the following recurrence

$$T(n) = 2T(n/2) + O(n); T(1) = O(1)$$

Solving this recurrence, we get the sequential time complexity as $T(n) = O(n \log n)$.

9.3 The Master Theorem

The Master Theorem provides a method for solving recurrence relations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^c)$$

where:

- $a \geq 1$ is the number of subproblems,
- $b > 1$ is the factor by which the problem size is reduced,
- $O(n^c)$ (with $c \geq 0$) is the cost of work done outside the recursive calls.

The time complexity is given by:

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } c < \log_b a \\ O(n^c \log n) & \text{if } c = \log_b a \\ O(n^c) & \text{if } c > \log_b a \end{cases}$$

Proof of Correctness

We prove the Master Theorem using recursion tree expansion.

Step 1: Expand the Recursion

$$\begin{aligned} T(n) &= aT(n/b) + O(n^c) \\ &= a\left(aT(n/b^2) + O((n/b)^c)\right) + O(n^c) \\ &= a^2T(n/b^2) + O(a(n/b)^c) + O(n^c) \end{aligned}$$

For k levels:

$$T(n) = a^kT(n/b^k) + \sum_{i=0}^{k-1} O(a^i(n/b^i)^c)$$

The sum is:

$$\sum_{i=0}^{k-1} a^i(n/b^i)^c = n^c \sum_{i=0}^{k-1} \left(\frac{a}{b^c}\right)^i$$

Step 2: Stopping Condition

The recursion stops when $n/b^k = 1$, so $k = \log_b n$:

$$T(n) = a^{\log_b n}T(1) + O\left(n^c \sum_{i=0}^{\log_b n - 1} \left(\frac{a}{b^c}\right)^i\right)$$

Since $a^{\log_b n} = n^{\log_b a}$ and $T(1) = O(1)$, the first term is $O(n^{\log_b a})$.

Step 3: Analyze the Sum

For $r = a/b^c$:

- If $c < \log_b a$, then $r > 1$, the sum is $O(n^{\log_b a})$, and $T(n) = O(n^{\log_b a})$.
- If $c = \log_b a$, then $r = 1$, the sum is $O(\log n)$, and $T(n) = O(n^c \log n)$.
- If $c > \log_b a$, then $r < 1$, the sum is $O(n^c)$, and $T(n) = O(n^c)$.

Example 1: Merge Sort (Case 2)

$$T(n) = 2T(n/2) + O(n)$$

$a = 2$, $b = 2$, $c = 1$. Since $\log_2 2 = 1 = c$, case 2 applies:

$$T(n) = O(n \log n)$$

Example 2: Binary Search (Case 1)

$$T(n) = T(n/2) + O(1)$$

$a = 1$, $b = 2$, $c = 0$. Since $\log_2 1 = 0 = c$, case 2 applies (noting $O(n^0) = O(1)$):

$$T(n) = O(\log n)$$

Example 3: Strassen's Matrix Multiplication (Case 1)

$$T(n) = 7T(n/2) + O(n^2)$$

$a = 7$, $b = 2$, $c = 2$. Since $\log_2 7 \approx 2.81 > 2$, case 1 applies:

$$T(n) = O(n^{2.81})$$

Example 4: Karatsuba's Algorithm (Case 1)

$$T(n) = 3T(n/2) + O(n)$$

$a = 3$, $b = 2$, $c = 1$. Since $\log_2 3 \approx 1.58 > 1$, case 1 applies:

$$T(n) = O(n^{1.58})$$

Example 5: (Case 3)

$$T(n) = 2T(n/2) + O(n^2)$$

$a = 2$, $b = 2$, $c = 2$. Since $\log_2 2 = 1 < 2$, case 3 applies. Thus:

$$T(n) = O(n^2)$$

Additional Examples

(a) $T(n) = 2T(n/2) + O(n^2)$

$a = 2$, $b = 2$, $c = 2$. Since $\log_2 2 = 1 < 2$, case 3 applies.

$$T(n) = O(n^2)$$

(b) $T(n) = 9T(n/3) + O(n)$

$a = 9$, $b = 3$, $c = 1$. Since $\log_3 9 = 2 > 1$, case 1 applies:

$$T(n) = O(n^2)$$

(c) $T(n) = T(2n/3) + O(1)$

$a = 1$, $b = 3/2$, $c = 0$. Since $\log_{3/2} 1 = 0 = c$, case 2 applies:

$$T(n) = O(\log n)$$

9.4 Nearest Neighbors in the Euclidean Space

We are given n points in the two-dimensional Euclidean space. Our goal is to find a pair of points with the smallest distance between them. That is, determine two points (p_i, p_j) such that the Euclidean distance between them is minimized:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

The easiest algorithm would be to compute the distance between every pair of points and choose a pair with the least distance. This approach requires $O(n^2)$ sequential time. Our goal is to reduce the sequential time complexity for this problem.

The divide-and-conquer approach for this problem is as follows. We divide the set of points into two sets S_1 and S_2. Suppose that we have the nearest neighbors in S_1 and S_2. Now, if we had the nearest neighbors such that one point is in S_1 and the second point is in S_2, then we simply have to choose the smallest of these three distances: nearest neighbors in S_1, nearest neighbors in S_2, and the nearest neighbors in across the partition. Suppose the distance of the nearest neighbors in S_1, or in S_2 is δ. Then, we only need to consider nearest neighbors across the partition if they are closer than δ. Let S_y be the set of points within δ of the dividing line sorted in the y-coordinate. We claim that if $s, s' \in S$ are such that $d(s, s') < \delta$, then s and s' are within 15 positions of each other in the sorted list S. Let L be the vertical line that splits the set of points based on x-coordinate. We consider boxes of size $\delta/2$ around L. We first show that there cannot be two points within a box. The largest distance within a box is $\sqrt{2}(\delta/2)$. This is equal to $\delta/\sqrt{2}$ which is less than δ.

We next claim that there cannot be two points that are sixteen positions apart in S_y and still have distance less than δ. Since the points are sixteen positions apart and at most one point in each box, we get that there are at least three rows of boxes between these two points. However, this means that the distance between them is at least $3 * \delta/2$ which is greater than δ.

Hence, it is sufficient to consider points that are at most 15 positions apart in S_y.

Algorithm Closest-Pair: Closest Pair of Points

```
Input: Set of n points P in 2D space
Output: The closest pair of points
Function ClosestPair(P):
    if n ≤ 3 then
        return BruteForce(P) // Base case: use brute-force for small input
    end
    Sort P by x-coordinate
    Split P into left (P_L) and right (P_R) halves
    (p1, p2) ← ClosestPair(P_L)
    (p3, p4) ← ClosestPair(P_R)
    δ ← min(d(p1, p2), d(p3, p4))
    S ← points within δ of the dividing line
    Sort S by y-coordinate
    for each point p in S do
        for each of the next 15 points q do
            if d(p, q) < δ then
                Update δ and closest pair
            end
        end
    end
    return closest pair
```

Consider the following set of points:

$$P = \{(1, 2), (3, 7), (5, 4), (8, 9), (9, 6), (10, 3)\}$$

Now the algorithm can be summarized as:
1. **Sorting:** Arrange by x-coordinate: $[(1,2),(3,7),(5,4),(8,9),(9,6),(10,3)]$
2. **Divide:** Split into two halves:

$$P_L = [(1,2),(3,7),(5,4)]$$
$$P_R = [(8,9),(9,6),(10,3)]$$

3. **Recursive Closest Pair:** - Left half: Closest pair is $(1,2)$ and $(5,4)$ with $d = 4.47$. - Right half: Closest pair is $(9,6)$ and $(10,3)$ with $d = 3.16$.
4. **Combine Step:** - Points near the vertical dividing line: are $(5,4),(8,9),(9,6)$. - Checking only points within $\delta = 3.16$:
- $(5,4)$ and $(9,6)$ have $d = 4.47$ (not better).
- $(8,9)$ and $(9,6)$ have $d = 3.16$ (same as right half).
- $(5,4)$ and $(9,6)$ have $d = 2.24$ (new minimum).
5. **Final Output:** $(5,4)$ and $(9,6)$ with distance 2.24.

The time complexity is as follows. For sorting the points, we spend $O(n \log n)$ time. To recursively solving subproblems, we use $O(n \log n)$ time. For checking points in the strip we spend $O(n)$ time. Thus, the overall time complexity is $O(n \log n)$.

9.5 Counting Inversions in an Array using Divide and Conquer

Given an array $A[1 \dots n]$, an **inversion** is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The goal is to count the total number of inversions in the array efficiently. A brute-force approach iterates through all pairs and checks if $A[i] > A[j]$, requiring $O(n^2)$ time. Using Merge Sort with an additional counting step, we can solve this problem in $O(n \log n)$ time.

We modify the *Merge Sort* algorithm to count inversions efficiently: (1) Recursively count inversions in the left and right halves. (2) Count the number of inversions across the two halves while merging. (3) Sum up all these counts.

```
Input: Array A of size n
Output: Total number of inversions
Function CountInversions(A, left, right):
    if left ≥ right then
        return 0 // Base case:  single element has no inversions
    end
    mid ← (left + right)/2
    invLeft ← CountInversions(A, left, mid)
    invRight ← CountInversions(A, mid + 1, right)
    invMerge ← MergeAndCount(A, left, mid, right)
    return invLeft + invRight + invMerge
Function MergeAndCount(A, left, mid, right):
    // left as an exercise
```

Consider the array:

$$A = [8,4,2,1]$$

Step-by-step inversion count:

- Left half $[8,4]$: 1 inversion $(8,4)$.
- Right half $[2,1]$: 1 inversion $(2,1)$.
- Merging left and right halves:

– $(8,2),(8,1),(4,2),(4,1)$ contribute 4 more inversions.

Total inversions: 6.

It is easy to see that the merge sort recursion takes $O(n \log n)$ time. The Merge step inversion counting takes $O(n)$ time. Thus, the overall time complexity is $O(n \log n)$.

9.6 Planar Convex Hull

Suppose that we are given S, a set of points on a plane. The convex hull of S is the smallest convex polygon that contains all points of S. The convex polygon can be described by the ordered list of points (in the clockwise direction) that defines the boundary of the polygon. Our task is to compute the convex hull of S.

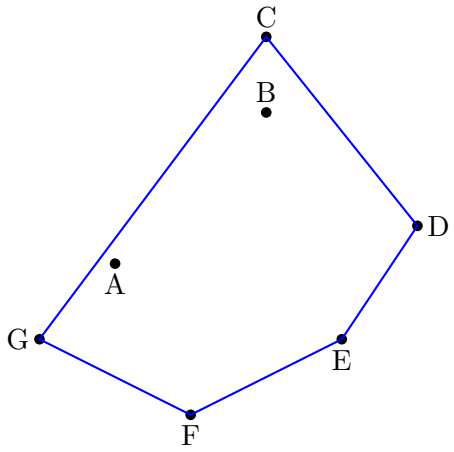


Figure 9.1: An example of Planar Convex Hull

Let points $\{p_1, ..., p_n\}$ be sorted in their x-coordinate. For simplicity, we assume that x coordinates are unique. It is easy to verify that the point with the smallest x-coordinate, p_1 and the largest x-coordinate p_n are always in the convex hull. These two points also divide up the convex hull into two sets: the upper hull and the lower hull. The upper hull consists of all points in the convex hull starting from p_1 and ending at p_n (but not including p_n) in the clockwise direction. The lower hull consists of all points starting from p_n and ending at p_1 (but not including p_1).

To compute the convex hull of S, we divide the set of points into two sets S_1 and S_2 such that S_1 has first $n/2$ points with smaller x coordinate and S_2 has $n/2$ points with the larger x coordinate. Similar to mergesort, we can compute convex hulls of S_1 and S_2. The only task left is to merge the convex hull of S_1 and S_2 to get the convex hull of S. We first get the upper hulls of S_1 and S_2 and merge them to get the upper hull of S. This procedure can be repeated for lower hulls. To merge upper hulls, we need to determine the upper common tangent of $UH(S_1)$ and $UH(S_2)$. It is known that the upper common tangent can be determined sequentially in $O(n)$ time. Hence, we get the following recurrence:

$$T(n) \leq 2T(n/2) + O(n)$$

Solving this recurrence, we get $T(n) = O(n \log n)$.

9.7 Karatsuba's Multiplication Algorithm

A notable application of divide and conquer approach is Karatsuba's multiplication algorithm, which computes the product of two large integers faster than the classical grade-school method. Introduced by Anatoly Karatsuba in 1960, it reduces the complexity from $O(n^2)$ to $O(n^{\log_2 3}) \approx O(n^{1.585})$, where n is the number of digits.

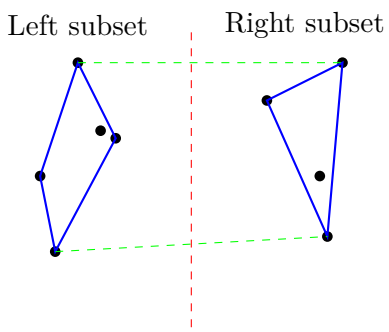


Figure 9.2: Using Divide and Conquer for Planar Convex Hull

Consider multiplying two polynomials $A(x) = a_0 + a_1 x$ and $B(x) = b_0 + b_1 x$, where coefficients are single-digit numbers (e.g., base-10 digits). Their product is:

$$C(x) = A(x) \cdot B(x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + a_1 b_1 x^2.$$

Naive multiplication computes each term $(a_0 b_0, a_0 b_1, a_1 b_0, a_1 b_1)$, requiring 4 multiplications and 1 addition, yielding $O(n^2)$ for degree-n polynomials.

Now, represent two n-digit numbers $X = x_1 \cdot 10^{n/2} + x_0$ and $Y = y_1 \cdot 10^{n/2} + y_0$ (assuming n is even), where x_1, x_0, y_1, y_0 are $n/2$-digit numbers. Their product is:

$$X \cdot Y = (x_1 \cdot 10^{n/2} + x_0)(y_1 \cdot 10^{n/2} + y_0) = x_1 y_1 \cdot 10^n + (x_1 y_0 + x_0 y_1) \cdot 10^{n/2} + x_0 y_0,$$

akin to polynomial multiplication with $x = 10^{n/2}$. Naive computation requires 4 multiplications of $n/2$-digit numbers, suggesting $O(n^2)$ overall. Karatsuba's algorithm optimizes this to 3 multiplications.

Karatsuba's insight reduces the number of multiplications by computing: 1. $p_1 = x_0 y_0$, 2. $p_2 = x_1 y_1$, 3. $p_3 = (x_0 + x_1)(y_0 + y_1)$.

Then:

$$X \cdot Y = p_2 \cdot 10^n + [p_3 - (p_1 + p_2)] \cdot 10^{n/2} + p_1.$$

$p_3 = x_0 y_0 + x_0 y_1 + x_1 y_0 + x_1 y_1 = p_1 + p_2 + (x_1 y_0 + x_0 y_1)$, Thus, $p_3 - (p_1 + p_2) = x_1 y_0 + x_0 y_1$.

This requires only 3 multiplications (p_1, p_2, p_3) instead of 4, plus additional additions/subtractions, which are $O(n)$.

For n-digit numbers:

- Split: $X = x_1 \cdot 10^{n/2} + x_0$, $Y = y_1 \cdot 10^{n/2} + y_0$.
- Recursively compute p_1, p_2, p_3 on $n/2$-digit numbers.
- Combine using the formula above.

The recurrence is:

$$T(n) = 3T(n/2) + O(n),$$

solving to $T(n) = O(n^{\log_2 3})$ via the Master Theorem (Case 1: $a = 3, b = 2, \log_b a \approx 1.585$).

Algorithm Karatsuba: Karatsuba's Multiplication Algorithm

Input: Numbers X, Y, digit length n (power of 2)
Output: Product $X \cdot Y$
if $n = 1$ **then**
  return $X \cdot Y$ `// base case: single-digit multiplication`
end
$x_1 := \lfloor X/10^{n/2} \rfloor$, $x_0 := X \mod 10^{n/2}$ `// high and low halves of` X
$y_1 := \lfloor Y/10^{n/2} \rfloor$, $y_0 := Y \mod 10^{n/2}$ `// high and low halves of` Y
$p_1 := \text{Karatsuba}(x_0, y_0, n/2)$ `// low product`
$p_2 := \text{Karatsuba}(x_1, y_1, n/2)$ `// high product`
$p_3 := \text{Karatsuba}(x_0 + x_1, y_0 + y_1, n/2)$ `// middle term helper`
return $p_2 \cdot 10^n + [p_3 - (p_1 + p_2)] \cdot 10^{n/2} + p_1$

Consider $X = 1234$, $Y = 5678$, $n = 4$ (pad to power of 2 if needed).

Step 1: Divide

- $10^{n/2} = 10^2 = 100$,
- $x_1 = \lfloor 1234/100 \rfloor = 12$, $x_0 = 1234 \mod 100 = 34$,
- $y_1 = \lfloor 5678/100 \rfloor = 56$, $y_0 = 5678 \mod 100 = 78$.

Step 2: Conquer (Recurse)

- $p_1 = 34 \cdot 78 = 2652$ (base case, $n/2 = 2$),
- $p_2 = 12 \cdot 56 = 672$ (base case),
- $p_3 = (34 + 12) \cdot (78 + 56) = 46 \cdot 134 = 6164$ (base case).

Step 3: Combine

- $p_3 - (p_1 + p_2) = 6164 - (2652 + 672) = 6164 - 3324 = 2840$,
- $X \cdot Y = 672 \cdot 10^4 + 2840 \cdot 10^2 + 2652 = 6720000 + 284000 + 2652 = 7006652$.

Verification: $1234 \cdot 5678 = 7006652$, correct.

Karatsuba's algorithm reduces multiplications from 4 to 3 per level, trading them for additions/subtractions. For $n = 2^k$:
- Levels: $k = \log_2 n$,
- Multiplications: $3^k = 3^{\log_2 n} = n^{\log_2 3}$,
- Additions: $O(n \log n)$.

Compared to FFT ($O(n \log n)$), Karatsuba is slower but avoids complex numbers, making it simpler for integer arithmetic. The example shows its practicality for small numbers, scaling efficiently for larger n.

9.8 Strassen's Matrix Multiplication

Strassen's algorithm is a divide-and-conquer approach to matrix multiplication that reduces the time complexity from the naive $O(n^3)$ to $O(n^{\log_2 7}) \approx O(n^{2.807})$ for multiplying two $n \times n$ matrices. Unlike the standard method, which performs 8 recursive multiplications for 2×2 submatrices, Strassen's method uses 7 multiplications by cleverly combining submatrix operations, trading some multiplications for additional additions.

Algorithm Strassen: Strassen's Matrix Multiplication

Data: A, B: $n \times n$ matrices, n a power of 2
Result: C: $n \times n$ matrix, $C = A \cdot B$

if $n = 1$ **then**
  $C[1,1] \leftarrow A[1,1] \cdot B[1,1]$;
  Return C ;
end
Divide A into $A_{11}, A_{12}, A_{21}, A_{22}$; // Each $n/2 \times n/2$
Divide B into $B_{11}, B_{12}, B_{21}, B_{22}$; // Each $n/2 \times n/2$

$M_1 \leftarrow \text{Strassen}(A_{11} + A_{22}, B_{11} + B_{22})$;
$M_2 \leftarrow \text{Strassen}(A_{21} + A_{22}, B_{11})$;
$M_3 \leftarrow \text{Strassen}(A_{11}, B_{12} - B_{22})$;
$M_4 \leftarrow \text{Strassen}(A_{22}, B_{21} - B_{11})$;
$M_5 \leftarrow \text{Strassen}(A_{11} + A_{12}, B_{22})$;
$M_6 \leftarrow \text{Strassen}(A_{21} - A_{11}, B_{11} + B_{12})$;
$M_7 \leftarrow \text{Strassen}(A_{12} - A_{22}, B_{21} + B_{22})$;

$C_{11} \leftarrow M_1 + M_4 - M_5 + M_7$;
$C_{12} \leftarrow M_3 + M_5$;
$C_{21} \leftarrow M_2 + M_4$;
$C_{22} \leftarrow M_1 - M_2 + M_3 + M_6$;

Return $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$;

Given two $n \times n$ matrices A and B, where n is a power of 2 (padded with zeros if necessary), Strassen's algorithm divides each matrix into four $n/2 \times n/2$ submatrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

The product $C = A \cdot B$ is computed as:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

where traditionally:

- $C_{11} = A_{11}B_{11} + A_{12}B_{21}$,
- $C_{12} = A_{11}B_{12} + A_{12}B_{22}$,
- $C_{21} = A_{21}B_{11} + A_{22}B_{21}$,
- $C_{22} = A_{21}B_{12} + A_{22}B_{22}$.

This requires 8 multiplications and 4 additions of $n/2 \times n/2$ matrices. Strassen's insight is to define 7 intermediate products (M_1 to M_7) that allow the computation of C with fewer multiplications:

The recurrence relation for Strassen's algorithm is:

$$T(n) = 7T(n/2) + O(n^2),$$

where 7 is the number of recursive multiplications, and $O(n^2)$ accounts for the additions/subtractions of $n/2 \times n/2$ matrices. Using the Master Theorem ($a = 7$, $b = 2$, $f(n) = O(n^2)$, $\log_b a = \log_2 7 \approx 2.807 > 2$):

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.807}).$$

This is an improvement over the standard $O(n^3)$.

9.9 Summary

In this chapter, we have shown that many problems can be solved using the divide-and-conquer approach. The following table summarizes some problem discussed in this chapter.

Problem	**Time Complexity**	**Description**
MergeSort	$O(n \log n)$	Splits array and merges sorted halves.
Closest Pair	$O(n \log n)$	Closest pair in 2D using strip merging.
Counting Inversions	$O(n \log n)$	Counts inversions via MergeSort.
Convex Hull	$O(n \log n)$	Computes smallest convex polygon.
Karatsuba Multiplication	$O(n^{\log_2 3})$	Multiply two integers.
Strassen Matrix Multiplication	$O(n^{\log_2 7})$	Multiply two matrices.

Table 9.1: Sequential Divide-and-Conquer Algorithms

9.10 Bibliographic Remarks

The general divide-and-conquer strategy and the Master Theorem are thoroughly treated in [CLRS09]. Strassen's Matrix Multiplication (Section 10), first proposed by [Str69], reduces the complexity to $O(n^{2.807})$. Karatsuba's Multiplication introduced by [KO62], achieves $O(n^{1.585})$. The Closest Pair problem leverages a divide-and-conquer approach from [SH75], optimized to $O(n \log n)$.

Counting Inversions via Mergesort, achieving $O(n \log n)$, is a standard application. The Planar Convex Hull builds on [Gra79], with its $O(n \log n)$.

For further reading, we refer the readers to [Hoa62], [Str69], [KO62], [SH75], and [Gra79].

9.11 Problems

1. Solve the problem of computing the Fast-Fourier-Transform in using the divide and conquer approach.

Chapter 10

Dynamic Programming

10.1 Introduction

Dynamic programming is applicable to problems where it is easy to set up a recurrence relation such that the solution of the problem can be derived from the solutions of problems with smaller sizes. The problem can be solved using recursion; however, recursion can result in many duplicate computations. By using memoization, we can avoid recomputing previously computed values. We assume that the problem is solved using dynamic programming with such bottom-up approach in this chapter.

The LLP algorithm views solving a problem as searching for an element in a finite distributive lattice [Bir67, DP90, Gar15] that satisfies a given predicate B. The predicate is required to be lattice-linear. For all the problems considered in this chapter, the longest subsequence problem, the optimal binary search tree problem, and the knapsack problem, this is indeed the case.

There are also some key differences between dynamic programming (the bottom-up approach) and the LLP algorithm. The usual dynamic programming problem seeks a structure that minimizes (or maximizes) some scalar. For example, the longest subsequence problem asks for the subsequence in an array $A[1..n]$ that maximizes the sum. In contrast, the LLP algorithm seeks to minimize or maximize a *vector*. In the longest subsequence problem with the LLP approach, we are interested in the longest subsequence in the array $A[1..i]$ for each $i \leq n$ that ends at the index i. Thus, instead of asking for a scalar, we ask for the vector of size n. We get an array $G[1..n]$ and the solution to the original problem is just the maximum value in the array G. Similarly, the optimal binary search tree problem [Knu71] asks for the construction of an optimal binary search tree on n symbols such that each symbol i has probability p_i of being searched. Our goal is to find the binary search tree that minimizes the expected cost of search in the tree. The LLP problem seeks the optimal binary search tree for all ranges $i \ldots j$ instead of just one range $1..n$. Finally, the knapsack problem [HS74, IK75] asks for the maximum valued subset of items that can fit in a knapsack so that the profit is maximized and the total weight of the knapsack is at most W. The LLP problem seeks the maximum profit obtained by choosing items from $1..i$ and the total weight from $1..W$. In all these problems, traditionally we are seeking a single structure that optimizes a single scalar; whereas the LLP algorithm asks for a vector. It turns out that in asking for an optimal *vector* instead of an optimal *scalar*, we do not lose much since the existing solutions also end up finding the optimal solutions for the subproblems. The LLP algorithm returns a vector G such that $G[i]$ is optimal for i.

Yet another difference between dynamic programming and the LLP algorithm is that we can use the LLP algorithm to solve a constrained version of the problem, so long as the constraint itself is lattice-linear. Suppose that we are interested in the longest subsequence such that successive elements differ by at least 2. It can be (easily) shown that this constraint is lattice-linear. Hence, the LLP algorithm is applicable because we are searching for an element that satisfies a conjunction of two lattice-linear predicates. Since the set of lattice-linear predicates is closed under conjunction, the resulting predicate is also lattice-linear and the LLP algorithm is applicable. Similarly, the

predicate that the symbol i is not a parent of symbol j is lattice-linear and the constrained optimal binary search tree algorithm returns the optimal tree that satisfies the given constraint. In the Knapsack problem, it is easy to solve the problem with the additional constraint that if the item x is included in the Knapsack, then the item y is also included.

This chapter is organized as follows. Section 10.4 applies the LLP method to the longest subsequence problem. Section 10.6 gives an LLP algorithm for the optimal binary search tree construction problem. Section 10.10 gives an LLP algorithm for the knapsack problem.

10.2 Recursion vs Dynamic Programming

Dynamic programming is applicable to problems in which it is easy to set up a recurrence relation so that the solution of the problem at hand can be derived from the solutions to problems with smaller sizes. One can solve the problem using recursion; however, recursion may result in many duplicate computations.

As a simple example, suppose that our goal is to compute the Fibonacci number F_i such that it satisfies the following conditions:

- $F_0 = 1$
- $F_1 = 1$
- $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$

A recursive algorithm is as follows:

```
F(n):
  if n < 2:
    return 1
  else:
    return F(n-1) + F(n-2)
```

If we use recursion to find $F(4)$, we will call $F(3)$ and $F(2)$. Now, $F(3)$ will call $F(2)$ and $F(1)$. Thus, $F(2)$ is being called twice.

Using *memoization* for dynamic programming, we avoid such duplicate computations. Memoization is used to remember previously computed values.

An implementation using memoization is as follows:

```
F(n):
  Table: map of (int, int)
  if (n, v) exists in Table for n:
    return v
  else:
    if n < 2:
      return 1
    else:
      answer = F(n-1) + F(n-2)
      Store (n, answer) in Table
```

Alternatively, instead of computing $F(n)$ in a top-down manner, one can compute it in a bottom-up fashion. Assume that we keep an array A such that $A[i]$ will eventually store $F(i)$. We start by filling in the array A from the lower to higher indices. The recurrence relation guarantees that if we have two previous entries, we can fill any entry $A[i]$. Thus, we can compute $F(n)$ in $O(n)$ time.

In this example, we had a simple array. For more non-trivial examples, we may have two-dimensional or three-dimensional tables.

10.3 Weighted Interval Scheduling

We revisit the problem of Interval Scheduling from Chapter 5. This time we consider a generalization of the problem in which each interval is assigned *weight* denoting its priority. We would like to compute a subset of intervals such that they are non-overlapping and have the maximum weight. In this generalization, we can no longer guarantee that the interval that finishes earliest and starts after the last chosen interval is always selected.

As before, we assume that the intervals are sorted according to their *finish* times. Let $Opt(j)$ denote the optimal value that can be obtained from the intervals $1 \ldots j$. When j is zero, the set of intervals selected is empty and $Opt(0)$ is zero. Let $p(j)$ be the largest interval before interval j which is disjoint from the interval j. The, we can set up a recurrence relation as follows. The maximum value that we can obtain from the intervals $1 \ldots j$ is equal to the maximum of two values obtained by including the interval j, or by not including the interval j. If we include the interval j, then the value obtained is $w_j + Opt(p(j))$. If we do not include the interval j, then the value obtained is $Opt(j-1)$. Thus, we have

$$Opt(j) = \max\{w_j + Opt(p(j)), Opt(j-1)\}$$

This recurrence relation allows us to obtain the algorithm for weighted interval scheduling as shown in Fig. WeightedIntervalSequential

Algorithm WeightedIntervalSequential: A Sequential Program for the Weighted Interval Scheduling problem

```
1  var G: array[0...n-1] of 0..1 initially 0;
2      // jobs are sorted by their finish times
3      const s: array[0...n-1] of int; // start times
4      const f: array[0...n-1] of int ; // finish times
5      const w: array[0...n-1] of int; //weight of the interval
6      p: array[0...n-1] of int; //previous non-overlapping interval
7      opt: array[0...n-1] of int; //optimal value
8  // p[j] is the largest interval i such that f[i] < s[j]
9  p[0] := 0;
10 // Compute p[j] as the largest i such that f[i] < s[j]; // Since f is sorted, we can compute all p[j]
     in O(log n) time using binary search
11 opt[0] := 0;
12 for int cur := 1 to n-1 do
13     opt[cur] := opt[cur-1];
14     if (w[cur] + opt[p[cur]] ≥ opt[cur-1]) then
15         opt[cur] := w[cur] + opt[p[cur]]
16         G[cur] := 1;
17     else
18         G[cur] := 0;
```

The sequential algorithm takes $O(n)$ time when p array is available. This time complexity is optimal. Computing p array takes $O(n \log n)$ time.

Consider the following 5 intervals, sorted by finish time:

Interval	s_i	f_i	w_i
1	1	3	4
2	2	5	6
3	4	6	5
4	6	8	3
5	5	9	7

The intervals are shown in Fig. 10.1. We first compute $p(i)$ (latest non-overlapping interval before i):
- $p(1) = 0$ (none before 1)
- $p(2) = 0$ ($f_1 = 3 > s_2 = 2$)
- $p(3) = 1$ ($f_1 = 3 < s_3 = 4$, $f_2 = 5 > s_3$)
- $p(4) = 3$ ($f_3 = 6 = s_4$, compatible)
- $p(5) = 2$ ($f_2 = 5 = s_5$, compatible)

Now, we compute $OPT(i)$:
- $OPT(0) = 0$
- $OPT(1) = \max(OPT(0), w_1 + OPT(p(1))) = \max(0, 4 + 0) = 4$ (select 1)
- $OPT(2) = \max(OPT(1), w_2 + OPT(p(2))) = \max(4, 6 + 0) = 6$ (select 2)
- $OPT(3) = \max(OPT(2), w_3 + OPT(p(3))) = \max(6, 5 + 4) = 9$ (select 1, 3)
- $OPT(4) = \max(OPT(3), w_4 + OPT(p(4))) = \max(9, 3 + 9) = 12$ (select 1, 3, 4)
- $OPT(5) = \max(OPT(4), w_5 + OPT(p(5))) = \max(12, 7 + 6) = 13$ (select 2, 5)

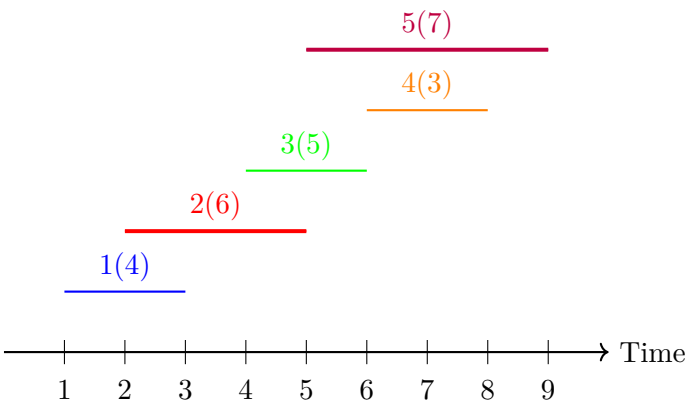


Figure 10.1: Weighted intervals (weights in parentheses) with optimal selection (thick lines).

10.4 Longest Increasing Subsequences

The Longest Increasing Subsequence (LIS) problem involves finding a subsequence of a given array of integers where the elements are in strictly increasing order, and the subsequence is as long as possible. A subsequence is derived by deleting zero or more elements from the original array without changing the order of the remaining elements.

For example, in the array $[10, 9, 2, 5, 3, 7, 101, 18]$, one possible LIS is $[2, 5, 7, 101]$ with length 4. The problem has applications in data analysis, sequence alignment, and more. We present an iterative dynamic programming solution to compute the LIS length efficiently.

Our problem can be stated as follows. Given an array $A = [a_1, a_2, \ldots, a_n]$ of n integers, find the length of the longest subsequence $[a_{i_1}, a_{i_2}, \ldots, a_{i_k}]$ such that:

$$i_1 < i_2 < \cdots < i_k \quad \text{and} \quad a_{i_1} < a_{i_2} < \cdots < a_{i_k}.$$

We use an iterative dynamic programming approach to solve the LIS problem. Define $dp[i]$ as the length of the longest increasing subsequence ending at index i (including a_i).

The idea is to build the dp array iteratively by comparing each element with all previous elements.

- Initialize $dp[i] = 1$ for all i (each element is an LIS of length 1 by itself).
- For each i, check all $j < i$. If $a[j] < a[i]$, update $dp[i]$ as $\max(dp[i], dp[j] + 1)$.
- The length of the LIS is the maximum value in the dp array.

Algorithm LongestIncreasingSubsequence: Longest Increasing Subsequence (Iterative)

Input: Array A of n integers
Output: Length of the LIS

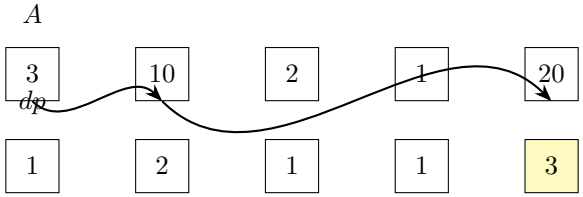
```
Initialize array dp[0...n-1] with all elements set to 1
for i = 1 to n-1 do
    for j = 0 to i-1 do
        if A[j] < A[i] then dp[i] = max(dp[i], dp[j] + 1)
    end
end
return max(dp[0], dp[1], ..., dp[n-1])
```

$dp[i]$ represents the longest increasing subsequence ending at $A[i]$, considering all prior elements. By checking all $j < i$, we ensure every possible subsequence ending at i is evaluated. The maximum $dp[i]$ gives the overall LIS length, as it captures the longest chain possible.

Consider the array $A = [3, 10, 2, 1, 20]$. We compute the LIS length step-by-step and visualize the process.

- $A[0] = 3$, $dp[0] = 1$ (base case).
- $A[1] = 10$, $3 < 10$, $dp[1] = \max(1, 1 + 1) = 2$.
- $A[2] = 2$, $3, 10 > 2$, $dp[2] = 1$.
- $A[3] = 1$, $3, 10, 2 > 1$, $dp[3] = 1$.
- $A[4] = 20$, $3 < 20$, $10 < 20$, $2 < 20$, $1 < 20$, $dp[4] = \max(1, 1 + 1, 2 + 1, 1 + 1, 1 + 1) = 3$.

Final $dp = [1, 2, 1, 1, 3]$, LIS length $= \max(dp) = 3$. One LIS is $[3, 10, 20]$.



The highlighted $dp[4] = 3$ indicates the LIS length. Arrows show one possible LIS: $[3, 10, 20]$.

The algorithm has two nested loops:
Outer loop: i from 1 to $n - 1$, $O(n)$ iterations.
Inner loop: j from 0 to $i - 1$, up to $O(n)$ iterations per i.

The total comparisons equal $0 + 1 + 2 + \cdots + (n - 1) = \frac{(n-1)n}{2}$, which is $O(n^2)$.

Thus, the overall time complexity is $O(n^2)$. Space complexity is $O(n)$ for the dp array.

10.5 Longest Increasing Subsequence Using LLP Method

We are given an integer array as input. For simplicity, we assume that all entries are distinct. Our goal is find for each index i the length of the longest increasing sequence that ends at i. For example, suppose the array A is `{35 38 27 45 32}`. Then, the desired output is `{1 2 1 3 2}`. The corresponding longest increasing subsequences are: `(35), (35, 38), (27), (35, 38, 45), (27, 32).`

We can define a graph H with indices as vertices. For this example, we have five vertices numbered v_1 to v_5. We draw an edge from v_i to v_j if i is less than j and $A[i]$ is also less than $A[j]$. This graph is clearly acyclic as an edge can only go from a lower index to a higher index. We use $pre(j)$ to be the set of indices which have an incoming edge to j. The length of the longest increasing subsequence ending at index j is identical to the length of the longest path ending at j.

To solve the problem using LLP, we model it as a search for the smallest vector G that satisfies the constraint $B \equiv \forall j : G[j] \geq 1 \wedge \forall j : G[j] \geq \max\{G[i]+1 \mid i \in pre(j)\}$.

The underlying lattice we consider is that of all vectors of natural numbers less than or equal to the maximum element in the lattice. A vector in this lattice is *feasible* if it satisfies B. We first show that the constraint B is lattice-linear.

Lemma 10.1 *The constraint* $B \equiv (\forall j : G[j] \geq 1) \wedge (\forall j : G[j] \geq \max\{G[i]+1 \mid i \in pre(j)\})$ *is lattice-linear.*

Proof: Since the predicate B is a conjunction of two predicates, it suffices to show that each of them is lattice-linear. The first conjunct is lattice linear because the constant function 1 is monotone. The second conjunct can be viewed as a conjunction over all j. For a fixed j, the predicate $G[j] \geq \max\{G[i]+1 \mid i \in pre(j)\}$ is lattice-linear.

■

Our goal is to find the smallest vector in the lattice that satisfies B. To ensure that no $G[j]$ is updated more than once, we introduce a boolean $fixed$ for each index such that we update $G[j]$ only when it is not fixed and all its predecessors are fixed. We now have the Algorithm LLP-Longest-Increasing-Subsequence.

Algorithm LLP-Longest-Increasing-Subsequence: Finding the Longest Increasing Subsequence.

```
1 P_j: Code for thread j
2 input: A:array of int;
3 var G: array[1 ... n] of int;
4       fixed: array[1 ... n] of boolean;
5 init: G[j] = 1; fixed[j] := false;
6       pre(j) := {i ∈ 1..j − 1|A[i] < A[j]};
7 forbidden: ¬fixed[j] ∧ (∀i ∈ pre(j) : fixed[i]);
8       advance: G[j] := max{G[i] + 1 | i ∈ pre(j)};
9                fixed[j] := true;
```

Let us now analyze the complexity of the algorithm. The sequential complexity is simple because we can maintain the list of all vertices that are forbidden because all its predecessors are fixed. Once we have processed a vertex, we never process it again. This is similar to a sequential algorithm of topological sort. In this case, we examine a vertex exactly once only after all its predecessors are fixed. The time complexity of this algorithm is $O(n^2)$.

We now add lattice-linear constraints to the problem. Instead of the longest increasing subsequence, we may be interested in the longest increasing subsequence that satisfies an additional predicate.

Lemma 10.2 *All the following predicates are lattice linear.*

1. *For any j, $G[j]$ is greater than or equal to the longest increasing subsequence of odd integers ending at j.*

2. *$G[j]$ is greater than or equal to the longest increasing subsequence such that j^{th} element in the subsequence exceeds $(j-1)^{th}$ element by at least k.*

Proof:

1. Since lattice-linear predicates are closed under conjunction, it is sufficient to focus on a fixed j. If $G[j]$ is less than the length of the longest increasing subsequence of odd integers ending at j, then the index j is forbidden. Unless j is increased the predicate can never become true.

2. We view this predicate as redrawing the directed graph H such that we draw an edge from v_i to v_j if i is less than j and $A[i]+k$ is less than or equal to $A[j]$.

■

10.6 Optimal Binary Search Tree

The Optimal Binary Search Tree (OBST) problem involves constructing a binary search tree (BST) for n symbols that minimizes the expected search cost, given the probability of each symbol's occurrence. Each symbol i (ranging from 0 to $n-1$) has a probability $p[i]$ of being searched, and these probabilities sum to 1, i.e., $\sum_{i=0}^{n-1} p[i] = 1$. The objective is to arrange the symbols in a BST to minimize the total cost, defined as the sum of each symbol's probability multiplied by its depth, where the root has depth 0. For instance, given three symbols A, B, and C with probabilities $p = [0.4, 0.3, 0.3]$, an optimal BST might place A as the root to reduce the average search time. This problem finds applications in areas like database indexing and compiler design, where efficient search structures are crucial. We present an iterative dynamic programming solution to compute the minimum cost.

Formally, given n symbols indexed from 0 to $n-1$, each associated with probability $p[i]$, and defining the sum of probabilities from i to j as $s(i,j) = \sum_{k=i}^{j} p[k]$, the task is to construct a BST that minimizes the expected search cost, expressed as $\text{Cost} = \sum_{i=0}^{n-1} p[i] \cdot \text{depth}(i)$, where $\text{depth}(i)$ represents the depth of symbol i in the tree.

To solve this, an iterative dynamic programming approach is employed. Two key arrays are defined: $dp[i][j]$ represents the minimum cost of a BST for symbols from i to j (inclusive), and $s(i,j)$ denotes the sum of probabilities from i to j, which serves as the cost increment at each level. The solution builds the dp table iteratively over increasing lengths of subarrays. When considering a single symbol ($j = i$), the cost is simply $dp[i][i] = p[i]$, reflecting a leaf node's contribution. For larger ranges, each symbol r from i to j is tested as the root, combining the left subtree $[i, r-1]$ and right subtree $[r+1, j]$. The total cost for a root r is computed as $dp[i][r-1] + dp[r+1][j] + s(i,j)$, where $s(i,j)$ accounts for the increased depth of all symbols in the subtree. To optimize, $s(i,j)$ is precomputed iteratively.

Here is the algorithm in pseudocode:

The correctness of this approach stems from the fact that $dp[i][j]$ computes the minimum cost by exhaustively trying each possible root r and combining the optimal costs of the resulting subproblems. The term $s(i,j)$ ensures that the cost reflects the depth increase for all symbols in the subtree. By filling the table bottom-up, from single symbols to the full range, the iterative method guarantees that $dp[0][n-1]$ yields the optimal cost.

Consider an example with $n = 4$ symbols A, B, C, and D, with probabilities $p = [0.25, 0.2, 0.3, 0.25]$. The computation proceeds as follows, and key steps are highlighted using itemization for clarity:

- For length $l = 0$, base cases are set: $dp[0][0] = 0.25$, $dp[1][1] = 0.2$, $dp[2][2] = 0.3$, $dp[3][3] = 0.25$, with corresponding $s[i][i] = p[i]$.

- For $l = 1$, consider $i = 0$, $j = 1$: $s[0][1] = 0.25 + 0.2 = 0.45$. Testing $r = 0$ gives $0.45 + 0 + 0.2 = 0.65$, $r = 1$ gives $0.45 + 0.25 + 0 = 0.7$, so $dp[0][1] = 0.65$ (root A). Similarly, $dp[1][2] = 0.65$ (root C), $dp[2][3] = 0.65$ (root C).

- For $l = 2$, consider $i = 0$, $j = 2$: $s[0][2] = 0.45 + 0.3 = 0.75$. Testing $r = 0$ gives $0.75 + 0 + 0.65 = 1.4$, $r = 1$ gives $0.75 + 0.25 + 0.3 = 1.3$, $r = 2$ gives $0.75 + 0.65 + 0 = 1.4$, so $dp[0][2] = 1.3$ (root B).

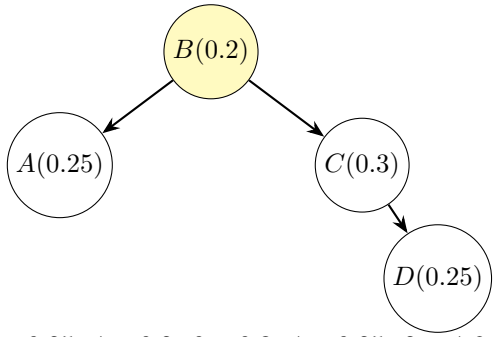
Algorithm Optimal-Binary-Search-Tree: Optimal Binary Search Tree (Iterative)

Input: Array $p[0 \ldots n-1]$ of probabilities
Output: Minimum cost of OBST

```
1  Initialize dp[0...n-1][0...n-1] and s[0...n-1][0...n-1] with 0
2  for i = 0 to n-1 do
3  |  dp[i][i] = p[i]
4  |  s[i][i] = p[i]
5  end
6  for l = 1 to n-1 do
7  |  for i = 0 to n-1-l do
8  |  |  j = i + l
9  |  |  s[i][j] = s[i][j-1] + p[j]
10 |  |  dp[i][j] = ∞
11 |  |  for r = i to j do
12 |  |  |  left_cost = 0
13 |  |  |  right_cost = 0
14 |  |  |  if r > i then left_cost = dp[i][r-1]
15 |  |  |  if r < j then right_cost = dp[r+1][j]
16 |  |  |  cost = s[i][j] + left_cost + right_cost
17 |  |  |  dp[i][j] = min(dp[i][j], cost)
18 |  |  end
19 |  end
20 end
21 return dp[0][n-1]
```

- For $l = 3$, $i = 0$, $j = 3$: $s[0][3] = 0.75 + 0.25 = 1.0$. Testing $r = 0$ gives $1.0 + 0 + 1.15 = 2.15$ (using $dp[1][3] = 1.15$), $r = 1$ gives $1.0 + 0.25 + 0.65 = 1.9$, $r = 2$ gives $1.0 + 0.65 + 0.25 = 1.9$, $r = 3$ gives $1.0 + 1.3 + 0 = 2.3$, so $dp[0][3] = 1.9$ (root B or C).

The minimum cost is $dp[0][3] = 1.9$, with B as a possible optimal root. The resulting optimal BST can be visualized as follows:



$$\text{Cost} = 0.25 \cdot 1 + 0.2 \cdot 0 + 0.3 \cdot 1 + 0.25 \cdot 2 = 1.9$$

In this tree, B is the root at depth 0, A and C are at depth 1, and D is at depth 2, resulting in a cost of $0.25 \cdot 1 + 0.2 \cdot 0 + 0.3 \cdot 1 + 0.25 \cdot 2 = 1.9$, matching $dp[0][3]$.

To analyze the time complexity, consider the three nested loops in the algorithm. The outer loop iterates over lengths l from 1 to $n - 1$, which takes $O(n)$ time. For each l, the middle loop iterates over starting indices i from 0 to $n - 1 - l$, also $O(n)$ per length. Within these, the inner loop tests each root r from i to j, which can be up to $O(n)$ iterations. This results in a total of $O(n) \cdot O(n) \cdot O(n) = O(n^3)$ operations. Precomputing $s[i][j]$ requires $O(n^2)$ time, as each entry builds on the previous, but this is dominated by the main computation. The overall time complexity is thus $O(n^3)$, with a space complexity of $O(n^2)$ for the dp and s tables.

In conclusion, this iterative dynamic programming solution constructs an OBST with the minimum expected search cost in $O(n^3)$ time.

10.7 Optimal Binary Search Tree with LLP

Suppose that we have a fixed set of n symbols called *keys* with some associated information called *values*. Our goal is to build a dictionary based on binary search tree out of these symbols. The dictionary supports a single operation search which returns the value associated with the the given key. We are also given the frequency of each symbol as the argument for the search query. The cost of any search for a given key is given by the length of the path from the root of the binary search tree to the node containing that key. Given any binary search tree, we can compute the total cost of the tree for all searches. We would like to build the binary search tree with the least cost.

Let the frequency of key i being searched is p_i. We assume that keys are sorted in the increasing order of symbols. Our algorithm is based on building progressively bigger binary search trees. The main idea is as follows. Suppose symbol k is the root of an optimal binary search tree for symbols in the range $[i..j]$. The root k divides the range into three parts – the range of indices strictly less than k, the index k, and the range of indices strictly greater than k. The left or the right range may be empty. Then, the left subtree and the right subtree must themselves be optimal for their respective ranges. Let $G[i, j]$ denote the least cost of any binary search tree built from symbols in the range $i..j$. We use the symbol $s(i, j)$ as the sum of all frequencies from the symbol i to j, i.e.,

$$s(i, j) = \sum_{k=i}^{j} p_k$$

For convenience, we let $s(i, j)$ equal to 0 whenever $i > j$, i.e., the range is empty.

We now define a lattice linear constraint on $G[i,j]$. Let $i \leq k \leq j$. Consider the cost of the optimal tree such that symbol k is at the root. The cost has three components: the cost of the left subtree if any, the cost of the search ending at this node itself and the cost of search in the right subtree. The cost of the left subtree is

$$G[i,k-1] + s(i,k-1)$$

whenever $i < k$. The cost of the node itself is $s(k,k)$. The cost of the right subtree is

$$G[k+1,j] + s(k+1,j)$$

Combining these expressions, we get

$$G[i,j] = \min_{i \leq k \leq j} (G[i,k-1] + s(i,j) + G[k+1,j])$$

This is also the least value of $G[i,j]$ such that

$$G[i,j] \geq \min_{i \leq k \leq j} (G[i,k-1] + s(i,j) + G[k+1,j])$$

We now show that the above predicate is lattice-linear.

Lemma 10.3 *The constraint* $B \equiv \forall i,j : G[i,j] \geq \min_{i \leq k \leq j}(G[i,k-1] + s(i,j) + G[k+1,j])$ *is lattice-linear.*

Proof: Suppose that B is false, i.e., $\exists i,j : G[i,j] < \min_{i \leq k \leq j}(G[i,k-1] + s(i,j) + G[k+1,j])$. This means that there exists i,j,k with $i \leq k \leq j$ such that $G[i,j] < (G[i,k-1] + s(i,j) + G[k+1,j])$. This means the the index (i,j) is forbidden and unless $G[i,j]$ is increased, the predicate B can never become true irrespective of how other components of G are increased.

■

We now have our LLP-based algorithm for Optimal Binary Search Tree as Algorithm LLP-OptimalBinarySearchTree. The program has a single variable G. It is initialized so that $G[i,i]$ equals $p[i]$ and $G[i,j]$ equals zero whenever i is not equal to j. The algorithm advances $G[i,j]$ whenever it is smaller than $\min_{i \leq k \leq j} G[i,k-1] + s(i,j) + G[k+1,j]$. In Algorithm LLP-OptimalBinarySearchTree, we have used the **always** clause as a macro that uses $s(i,j)$ as a short form for $\sum_{k=i}^{j} p[k]$.

Algorithm LLP-OptimalBinarySearchTree: Finding An Optimal Binary Search Tree

```
1 P_{i,j}: Code for thread (i,j)
2 input: p:array of real;// frequency of each symbol
3 init: G[i,j] = 0 ∀i ≠ j;
4       G[i,i] = p[i];
5 always: s(i,j) = Σ_{k=i}^{j} p[k]
6 ensure:
7       G[i,j] ≥ min_{i≤k≤j} G[i,k-1] + s(i,j) + G[k+1,j]
8 priority: (j - i)
```

Although, the above algorithm will give us correct answers, it is not efficient as it may update $G[i,j]$ before $G[i,k]$ and $G[k,j]$ for $i \leq k \leq j$ have stabilized. However, the following scheduling strategy ensures that we update $G[i,j]$ at most once. We check for whether $G[i,j]$ is forbidden in the order of $j-i$. Hence, initially all $G[i,j]$ such that $j = i+1$ are updated. This is followed by all $G[i,j]$ such that $j = i+2$, and so on. We capture this scheduling strategy with the **priority** statement. We pick $G[i,j]$ to update such that $(j-i)$ have minimal values. Of course, our goal is to compute $G[1,n]$. With the above strategy of updating $G[i,j]$, we get that $G[i,j]$ is updated at most once. Since there are $O(n^2)$ possible values of $G[i,j]$ and each takes $O(n)$ work to update, we get the work complexity of $O(n^3)$.

We now consider the constrained versions of the problem.

Lemma 10.4 *All the following predicates are lattice linear.*

1. *Key x is not a parent for any key.*
2. *The difference in the sizes of the left subtree and the right subtree is at most* 1.

Proof:

1. This requirement changes the ensure predicate to $G[i,j] \geq \min_{i \leq k \leq j, k \neq x} G[i,k-1] + s(i,j) + G[k+1,j]$. The right hand side of the constraint continues to be monotonic and therefore it is lattice linear.
2. This requirement changes the ensure predicate to $G[i,j] \geq \min_{i \leq k \leq j, |k-1-i, j-k-1| \leq 1} G[i,k-1] + s(i,j) + G[k+1,j]$. This change simply restricts the values of k, and the right hand side continues to be monotonic.

■

10.8 Chain Matrix Multiplication

A problem very similar to Optimal Binary Search tree is that of constructing an optimal way of multiplying a chain of matrices. Since matrix multiplication is associative, the product of matrices $(M_1 * M_2) * M_3$ is equal to $M_1 * (M_2 * M_3)$. However, depending upon the dimensions of the matrices, the computational effort may be different. For example, consider the dimensions (30 times 10), (10 times 30), (30 times 2) results in a matrix of size 30 times 2. If we multiply the first two matrices using the standard simple matrix multiplication, we get 30 times 10 times $30 = 9000$ operations to get 30 times 30 matrix. Multiplying M_3 requires 1800 additional operations. If we multiply M_2 and M_3 first, then we require 600 operations first. Multiplying M_1 adds $30 \times 10 \times 2 = 600$ additional operations. We let the dimension of matrix M_i be $m_{i-1} \times m_i$. Note that this keep the matrix product well-defined because the dimension of matrix M_{i+1} would be $m_i \times m_{i+1}$ and the product $M_i \times M_{i+1}$ is well-defined. We can view any evaluation of a chain as a binary tree where the intermediate notes are the multiplication operation and the leaves are the matrices themselves. Suppose, our goal is to compute the optimal binary tree for multiplying matrices in the range $M_i \ldots M_j$. Borrowing ideas from the previous section, we let $G[i,j]$ denote the optimal cost of computing the product of matrices in the range $M_i \ldots M_j$. Suppose that this product is broken into products of $M_i \ldots M_k$ and $M_{k+1} \ldots M_j$ and then multiplication of these two matrices. We can compute the cost of this tree as

$$G[i,k] + G[k+1,j] + m_{i-1} m_k m_j$$

Then, we have the following predicate on G.

$$G[i,j] \geq \min_{i \leq k < j} (G[i,k] + m_{i-1} m_k m_j + G[k+1,j]$$

The reader will notice the similarity with the optimal binary search tree problem and this problem and the same algorithm can be adapted to solve this problem.

10.9 Segmented Least Squares Problem

The Segmented Least Squares problem involves fitting a set of n points in the x-y plane with a sequence of straight line segments. The goal is to minimize the total sum of squared errors between the points and their corresponding line segments, plus a penalty C for each segment used. This balances fitting accuracy with model simplicity. We present a dynamic programming solution, followed by an example with illustrations.

Given n points $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)$, sorted by increasing x-coordinates ($x_1 < x_2 < \cdots < x_n$), we aim to:

- Partition the points into segments.
- For each segment, compute the least squares line that minimizes the sum of squared errors.

- Minimize the total cost: sum of squared errors across all segments plus C times the number of segments.

For points $p_i, p_{i+1}, \ldots, p_j$, the least squares line minimizes:

$$e_{ij} = \sum_{k=i}^{j} (y_k - (ax_k + b))^2,$$

where a and b are the slope and intercept of the line, computed as:

$$a = \frac{n\sum(x_k y_k) - \sum x_k \sum y_k}{n\sum x_k^2 - (\sum x_k)^2}, \quad b = \frac{\sum y_k - a\sum x_k}{n},$$

for $k = i$ to j, and $n = j - i + 1$.

The total cost for a segmentation is:

$$\text{Cost} = \sum_{\text{segments}} e_{ij} + C \cdot (\text{number of segments}).$$

Define $OPT(j)$ as the minimum cost of segmenting points $p_1, p_2, \ldots, p_j$. The recurrence relation is:

$$OPT(j) = \min_{0 \le i < j} \left[OPT(i) + e_{i+1,j} + C \right],$$

where:

- $OPT(0) = 0$ (base case: no points),
- $e_{i+1,j}$ is the error of fitting a single line to points p_{i+1} to p_j,
- C is the penalty for adding a new segment starting at p_{i+1}.

To compute $OPT(j)$ for all $j = 1$ to n:

1. Precompute e_{ij} for all i, j pairs ($O(n^3)$ time with $O(n^2)$ space).
2. For each j, evaluate the minimum over all $i < j$ ($O(n)$ per j).
3. Store the optimal i for each j to reconstruct the segments.

Total time complexity: $O(n^3)$, space complexity: $O(n^2)$.

Algorithm Segmented-Least-Squares: Segmented Least Squares

Input: Points $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, penalty C
Output: Optimal cost $OPT(n)$, segment boundaries

```
// Precompute error e_ij for all segments;
for i = 1 to n do
    for j = i to n do
        Compute e_ij using least squares fit for p_i to p_j;
    end
end
// Initialize DP array;
OPT[0] = 0;
for j = 1 to n do
    OPT[j] = ∞;
    for i = 0 to j − 1 do
        cost = OPT[i] + e_{i+1,j} + C;
        if cost < OPT[j] then
            OPT[j] = cost;
            last[j] = i // Store the last break point;
        end
    end
end
// Reconstruct segments;
segments = [];
j = n;
while j > 0 do
    i = last[j];
    Add segment from p_{i+1} to p_j to segments;
    j = i;
end
return OPT[n], segments
```

Consider $n = 6$ points: $(1, 1), (2, 2), (3, 4), (4, 3), (5, 2), (6, 1)$, with penalty $C = 1$.
Compute e_{ij} for key segments (simplified for brevity):

- $e_{1,2}$: Line for $(1, 1), (2, 2)$, $y = x$, error $= 0$.
- $e_{1,3}$: Line for $(1, 1), (2, 2), (3, 4)$, $y = 1.5x - 0.5$, error $= 0.5$.
- $e_{3,4}$: Line for $(3, 4), (4, 3)$, $y = -x + 7$, error $= 0$.
- $e_{4,6}$: Line for $(4, 3), (5, 2), (6, 1)$, $y = -x + 7$, error $= 0$.
- $e_{1,6}$: Line for all points, $y = 2.2 - 0.2x$, error $= 7.6$.
- $OPT(0) = 0$,
- $OPT(1) = e_{1,1} + C = 0 + 1 = 1$,
- $OPT(2) = \min(OPT(0) + e_{1,2} + C, OPT(1) + e_{2,2} + C) = \min(0 + 0 + 1, 1 + 0 + 1) = 1$ (segment $1 - 2$),
- $OPT(3) = \min(OPT(0) + e_{1,3} + C, OPT(2) + e_{3,3} + C) = \min(0 + 0.5 + 1, 1 + 0 + 1) = 1.5$ (segment $1 - 3$),
- $OPT(4) = \min(OPT(0)+e_{1,4}+C, OPT(2)+e_{3,4}+C, OPT(3)+e_{4,4}+C) = \min(2.67+1, 1+0+1, 1.5+0+1) = 2$ (segments $1 - 2, 3 - 4$),
- $OPT(5) = \min(\ldots) = 3$ (segments $1 - 2, 3 - 4, 5 - 5$),

- $OPT(6) = \min(OPT(0)+e_{1,6}+C, OPT(2)+e_{3,6}+C, OPT(3)+e_{4,6}+C) = \min(7.6+1, 1+1+1, 1.5+0+1) = 2.5$ (segments $1-3, 4-6$).

Optimal solution: Segments $1-3$ ($y = 1.5x - 0.5$) and $4-6$ ($y = -x + 7$), cost $= 0.5 + 0 + 2C = 2.5$.

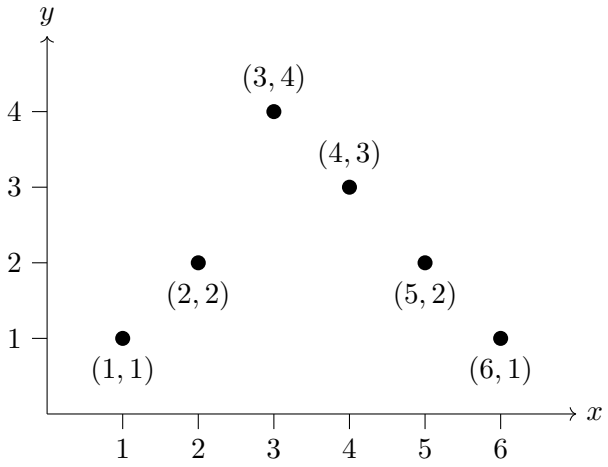


Figure 10.2: Input points $(1,1), (2,2), (3,4), (4,3), (5,2), (6,1)$ in the x-y plane.

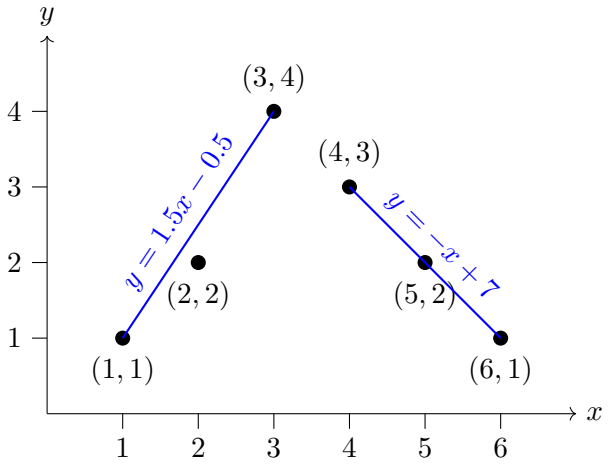


Figure 10.3: Segmented least squares solution: segments $1-3$ ($y = 1.5x - 0.5$) and $4-6$ ($y = -x + 7$).

10.10 Knapsack Problem

We are given n items with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$. We are also given a knapsack that has a capacity of W. Our goal is to determine the subset of items that can be carried in the knapsack and that maximizes the total value. The standard dynamic programming solution is based on memoization of the following dynamic programming formulation [Vaz01, DPW10]. Let $G[i, w]$ be the maximum value that can be obtained by picking items from $1..i$ with the capacity constraint of w. Then, $G[i, w] = max(G[i-1, w-w_i] + v_i, G[i-1, w])$. The first argument of the max function corresponds to the case when the item i is included in the optimal set from $1..i$, and

the second argument corresponds to the case when the item i is not included and hence the entire capacity can be used for the items from $1..i-1$. If $w_i > w$, then the item i can never be in the knapsack and can be skipped. The base cases are simple. The value of $G[0, w]$ and $G[i, 0]$ is zero for all w and i. Our goal is to find $G[n, W]$. By filling up the two dimensional array G for all values of $0 \leq i \leq n$ and $0 \leq w \leq W$, we get an algorithm with time complexity $O(nW)$.

The following algorithm computes $G[n, W]$ and tracks selected items:

Algorithm Knapsack Algorithm: 0/1 Knapsack Dynamic Programming

Input: Weights $w[1..n]$, Values $v[1..n]$, Capacity W
Output: Maximum value and selected items

```
for w = 0 to W do  G[0, w] ← 0 ;
for i = 0 to n do  G[i, 0] ← 0 ;
for i = 1 to n do
    for w = 1 to W do
        if w_i > w then  G[i, w] ← G[i − 1, w] ;
        else  G[i, w] ← max(G[i − 1, w], G[i − 1, w − w_i] + v_i) ;
    end
end
selected ← ∅;
i ← n;
w ← W;
while (i > 0) ∧ (w > 0) do
    if G[i, w] ≠ G[i − 1, w] then
        selected ← selected ∪ {i};
        w ← w − w_i;
    end
    i ← i − 1;
end
return G[n, W], selected;
```

Consider $n = 4$ items with weights $w = [2, 3, 4, 5]$, values $v = [3, 4, 5, 6]$, and capacity $W = 8$. The dynamic programming table $G[i, w]$ is computed as follows (key steps shown):

- $G[0, w] = 0$ for $w = 0$ to 8.
- $i = 1, w_1 = 2, v_1 = 3$:
  - $w = 2$: $G[1, 2] = \max(0, 3) = 3$
  - $w = 8$: $G[1, 8] = 3$
- $i = 2, w_2 = 3, v_2 = 4$:
  - $w = 5$: $G[2, 5] = \max(3, 7) = 7$
  - $w = 8$: $G[2, 8] = 7$
- $i = 3, w_3 = 4, v_3 = 5$:
  - $w = 8$: $G[3, 8] = \max(7, 9) = 9$
- $i = 4, w_4 = 5, v_4 = 6$:
  - $w = 8$: $G[4, 8] = \max(9, 9) = 9$

Final value: $G[4, 8] = 9$. Backtracking: - $G[4, 8] = G[3, 8]$, item 4 not included. - $G[3, 8] > G[2, 8]$, item 3 included ($w = 8 - 4 = 4$). - $G[2, 4] > G[1, 4]$, item 2 included ($w = 4 - 3 = 1$). - $G[1, 1] = 0$, item 1 not included.

Selected items: $\{2, 3\}$, weight $= 3 + 4 = 7 \leq 8$, value $= 4 + 5 = 9$.

10.11 Knapsack Using LLP Formulation

We can model this problem using lattice-linear predicates as follows. We model the feasibility as $G[i,w] \geq \max(G[i-1, w-w_i]+v_i, G[i-1,w])$ for all $i, w > 0$ and $w_i \leq w$. Also, $G[i,w] = 0$ if $i = 0$ or $w = 0$. Our goal is to find the minimum vector G that satisfies feasibility.

Lemma 10.5 *The constraint $B \equiv \forall i, w : G[i,w] \geq \max(G[i-1, w-w_i]+v_i, G[i-1,w])$ for $w_i \leq w$ is lattice-linear.*

Proof: If the predicate B is false, there exists i and w such that $G[i,w] < \max(G[i-1, w-w_i]+v_i, G[i-1,w])$. The value $G[i,w]$ is forbidden; unless $G[i,w]$ is increased the predicate can never become true.

■

Algorithm LLP-Knapsack: Finding An Optimal Solution to the Knapsack Problem

```
1 P_{i,j}: Code for thread (i, j)
2 input: w, v:array[1..n] of int;// weight and value of each item
3 var: G:array[0...n, 0...W] of int;
4 init: G[i,j] = 0 if (i = 0) ∨ (j = 0);
5 ensure:
6 G[i,j] ≥ max{G[i-1, j-w_i] + v_i, G[i-1, j]}   if  j ≥ w_i
7            ≥ G[i-1, j], otherwise.
```

Algorithm LLP-Knapsack updates the value of $G[i,j]$ based only on the values of $G[i-1,.]$. Furthermore, $G[i,j]$ is always at least $G[i-1,j]$. Based on this observation, we can simplify the algorithm as follows. We consider the problem of adding just one item to the knapsack given the constraint that the total weight does not exceed W. We maintain the list of all optimal configurations for each weight less than W.

Algorithm LLP-IncrKnapsack2: Finding An Optimal Solution to the Incremental Knapsack Problem

```
1 P_j: Code for thread j
2 input: w, v: int;// weight and value of the next item
3       C: array[0...W] of int;
4 var: G:array[0...W] of int;
5 init: ∀j : G[j] = C[j];
6 ensure:
7       G[j] ≥ C[j-w] + v   if  j ≥ w
```

The incremental algorithm can be implemented in $O(1)$ parallel time using $O(W)$ processors as shown in Fig. LLP-IncrKnapsack2. Each processor j can check whether $G[j]$ needs to be advanced.

We can now invoke the incremental Knapsack algorithm as Algorithm Knapsack2.

We now add some lattice-linear constraints to the Knapsack problem. In many applications, some items may be related and the constraint $x_a \Rightarrow x_b$ means that if the item x_a is included in the Knapsack then the item x_b must also be included. Thus, the item x_a has profit of zero if x_b is not included. The item x_b has utility even without x_a but not vice-versa. Without loss of generality, we assume that all weights are strictly positive, and that index $b < a$. In the following Lemma, we use an auxiliary variable $S[i,j]$ that keeps the set of items included in $G[i,j]$ and not just the profit from those items.

Algorithm Knapsack2: Finding An Optimal Solution to the Knapsack Problem

```
1 P_j: Code for thread j
2 input: w, v:array[1..n] of int;// weight and value of each item
3 var: G:array[0...W] of int;
4 init: ∀j : G[j] = 0;
5 for i := 1 to n do
6       G := IncrKnapsack2(w[i], v[i], G);
```

Lemma 10.6 *First assume that* $(i \neq a)$. *Let* $B(i,w) \equiv G[i,w] \geq \max(G[i-1,w-w_i]+v_i, G[i-1,w])$ *for* $(w_a \leq w)$ *and* $G[i,w] \geq G[i-1,w]$, *otherwise. This predicate corresponds to any item* i *different from* a. *The value with a bag of capacity* w *is always greater than or equal to the choice of picking the item or not picking the item.*

Let $B(a,w) \equiv G[a,w] \geq \max(G[a-1,w-w_a]+v_a, G[a-1,w])$ *if* $b \in S[a-1,w-w_a] \wedge (w_a \leq w)$ *and* $G[a,w] \geq G[a-1,w]$, *otherwise.*

Then, $B(i,w)$ *is lattice-linear for all* i *and* w.

Proof: Suppose that $B(i,w)$ is false for some i and w. Unless $G[i,w]$ is increased, it can never become true.

■

10.12 Problems

1. We are given a directed acyclic graph (V,E) with n nodes and m edges. Our goal is to assign a number, *label* to each vertex from $1..n$ such that for all edges $(i,j) \in E$, $label[i] < label[j]$. Define B as $\forall (i,j) \in E : label[j] \geq label[i]+1$. Show that B is lattice-linear predicate.

2. Modify the algorithm for Optimal Binary Search Tree to return not only the optimal cost but also the tree itself.

3. **(Longest Path in Directed Acyclic Graphs)** We are given a directed acyclic graph (V,E) with n nodes and m edges such that each edge has a positive real cost. We are also given a distinguished vertex v_0. Give an LLP based algorithm to find the longest path from v_0 to all vertices.

4. We are given a polygon and are required to triangulate it optimally. We are given a weight function that takes a triangle on vertices v_i, v_j and v_k and returns a real-valued function $w(v_i, v_j, v_k)$. Our goal is to triangulate the given polynomial and triangulate it to minimize the sum of weights of all triangles. *(Hint: Devise a recurrence relation similar to the matrix chain product problem.)*

10.13 Bibliographic Remarks

This chapter 14 explores the paradigm of solving optimization problems by breaking them into overlapping subproblems, with applications to the Longest Increasing Subsequence (LIS), Optimal Binary Search Tree (OBST), Knapsack, Weighted Interval Scheduling, Segmented Least Squares, and Chain Matrix Multiplication. The chapter introduces dynamic programming (DP) as a method to establish recurrence relations, avoiding redundant computations through memoization or bottom-up computation, as originally formalized by Bellman [Bel52]. Cormen et al. [CLRS01] provide a comprehensive treatment of sequential DP algorithms for these problems.

The chapter contrasts traditional DP with the Lattice-Linear Predicate (LLP) algorithm, which models problems as searches for extremal vectors in a finite distributive lattice. This approach, detailed in Garg [Gar22], enables parallel computation with minimal synchronization overhead and supports lattice-linear constraints.

The OBST problem, introduced by Knuth [Knu71], minimizes expected search cost in a binary search tree, with the LLP approach achieving $O(n \log n)$ parallel time. The Knapsack problem, formulated in Horowitz and Sahni [HS74] and Ibarra and Kim [IK75], is solved in $O(nW)$ time via DP, with LLP reducing parallel time to $O(n)$ using W processors. Vazirani [Vaz01] and Williamson [DPW10] provide modern DP formulations for Knapsack, including approximation techniques.

Weighted Interval Scheduling and Segmented Least Squares, covered in Kleinberg and Tardos [KT06], extend greedy and DP techniques to optimization problems with weighted objectives and error minimization, respectively. The Chain Matrix Multiplication problem, structurally similar to OBST, is addressed using a recurrence akin to Knuth's formulation, as noted in Aho, Hopcroft, and Ullman [AHU74]. Papadimitriou and Steiglitz [PS82] offer additional context for combinatorial optimization problems like Knapsack and matrix multiplication.

Chapter 11

Max Flow

11.1 Introduction

The Ford-Fulkerson algorithm is a greedy algorithm that computes the maximum flow in a flow network. It was published in 1956 by L. R. Ford, Jr. and D. R. Fulkerson.

The maximum flow problem is one of the most fundamental problems in network theory with diverse applications, ranging from traffic systems and communication networks to supply chains and project scheduling. Understanding how to efficiently compute the maximum flow allows for optimizing resources and improving system performance.

This chapter provides a comprehensive overview of the Ford-Fulkerson algorithm, including its theoretical foundations, correctness proof, and complexity analysis. Additionally, we explore related concepts such as the Max-Flow Min-Cut Theorem and discuss strongly polynomial algorithms in the context of network flows.

Definition 11.1 (Flow Network) *A flow network is a directed graph $G = (V, E)$ with a source vertex $s \in V$, a sink vertex $t \in V$, and a capacity function $c : E \to \mathbb{R}^+$ that assigns a non-negative capacity $c(u, v)$ to each edge $(u, v) \in E$.*

Definition 11.2 (Flow) *A flow in a flow network G is a function $f : E \to \mathbb{R}^+$ that satisfies:*

1. *Capacity constraint: $\forall (u, v) \in E, 0 \leq f(u, v) \leq c(u, v)$*
2. *Flow conservation: $\forall v \in V \setminus \{s, t\}, \sum_{u:(u,v)\in E} f(u, v) = \sum_{w:(v,w)\in E} f(v, w)$*

Definition 11.3 (Value of Flow) *The value of a flow f is defined as the total flow out of the source:*

$$|f| = \sum_{v:(s,v)\in E} f(s, v)$$

Definition 11.4 (Residual Network) *Given a flow network $G = (V, E)$ and a flow f, the residual network $G_f = (V, E_f)$ consists of edges with remaining capacity, defined as:*

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

where $c_f(u, v)$ is the residual capacity defined as:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$$

Definition 11.5 (Augmenting Path) *An augmenting path is a simple path from source s to sink t in the residual network G_f.*

11.2 The Ford-Fulkerson Algorithm

The algorithm works by finding augmenting paths in the residual network and increasing the flow along these paths until no more augmenting paths can be found.

Algorithm 2: Ford-Fulkerson Algorithm

Input: Graph G, source s, sink t
Output: Maximum flow f

Initialize flow $f(u, v) = 0$ for all edges (u, v) in G
while *there exists an augmenting path p from s to t in the residual network G_f* **do**
  Let $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$ be the residual capacity of path p
  for *each edge (u, v) in p* **do**
    if $(u, v) \in E$ **then**
      $f(u, v) = f(u, v) + c_f(p)$ `// Increase flow`
    else
      $f(v, u) = f(v, u) - c_f(p)$ `// Decrease flow in reverse edge`
    end
  end
end
return f

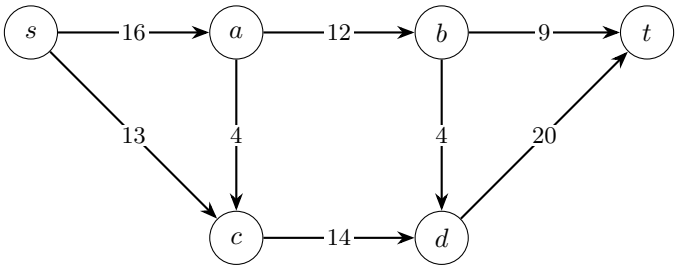
Theorem 11.6 *The Ford-Fulkerson algorithm correctly computes the maximum flow in any flow network with finite capacities.*

Proof: The algorithm repeatedly augments the flow along augmenting paths found in the residual graph. Each augmentation strictly increases the total flow value by at least the minimum residual capacity of the augmenting path. Since the capacities are finite and non-negative, the total flow is bounded above.

The algorithm terminates when there are no more augmenting paths in the residual graph. By the Max-Flow Min-Cut Theorem, a flow is maximum if and only if there is no augmenting path in the residual graph. Therefore, when the algorithm terminates, the current flow is maximal.

■

Let's consider a simple flow network and demonstrate the Ford-Fulkerson algorithm step by step.
Consider a flow network with 6 vertices {s, a, b, c, d, t} and the following capacities:

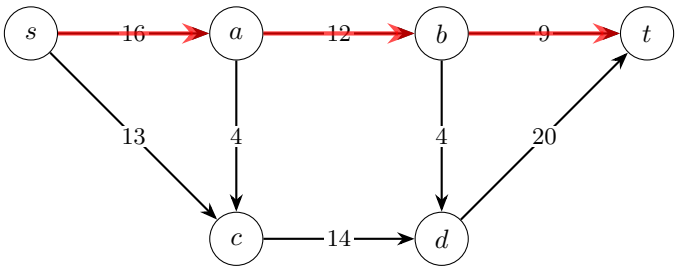


Initially, all flow values are 0. The edge labels show capacity.

First Iteration

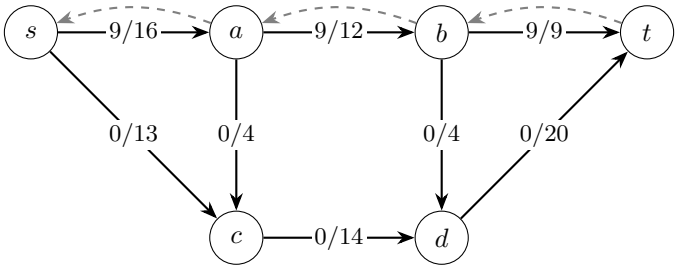
Finding the First Augmenting Path

We look for a path from s to t in the residual network. Let's choose the path $s \to a \to b \to t$.



The minimum residual capacity on this path is $\min\{16, 12, 9\} = 9$. We augment the flow by 9 units along this path.

Updated Flow After First Iteration

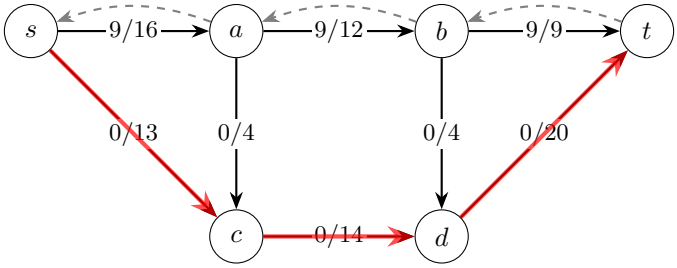


The flow value is now 9. The edge labels show flow/capacity. The residual network now includes reverse edges where flow was pushed.

Second Iteration

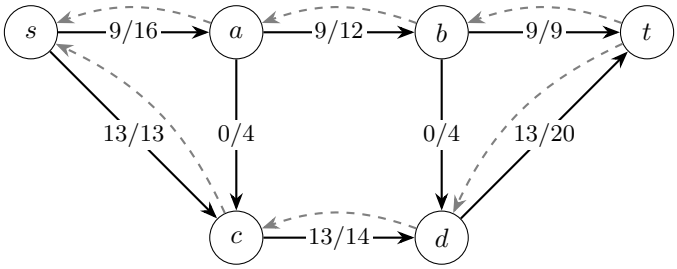
Finding the Second Augmenting Path

We find another path in the residual network: $s \to c \to d \to t$.



The minimum residual capacity on this path is $\min\{13, 14, 20\} = 13$. We augment the flow by 13 units.

Updated Flow After Second Iteration

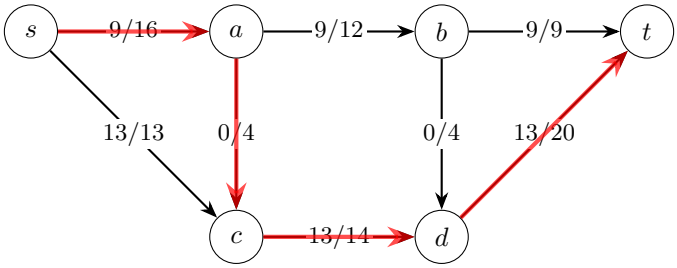


The flow value is now $9 + 13 = 22$.

Third Iteration

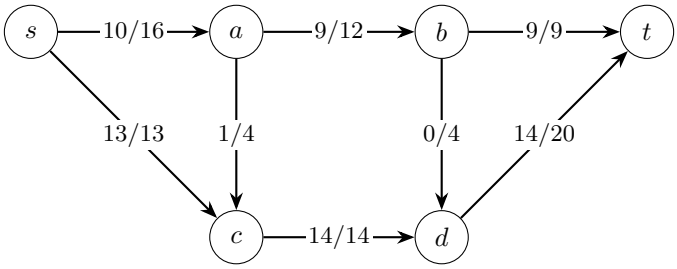
Finding the Third Augmenting Path

We find another path: $s \to a \to c \to d \to t$.



The minimum residual capacity on this path is $\min\{7, 4, 1, 7\} = 1$. We augment the flow by 1 unit.

Updated Flow After Third Iteration

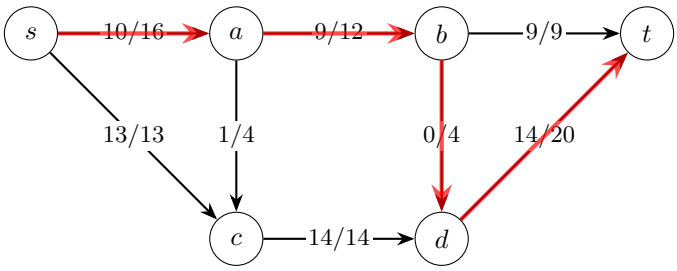


The flow value is now $10 + 13 = 23$.

Fourth Iteration

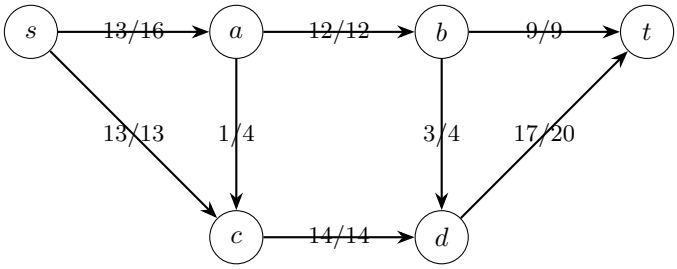
Finding the Fourth Augmenting Path

We find one more path: $s \to a \to b \to d \to t$.



The minimum residual capacity on this path is $\min\{6, 3, 4, 6\} = 3$. We augment the flow by 3 units.

Final Flow



The flow value is now 13 + 13 = 26. There are no more augmenting paths in the residual network, so the algorithm terminates. The maximum flow is 26.

11.3 Min-Cut Interpretation

Theorem 11.7 *In any flow network, the maximum value of a flow is equal to the minimum capacity of an s-t cut.*

Proof: Let f be a flow with maximum value, and let (S, T) be any cut of the network, where $s \in S$ and $t \in T$. The value of the flow $|f|$ cannot exceed the capacity of the cut $c(S, T)$, since no more than $c(S, T)$ units of flow can cross from S to T.

When the Ford-Fulkerson algorithm terminates, there are no augmenting paths in the residual graph. Define S as the set of vertices reachable from s in the residual graph, and $T = V \setminus S$. Since no augmenting paths remain, all edges from S to T in the original graph are saturated, and all edges from T to S carry no flow.

Therefore, the value of the flow equals the capacity of the cut (S, T), proving that the maximum flow equals the minimum cut capacity.

■

According to the Max-Flow Min-Cut theorem, the value of the maximum flow equals the capacity of the minimum cut. In our example, we can identify the minimum cut by finding the set of vertices reachable from s in the final residual network.

Let's identify the cut. The set of vertices reachable from s in the final residual network are $\{s, a, c\}$. The remaining vertices $\{b, d, t\}$ form the other side of the cut.

The cut edges are:

- (a, b) with capacity 12
- (c, d) with capacity 14

The total capacity is $12 + 14 = 26$, which is exactly equal to the maximum flow, confirming the Max-Flow Min-Cut theorem.

Let us determine the running time of Ford-Fulkerson algorithm. Suppose that all capacities in the graph are integral. Let f^* be the maximum flow in the graph. Every iteration of Ford-Fulkerson's algorithm finds an augmenting path in the residual network of at least one unit. This step takes $O(|E|)$ time. Hence, the total running time is $O(f^*|E|)$. This algorithm is pseudo-polynomial because it depends on the unary representation of the capacity C of edges. It is desirable to get an algorithm with the time complexity that is dependent only on the number of vertices and the edges of the graph and not the capacity. For large capacities, this can lead to inefficiencies, motivating strongly polynomial algorithms like Edmond-Karp and Dinitz's algorithm.

11.4 Edmond-Karp Algorithm

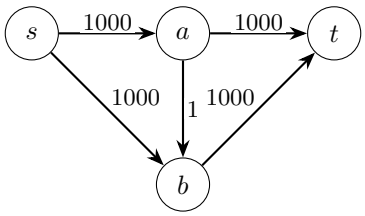
The Edmond-Karp modification improves the Ford-Fulkerson algorithm by selecting augmenting paths with the fewest possible edges, using a breadth-first search (BFS) to find shortest paths in the residual graph. This ensures that each augmentation is performed efficiently.

This approach guarantees a strongly polynomial runtime of $O(|E|^2|V|)$, independent of the numerical values of capacities. The algorithm works in phases, where each phase builds a level graph and finds blocking flows, reducing the number of necessary augmentations. Overall, Edmonds-Karp's algorithm does at most $(|E||V|)$ augmentations and each augmentation takes at $O(|E|)$ time (via BFS) giving us the time complexity of $O(|E|^2|V|$.

By focusing on shortest paths and limiting the number of augmentations, the Edmond-Karp algorithm provides significant performance improvements, especially for large-scale networks.

Consider a network with vertices s, a, b, t and edges: $s \to a$ with capacity 1000, $s \to b$ with capacity 1000, $a \to b$ with capacity 1, $a \to t$ with capacity 1000, $b \to t$ with capacity 1000.

In the basic Ford-Fulkerson algorithm, choosing poor augmenting paths may lead to many iterations due to the small capacity edge $a \to b$. In the example with vertices s, a, b, t, Ford-Fulkerson may choose paths like $s \to a \to b \to t$ repeatedly. Due to the bottleneck capacity of 1 on the edge $a \to b$, only 1 unit of flow is added per iteration. This leads to many redundant iterations before utilizing the larger capacity paths effectively. However, with Edmonds-Karp's method using BFS, the shortest paths are found, and the blocking flow quickly saturates the large capacity edges in just a few phases, significantly reducing the total number of iterations.



Algorithm 3: Edmonds-Karp Algorithm

Input: Graph $G = (V, E)$, capacities c, source s, sink t
Output: Maximum flow f

Initialize $f(u, v) \leftarrow 0$ for all $(u, v) \in E$;
while *there exists an augmenting path in* G_f **do**
  Use BFS to find shortest path p from s to t in G_f;
  Compute $c_f(p) = \min_{(u,v)\in p} c_f(u, v)$;
  for *each edge* $(u, v) \in p$ **do**
    $f(u, v) \leftarrow f(u, v) + c_f(p)$;
    $f(v, u) \leftarrow f(v, u) - c_f(p)$;
  end
end
return f;

11.5 Lattice of Mincuts

In this section, we first show that the property of a cut (S, T) being a mincut is lattice-linear. A set $S \subseteq V$ is a mincut if:

- $s \in S$, $t \notin S$ (i.e., S is a cut).
- $\text{val}(S) = \sum_{(u,v)\in E: u\in S, v\in V\setminus S} c(u, v) \leq \text{val}(S')$ for all cuts S'.

We can now show:

Lemma 11.8 *The predicate $B(S) = (s \in S$ and $t \notin S)$ and $(val(S)$ is minimum$)$ is lattice-linear with efficient advancement.*

Proof: We use submodularity of the cut function. For min cuts S_1, S_2, the submodularity inequality is:

$$\text{val}(S_1) + \text{val}(S_2) \geq \text{val}(S_1 \cup S_2) + \text{val}(S_1 \cap S_2).$$

If $\text{val}(S_1) = \text{val}(S_2) = \mu$ (the min cut capacity), then $\text{val}(S_1 \cup S_2) + \text{val}(S_1 \cap S_2) \leq 2\mu$. Since $\text{val}(S) \geq \mu$ for any cut, each term must equal μ. Thus, $S_1 \cap S_2$ and $S_1 \cup S_2$ are min cuts (noting $s \in S_1 \cap S_2$, $t \notin S_1 \cup S_2$).

For the advancement property, suppose S is not a min cut (e.g., $\text{val}(S) > \mu$). If S equals $V - \{t\}$, then we know that there is no $S' \supseteq S$ such that S' is a mincut and thus all indices are forbidden. Otherwise, there is at least some vertex v such that there exists an edge (u, v), $u \in S$, $v \notin S$, with residual capacity $c(u, v) > f(u, v)$. If v equals t, we know that there is no $S' \supseteq S$ such that S' is a mincut and thus all indices are forbidden. Otherwise, unless we advance by including v in S, the cut can not be a mincut.

■

11.6 An Algorithm to Find Mincuts Satisfying a Lattice-Linear Predicate B

Since finding a mincut satisfying B is NP-complete in general, we focus on special cases of B. We first assume that the given predicate B is lattice-linear. This means that the set of cuts that satisfy B form an inf-semilattice and we have efficient advancement property. Since the set of mincuts form a sublattice, we get that the set of mincuts that satisfy B also forms an inf-semilattice.

We now have a simple algorithm to find a mincut that satisfies B.

The algorithm alternates between finding the mincut after G and finding G that satisfies B. If it cannot find a mincut or a cut satisfying B, it returns null. Assuming that the complexity of computing $\exists j : forbidden(G, j, B)$ is $O(m)$, the above algorithm takes $O(MF(n, m) + mn)$ given the join-irreducible elements of the mincut lattice. The outer while loop executes at most n times because at least one bit in the vector G becomes 1 after every iteration.

We now show that for many lattice-linear predicates, the complexity of the above algorithm is $O(MF(n, m)+kn)$ where k is a parameter dependent upon the predicate B.

To concretize our discussion, consider the predicate $B(S)$ that holds whenever either both vertices u and v belong to S or neither belong to S. We are interested in finding S such that S is a mincut and satisfies $B(S)$.

We note here that the predicate when evaluated over an increasing sets of S, can transition at most twice. Either it is initially false and then turns true. This happens when S initially has u or v, and then later it gets both. Alternatively, it may be initially true, then turn false and then finally turn true. This happens when initially, neither u nor v is in S. Then, one of them gets added to S and finally both of them get added.

The predicate B that can transition at most k times is termed as *k-Transition predicate*. Formally,

Definition 11.9 *A predicate $B : J(P) \to \{true, false\}$ on the ideals of a poset P is a* k-Transition Predicate *if, for any chain of ideals $I_1 \subseteq I_2 \subseteq \cdots \subseteq I_n \in J(P)$, the sequence $B(I_1), B(I_2), \ldots, B(I_n)$ has at most k transitions, where a transition occurs at index i if $B(I_i) \neq B(I_{i+1})$.*

Algorithm LLP-MincutWithB: To find the minimum vector that satisfies B

```
1  vector function getLeastMincutSatisfyingPred( B: predicate)
2  var G: vector of reals initially ∀i : G[i] = 0;
3  G ← LeastMincut
4  while ¬B(G) do
5       while ∃j : forbidden(G, j, B) do
6                 if (G[j] = 1) then return null;
7                 else G[j] := 1;
8       endwhile;
9       if there does not exist LeastMincut ≥ G
10           return null; //no such cut
11      else G ← LeastMincut ≥ G
12 endwhile;
13 return G; // the optimal solution
```

For example, the *stable* predicates [CL85] are 1-Transition predicates. Similarly, predicates that are true on a single mincut [GS24] are also 1-Transition predicates. We will assume that the algorithm for detecting whether B is true on a cut S is $O(m)$.

Thus, if the lattice-linear predicate is a k-transition predicate, then the time complexity is bounded by $O(MF(n,m)+km)$.

11.7 Bibliographic Remarks

The maximum flow problem and its solutions have been extensively studied in combinatorial optimization and network theory. The original Ford-Fulkerson algorithm was introduced in 1956 [JF56], with the foundational Max-Flow Min-Cut Theorem proving its correctness. The Dinitz algorithm [Din70], and the algorithm by Karp and Edmonds [EK72], offered significant improvements with strongly polynomial time guarantees.

More advanced algorithms such as Goldberg and Tarjan's push-relabel method [GT88] provide even better performance in practice and theory, particularly for dense networks. For further reading and comprehensive treatments of flow algorithms, we refer to standard textbooks such as Ahuja, Magnanti, and Orlin [AMO93], Schrijver [Sch03], and Korte and Vygen [KV18].

Chapter 12

Bipartite Matching

12.1 Introduction

In this chapter, we discuss algorithms for the matching problems in bipartite graphs. This chapter is organized as follows. Section 12.2 describes the classical sequential augmenting path algorithm. The algorithm allows one to increase the size of any matching by finding an augmenting path in the bipartite graph. Section 12.3 describes an algorithm to find the chain cover of a poset given a matching algorithm.

12.2 Sequential Algorithm

Given an undirected graph (V, E), a *matching* M is a subset of edges E such that no two edges are incident on the same vertex. As an example, consider a bipartite graph (L, R, E) such that the vertex set L denotes a set of men, R denotes a set of women and E denotes a symmetric "likes" relation, Then, a matching, M, is simply a set of edges such that no man or a woman is adjacent to two edges in M. If the number of men and and the number of women are both equal to n and every man gets matched, then we call that matching, a *perfect matching*. The perfect matching may not always exist. The following famous Theorem gives a necessary and sufficient condition for a perfect matching to exist in a bipartite graph. It is based on the notion of *overdemanded* subset. We say that $L' \subseteq L$ is overdemanded if there exists $R' \subseteq R$ such that the set of neighbors of R' in the graph is L' and the size of R' is strictly bigger than L'. Intuitively, L' is overdemanded because R' can only match with vertices in L' but the size of L' is smaller than size of R'. We can now state the Theorem.

Theorem 12.1 (Hall's Theorem) *The necessary and sufficient condition for a perfect matching to exist in a bipartite graph (L, R, E) where $|L| = |R|$, is that L does not have any overdemanded subset L'.*

Observe that even though Hall's theorem gives us insight about the perfect matching, it cannot be directly applied to efficiently check for perfect matching because there are 2^n possible subsets of L.

We now give an efficient algorithm for a more general problem called the *maximum cardinality matching problem*. This problem requires us to find a matching of the largest size. When the number of men and the number of women are equal to n, then a perfect matching exists iff the size of the maximum cardinality matching is n. The algorithm is based on the idea of increasing the size of a matching by the idea of an *augmenting path*. Suppose the algorithm has a matching M_t at iteration t. Then, the algorithm is guaranteed to find a matching M_{t+1} of size one bigger than M_t whenever a matching with a bigger size exists. If we can make this idea work, then we can start with M_0 as the empty matching and then keep applying the idea to reach a maximum cardinality matching.

To understand the idea of an augmenting path for a matching M, we define a vertex to be *exposed* if it is not incident on any edge in M. Now suppose that there exists a path from an exposed vertex to another exposed vertex that alternates between unmatched and matched edges. Such a path must start with an unmatched edge and end with an unmatched edge (by the definition of exposed vertices). Furthermore, such a path has an odd number of

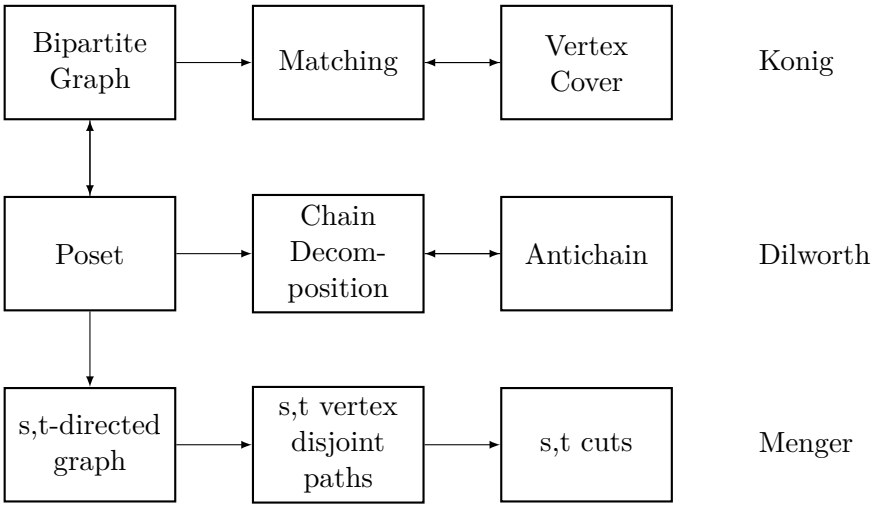


Figure 12.1: Various Structures and Transformations between them

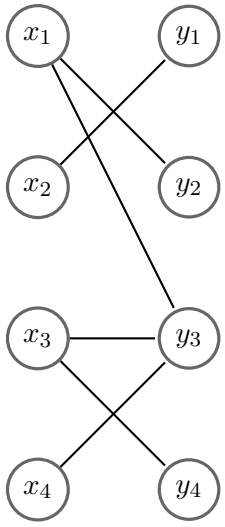


Figure 12.2: A Bipartite Graph

edges with the unmatched edges exactly one more than the matched edges. Then, by simply switching the matched with unmatched edges along that path we can increase the size of matching by one.

Algorithm Seq-AugmentingPath: Finding a maximum cardinality matching

```
1 M := {};
2 while there exists an alternating path P in (L, R, E) for M
3       M := matching M with edges switched in the path P
4 end;
5 return M;
```

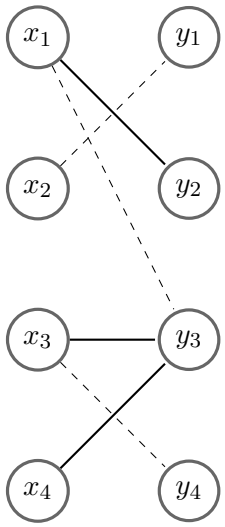


Figure 12.3: A Matching M shown with dashed edges in the Bipartite Graph

How do we find an alternating path? One can start with any exposed vertex and do a breadth-first-search such that the bfs tree alternates between unmatched and matched edges. Such a search will either result in finding an alternating path or an overdemanded set. In our example of the matching M in Fig. 12.3, if we start from x_4, we get the alternating path x_4, y_3, x_1, y_2. If there are $2n$ vertices and m edges, every iteration of the *while* loop takes $O(m)$ time for breadth-first-search and there can be at most n iterations, giving us the time complexity of $O(nm)$. By augmenting along multiple paths in every iteration, the complexity can be reduced to $O(\sqrt{n}m)$ [HK71].

We now give a sequential algorithm for a slight variant of the matching problem in a bipartite graph (L, R, E). Let vertices in L be numbered $1..n$. Let L_i denote the set of vertices $\{v_1, v_2, \ldots, v_i\}$, for all $1 \leq i \leq n$. The output of our algorithm is a vector G such that $\sum_i G[i]$ is the size of the maximum matching in the graph (L_i, R, E_i) where E_i is the edges restricted to the set $L_i \cup R$. For simplicity, we set L_0 to $\emptyset$. We let S be the matched vertices in R. It is sufficient to describe the sequential algorithm when a new vertex v_i is added to the left hand side. We simply look for an alternating path from v_i to any unmatched vertex in R. If we find any such path, then the newly matched vertex in R is added to S. If there is no path, then L_i is a constricted set, and the set of vertices in R matched to L_{i-1} is identical to the set of vertices matched to L_i. In either case, we continue the procedure to the next vertex in L, if any.

Algorithm BipartiteMatching searches the element in the boolean lattice B_n such that $G[i]$ equals 1 if the size of the bipartite matching for (L_i, R, E_i) is one more than for (L_{i-1}, R, E_{i-1}), and 0 otherwise. The time complexity of the algorithm is $O(nm)$ because the *for* loop has n iterations and each iteration takes $O(m)$ to search for a path. The matching itself is stored in all nonzero S entries.

Algorithm BipartiteMatching: A Sequential Algorithm for Bipartite Matching

```
1  input: (L, R, E): bipartite graph
2  var: G: array[1..n] of int ;
3  init: ∀j : G[j] := 0;
4  S: array[1..n] of 0..n;
5  init: ∀j : S[j] := 0;
6  for j := 1 to n do
7      if there exists an alternating path from v_j to any vertex k in R with (S[k] = 0)
8          Assign S[] in the alternating path;
9          G[j] := 1;
10 endfor
11 return G
```

A key advantage of Algorithm BipartiteMatching is that it is a *deterministic* algorithm. Given a labeling of indices in L, it produces a single fixed G.

12.3 Chain Partition of a Poset

In this section, we show that maximum matching in a bipartite graph also allows us to partition a poset into the minimum number of chains.

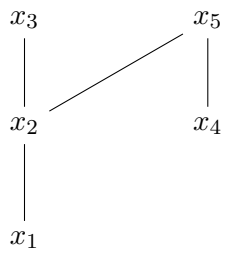


Figure 12.4: A poset

Assume that we have an algorithm for bipartite matching. How do we use it for chain decomposition? This relationship between chain decomposition and bipartite matching is based on the idea of a strict split of a poset.

Definition 12.2 (Strict Split of a Poset) *Given a poset P, the* **strict split** *of the poset P, denoted by $S(P)$ is a bipartite graph (L, R, E) where $L = \{x^-|x \in P\}$, $R = \{x^+|x \in P\}$, and $E = \{(x^-, y^+)|x < y \text{ in } P\}$.*

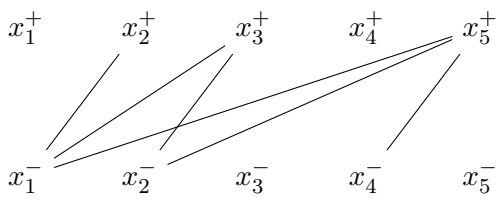


Figure 12.5: A Strict Split of the Poset in Fig. 12.4

We state the following theorem due to Fulkerson [Ful56].

Theorem 12.3 *[Ful56] Any matching in $S(P)$ can be reduced to a chain cover of P.*

Proof: We start with the trivial chain partition of P with each element in P being a chain. Thus, initially we have n chains for n elements. We now consider each edge (x^-, y^+) in the matching. Each such edge implies that $x < y$ in P and we combine the chains corresponding to x and y. Since there can be at most one edge incident on x^- in a matching, x is the top element of its chain and similarly y is the lowest element of the chain. Therefore, these chains can be combined into one chain. If there are e edges in the matching, after this procedure we have a chain cover with $n - e$ chains.

■

Thus one can use any algorithm for bipartite matching to determine a chain cover.

12.4 Problems

1. Show that the minimum size of a vertex cover is equal to the maximum size of a matching in a bipartite graph.
2. Give a linear program formulation for the maximum matching problem and show that its dual corresponds to the minimum vertex cover problem.
3. Prove Hall's Theorem.
4. Edmond's matrix is defined for a bipartite graph (U, V, E) with $|U| = |V| = n$ as follows. Let the vertices in U be $u_1, u_2, \ldots u_n$ and the vertices in V be $v_1, v_2, \ldots, v_n$. We set $A[i, j] = x_{i,j}$ if $(u_i, v_j) \in E$ and 0 otherwise. Show that Edmond's matrix of a balanced bipartite graph has a perfect matching iff the polynomial corresponding to the determinant of Edmond's matrix is not identically zero.
5. We are given an unweighted graph with two distinguished vertices s and t. We are required to find the total number of paths from s to t such that these paths do not share any edge. Give an algorithm to find this number, and justify its correctness. Suppose that the graph has n vertices and m edges, give the time complexity of your algorithm.
6. Modify Algorithm BipartiteMatching to output the vertex cover in addition to bipartite matching.

Bibliographic Remarks

The sequential augmenting path algorithm, detailed in Section 12.2, is a cornerstone of matching theory, achieving a time complexity of $O(nm)$ for a graph with n vertices and m edges. This algorithm is enhanced by the seminal work of Hopcroft and Karp [HK71], who introduced a method to augment multiple paths per iteration, reducing the complexity to $O(\sqrt{n}m)$. Their approach remains a standard for efficient bipartite matching. Hall's Theorem, presented in the chapter, is a fundamental result characterizing the existence of perfect matchings, with roots in early combinatorial optimization. For a deeper exploration of matching theory, including Hall's Theorem and its extensions, Lovász and Plummer [LP86] offer a classic and authoritative reference. Section 12.3 connects bipartite matching to poset chain partitions through the strict split construction, leveraging Fulkerson's theorem [Ful56]. Fulkerson's result elegantly reduces matchings in the bipartite graph $S(P)$ to chain covers in the poset P, demonstrating the interplay between graph theory and order theory.

Chapter 13

Intractability

This chapter discusses NP-completeness, a foundational concept in computational complexity that identifies the hardest problems in NP. We define P and NP, explore polynomial-time reductions, prove NP-completeness for a broad set of problems: 3-SAT, Clique, Vertex Cover, Hamiltonian Path, Independent Set, and TSP. We also discuss methods to deal with NP-complete problem.

The classes P and NP classify computational problems by their solvability and verifiability. P includes problems with polynomial-time deterministic algorithms, while NP includes those verifiable in polynomial time nondeterministically. The unresolved question $P = NP$ drives much of complexity theory.

The problems are specified as requiring an *output* for the given *input*. We will use n for the size of the input. Our interest would be in determining the time required to compute the output. The output that we require would be *binary*. For example, our input may be a boolean expression B on boolean variables. We would require the output to be 1 if the boolean expression is satisfiable and 0 otherwise. At first, it may seem that in practical applications, the user may be interested in finding a satisfying assignment rather than simply that B is satisfiable. However, if somebody gave us a method f to efficiently check whether B is satisfiable then we can use f to determine a satisfying assignment. If B is satisfiable, then we can easily compute another boolean expression B' in which the boolean variable x_1 is set to true. We now ask whether B' is satisfiable. If B' is satisfiable, we continue with other variables. If B' is not satisfiable, then we know that x_1 must be false. We compute a boolean expression B'' from B by setting x_1 to false. Thus, in n applications of f, we would have found a satisfying assignment. This idea is applicable to all problems considered in this chapter and we will restrict ourselves to such problems.

13.1 Class P

The complexity class **P** is the set of all decision problems $L \subseteq \{0,1\}^*$ for which there exists a deterministic Turing machine M and a polynomial function $p : \mathbb{N} \to \mathbb{N}$ such that:

1. M halts on every input $x \in \{0,1\}^*$ in at most $p(|x|)$ steps, where $|x|$ denotes the length of x,
2. M accepts x if and only if $x \in L$.

Formally, $\mathbf{P} = \{L \mid$ there exists a deterministic Turing machine M and a polynomial p such that M decides L in time where n is the input size.

- **P**: The complexity class of decision problems solvable in polynomial time by a deterministic Turing machine. The boldface emphasizes it as a formal class name.
- L: A language, representing a decision problem, which is a set of strings over the binary alphabet $\{0,1\}$.
- M: A deterministic Turing machine, a theoretical model of computation with a single, fixed sequence of steps for any input. For our purposes, we can use any practical programming language such as Java or C++.

- p: A polynomial function $p : \mathbb{N} \to \mathbb{N}$, mapping natural numbers (input sizes) to natural numbers (time bounds).

Informally, a problem L is in P if we can write a program that determines if the given instance x is in L in time that is polynomial in the size of x. Most problems we have seen in this book so far belong to **P**.

13.2 Polytime Reductions

We define a relation between various problems as follows. We say that a problem π_1 is at least as hard as another problem π_2 if by using solutions to the problem π_2 we can solve the problem π_1 in time polynomial in the size of the problem. Formally, π_2 is *polynomially reducible* to π_1 if arbitrary instances of π_2 can be solved using a function that is a polynomial in the size of input and makes at most polynomial calls to a function that solves π_1. We denote this relation as $\pi_2 \leq_P \pi_1$. It is easy to verify that $\leq_P$ is a reflexive and transitive relation.

A consequence of the above definition is that if there exists an efficient algorithm to solve π_1, then we can use that solution to solve π_2. Another consequence, important to this chapter, is that if π_2 is *hard* to solve, then π_1 is also hard to solve.

Independent Set $\leq_P$ Vertex Cover

Let us show that the problem of vertex cover is harder than the problem of independent set in an undirected graph. Let $G = (V, E)$ be an undirected graph with vertex set V and edge set E.

The *Independent Set* problem asks: Given a graph G and an integer k, does there exist a subset $S \subseteq V$ of at least k vertices such that no two vertices in S are adjacent (i.e., $\forall u, v \in S, \{u, v\} \notin E$)? Such a set S is called an *independent set*.

The *Vertex Cover* problem asks: Given a graph G and an integer k, does there exist a subset $C \subseteq V$ of at most k vertices such that every edge in E is incident to at least one vertex in C (i.e., $\forall \{u, v\} \in E, u \in C$ or $v \in C$)? Such a set C is called a *vertex cover*.

We argue that the Vertex Cover problem is at least as hard as the Independent Set problem by providing a polynomial-time reduction from Independent Set to Vertex Cover. Given an instance $(G = (V, E), k)$ of Independent Set, construct an instance of Vertex Cover as follows:

- Use the same graph $G = (V, E)$.
- Set the parameter $k' = |V| - k$.

The reduction is polynomial-time since it only involves computing $|V|$ and subtracting k.

We claim that G has an independent set of size at least k if and only if G has a vertex cover of size at most $k' = |V| - k$.

- ($\Rightarrow$): Suppose $S \subseteq V$ is an independent set with $|S| \geq k$. Define $C = V \setminus S$. Since S is independent, no edge $\{u, v\} \in E$ has both $u, v \in S$. Thus, for every edge $\{u, v\} \in E$, at least one of u or v is in $C = V \setminus S$. Hence, C is a vertex cover. Moreover, $|C| = |V| - |S| \leq |V| - k = k'$.
- ($\Leftarrow$): Suppose $C \subseteq V$ is a vertex cover with $|C| \leq k' = |V| - k$. Define $S = V \setminus C$. For any edge $\{u, v\} \in E$, since C is a vertex cover, at least one of u or v is in C, so at most one is in S. But if both $u, v \in S$, then neither is in C, contradicting that C covers $\{u, v\}$. Thus, no edge connects vertices in S, so S is an independent set. Moreover, $|S| = |V| - |C| \geq |V| - k' = |V| - (|V| - k) = k$.

Since Independent Set is NP-complete, and we have reduced Independent Set to Vertex Cover in polynomial time, Vertex Cover is at least as hard as Independent Set (in the sense that solving Vertex Cover efficiently would imply solving Independent Set efficiently). Formally, Independent Set $\leq_p$ Vertex Cover.

Vertex Cover $\leq_P$Independent Set

Conversely, we argue that the Independent Set problem is at least as hard as the Vertex Cover problem by providing a polynomial-time reduction from Vertex Cover to Independent Set.

Given an instance $(G = (V, E), k)$ of Vertex Cover, construct an instance of Independent Set as follows:

- Use the same graph $G = (V, E)$.
- Set the parameter $k' = |V| - k$.

The reduction is polynomial-time, identical to the previous reduction.

Since we have reduced Vertex Cover to Independent Set in polynomial time, Independent Set is at least as hard as Vertex Cover. Formally, Vertex Cover $\leq_p$ Independent Set.

The bidirectional reductions show that Independent Set and Vertex Cover are computationally equivalent in terms of polynomial-time solvability: Independent Set $\leq_p$ Vertex Cover and Vertex Cover $\leq_p$ Independent Set. The reductions exploit the complementary relationship: S is an independent set if and only if $V \setminus S$ is a vertex cover.

Set Cover Problem: Vertex Cover $\leq_P$ Set Cover

We now show that the problem of *Set Cover* is harder than the vertex cover. The *Set Cover* problem is defined as follows: Given a universe U of n elements $\{e_1, e_2, \ldots, e_n\}$, a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of subsets of U such that $\bigcup_{i=1}^{m} S_i = U$, and an integer k, does there exist a subcollection $\mathcal{C} \subseteq \mathcal{S}$ of at most k subsets such that $\bigcup_{S_i \in \mathcal{C}} S_i = U$? In other words, we seek a cover of U using at most k subsets from $\mathcal{S}$.

It is easy to see that the vertex cover is a special case of set cover. Let $\{e_1, e_2, \ldots, e_n\}$ be the edges in an undirected graph. For each vertex v_i, we define a set S_i as all the edges that are incident on v_i. Then, choosing k sets to cover the universe is identical to choosing k vertices that cover all the edges. Hence, we conclude that Vertex Cover $\leq_P$ Set Cover.

Satisfiability and 3-SAT

So far, our problems were graph-theoretic. Let us now investigate problems that relate to satisfiability of boolean expression. A boolean expression is said to be in Conjunctive Normal Form (CNF) if it is a conjunction of *clauses* where each clause is a disjunction of *literals*. A literal is any boolean variable or its negation. We call the problem of checking satisfiability of a boolean expression in CNF as the problem SAT. We define 3-SAT as the special case of SAT where each clause has exactly three literals. We now investigate the complexity of 3-SAT problem compared to the independent set problem.

SAT $\leq_P$ 3-SAT

It is clear that SAT is a harder problem than 3-SAT because 3-SAT is a special case of the SAT problem. Somewhat surprisingly, we now show that 3-SAT is as hard as SAT problem. Suppose that we are given an instance of the SAT problem. We convert it into 3-SAT problem as follows. Every clause with a single literal can be changed to four clauses with Given a CNF formula $\phi = C_1 \wedge \cdots \wedge C_m$, where $C_i = l_{i1} \vee \cdots \vee l_{ik_i}$, we construct a 3-CNF formula ψ such that ϕ is satisfiable if and only if ψ is satisfiable.

For each clause C_i:

- **Case 1:** $k_i = 1$ (e.g., (l_1)): Introduce dummy variables y_i, z_i. Replace with:

$$(l_1 \vee y_i \vee z_i) \wedge (l_1 \vee y_i \vee \neg z_i) \wedge (l_1 \vee \neg y_i \vee z_i) \wedge (l_1 \vee \neg y_i \vee \neg z_i).$$

- **Case 2:** $k_i = 2$ (e.g., $(l_1 \vee l_2)$): Introduce y_i. Replace with:

$$(l_1 \vee l_2 \vee y_i) \wedge (l_1 \vee l_2 \vee \neg y_i).$$

- **Case 3:** $k_i = 3$: Keep unchanged: $(l_1 \vee l_2 \vee l_3)$.
- **Case 4:** $k_i > 3$ (e.g., $(l_1 \vee \cdots \vee l_k)$): Introduce $y_{i,1}, \ldots, y_{i,k-2}$. Replace with:

$$(l_1 \vee l_2 \vee y_{i,1}) \wedge (\neg y_{i,1} \vee l_3 \vee y_{i,2}) \wedge \cdots \wedge (\neg y_{i,k-3} \vee l_{k-1} \vee l_k).$$

It is easy to verify that ϕ is true iff ψ is true. It is also easily verified that this reduction can be done in polynomial time.

3-SAT $\leq_P$ Independent set

We claim that 3-SAT $\leq_P$ Independent set. Suppose that someone gives us an instance of 3-SAT problem defined as follows: Given a Boolean formula ϕ in conjunctive normal form (CNF), where each clause contains exactly three literals, does there exist an assignment of truth values to the variables that satisfies ϕ? We construct a reduction from a 3-SAT instance ϕ with n variables and m clauses to an Independent Set instance $(G = (V, E), k)$ such that ϕ is satisfiable if and only if G has an independent set of size at least k. The graph G encodes the clauses and variable assignments, ensuring that selecting a valid independent set corresponds to a satisfying truth assignment. Given a 3-SAT formula ϕ with variables $x_1, x_2, \ldots, x_n$ and clauses $C_1, C_2, \ldots, C_m$, where each clause $C_j = (l_{j1} \vee l_{j2} \vee l_{j3})$ and l_{jk} is a literal (x_i or $\neg x_i$), construct the graph $G = (V, E)$ and integer k as follows:

For each clause C_j ($j = 1, 2, \ldots, m$), create three vertices v_{j1}, v_{j2}, v_{j3}, one for each literal l_{j1}, l_{j2}, l_{j3}. These form a *clause gadget* for C_j, represented as a triangle (complete subgraph K_3). Thus, $V = \{v_{j1}, v_{j2}, v_{j3} \mid j = 1, 2, \ldots, m\}$. We construct edges as follows.

- **Within clause gadgets**: For each clause C_j, add edges to form a triangle: $\{v_{j1}, v_{j2}\}$, $\{v_{j2}, v_{j3}\}$, $\{v_{j3}, v_{j1}\}$. This ensures at most one vertex per clause is selected in an independent set.
- **Between clause gadgets**: For any two vertices v_{jk} and $v_{j'k'}$ ($j \neq j'$), add an edge $\{v_{jk}, v_{j'k'}\}$ if the literals l_{jk} and $l_{j'k'}$ are *inconsistent*. Two literals are inconsistent if one is x_i and the other is $\neg x_i$ for the same variable x_i.

Set $k = m$, the number of clauses. We claim that ϕ is satisfiable if and only if G has an independent set of size at least $k = m$.

- ($\Rightarrow$) : Suppose ϕ is satisfiable with a truth assignment $\tau : \{x_1, \ldots, x_n\} \to \{\text{true}, \text{false}\}$. For each clause $C_j = (l_{j1} \vee l_{j2} \vee l_{j3})$, at least one literal, say l_{jk}, is true under τ. Select vertex v_{jk} in G for clause C_j. Construct $S = \{v_{jk} \mid l_{jk} \text{ is true in } C_j, j = 1, \ldots, m\}$, choosing one vertex per clause. Thus, $|S| = m$. It is easily verified that S is independent:
- ($\Leftarrow$): Suppose G has an independent set S with $|S| \geq m$. Since each clause gadget is a triangle, at most one vertex per clause can be in S (vertices v_{j1}, v_{j2}, v_{j3} are pairwise adjacent). With m clauses and $|S| \geq m$, S must include exactly one vertex from each clause's triangle, so $|S| = m$. For each C_j, let $v_{jk} \in S$. Assign truth values to make l_{jk} true:
  - If $l_{jk} = x_i$, set $x_i = \text{true}$.
  - If $l_{jk} = \neg x_i$, set $x_i = \text{false}$.

  This assignment is consistent because if $x_i = \text{true}$ and $x_i = \text{false}$ were assigned (via $v_{jk} = x_i$ and $v_{j'k'} = \neg x_i$ in S), then $\{v_{jk}, v_{j'k'}\} \in E$, contradicting that S is independent. Thus, the assignment satisfies each clause C_j (since l_{jk} is true), making ϕ satisfiable.

13.3 NP-Complete Problems

In this section, we define the class of problems in NP and NP-Complete. As we saw earlier, the class P includes problems that are solvable in $O(n^k)$ time with input size n for some fixed k. The class NP includes problems, where

a *yes* certificate of the problem is verifiable in $O(n^k)$. What is a *yes* certificate? It is an input for all the *yes* instances of the problem that is at most polynomial in size of n and certifies that the given instance is indeed an *yes* instance. For the satisfiability problem, the assignment to the boolean variables gives us a *yes* certificate. The assignment itself is of size n when the boolean expression is on n variables. Furthermore, by substituting the variables using the assignment in the certificate, we can verify in polynomial time that the boolean expression is indeed true. As another example, consider the subset sum problem. We are given a set, S, of integers, another integer t, and we are asked if there is a subset S' such that the subset S' sums to t. Then, the *yes* certificate is simply the set S'. The following polynomial algorithm can be used to check that the certificate is indeed a *yes* certificate.

Algorithm 4: SubsetSumVerify(S, t)

```
   Input  : Set S of integers, target t
   Output : True if a subset sums to t, False otherwise
   Input  : Certificate: Subset S' ⊆ S
1  sum ← 0 ;
2  for each x in S' do
3  |  sum ← sum + x ;
4  end
5  if sum = t then
6  |  return True
7  end
8  return False
```

Observe that we ask for certification for only *yes* instances of the problem.

We now claim that

$$P \subseteq NP.$$

Given any problem π in P, there exists a polynomial time algorithm to solve π. Hence, certification is trivial. It could be just one bit specifying whether π is a *yes* instance or not. Since the problem itself is in polynomial class, we can simply run the algorithm as a checker to verify that the problem is a *yes* instance.

This naturally leads to the problem whether the containment is strict, i.e., is there any problem that is in NP but not in P. In other words, is there a problem for which a solution can be checked efficiently but computing the solution is inefficient. This is the most fundamental problem of the theory of computation.

A problem is *NP-complete* if it is in NP and every NP problem reduces to it in polynomial time. Reductions transform instances while preserving answers, establishing relative hardness.

Theorem 13.1 (Cook-Levin Theorem) *SAT is NP-complete.*

Proof: *Sketch*: SAT is in NP (verify assignment in $O(n)$). Any Nondeterministic computation can be encoded as a satisfiable Boolean formula in polynomial time, reducing all NP problems to SAT.

■

The following figure illustrates the relationship between P, NP-complete, and NP. P is a subset of NP, as polynomial-time solvable problems are verifiable in polynomial time. NP-complete problems are the hardest in NP, with every NP problem reducing to them in polynomial time. If P $\neq$ NP, NP-complete problems are disjoint from P.

3-SAT $\leq_P$ Clique

As yet another reduction, we show that the Clique problem is at least as hard as 3-SAT problem. Given $G = (V, E)$ and integer k, does G have a k-clique?

We construct a polynomial-time reduction from 3-SAT to Clique, showing that a 3-SAT formula ϕ is satisfiable if and only if the constructed graph G has a clique of size at least k. Given a 3-SAT formula ϕ with n variables $x_1, x_2, \ldots, x_n$ and m clauses $C_1, C_2, \ldots, C_m$, where each clause $C_j = (l_{j1} \vee l_{j2} \vee l_{j3})$ and l_{jk} is a literal (x_i or $\neg x_i$), construct an instance $(G = (V, E), k)$ of Clique as follows:

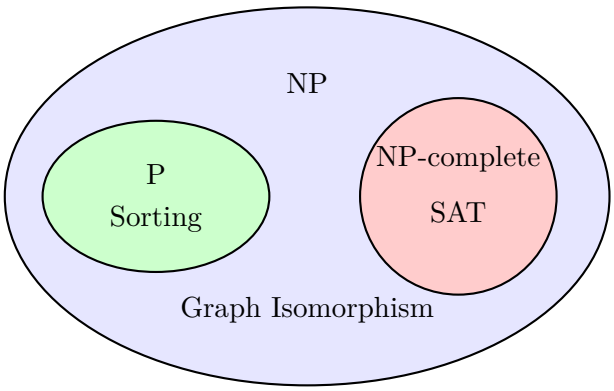


Figure 13.1: Relationship between P, NP-complete, and NP. P is a subset of NP, NP-complete problems are a subset of NP (disjoint from P if P $\neq$ NP), and NP contains problems with polynomial-time verifiable yes instances.

1. **Vertices**: For each literal l_{jk} in clause C_j (i.e., the k-th literal in clause j, where $k = 1, 2, 3$), create a vertex v_{jk}. Thus, $V = \{v_{jk} \mid j = 1, \ldots, m, k = 1, 2, 3\}$, with $|V| = 3m$.
2. **Edges**: Add an edge $\{v_{jk}, v_{j'k'}\}$ between vertices v_{jk} and $v_{j'k'}$ (where $j \neq j'$) if:
   - The literals l_{jk} and $l_{j'k'}$ are *consistent*, meaning they do not contradict each other (e.g., $l_{jk} = x_i$ and $l_{j'k'} = \neg x_i$ is inconsistent, so no edge is added).

   No edges are added between vertices within the same clause ($j = j'$), as they represent literals in the same clause.
3. **Parameter**: Set $k = m$, the number of clauses.

We claim that ϕ is satisfiable if and only if G has a clique of size at least $k = m$.

- ($\Rightarrow$): Suppose ϕ is satisfiable with a truth assignment $\tau : \{x_1, \ldots, x_n\} \to \{\text{true}, \text{false}\}$. For each clause C_j, at least one literal l_{jk} is true under τ. Select one such vertex v_{jk} for each clause C_j, forming $S = \{v_{jk} \mid l_{jk} \text{ is true}, j = 1, \ldots, m\}$. Since we pick one vertex per clause, $|S| = m$. Check if S is a clique:
  - For $v_{jk}, v_{j'k'} \in S$ ($j \neq j'$), an edge exists if l_{jk} and $l_{j'k'}$ are consistent. Since τ is a valid assignment, l_{jk} and $l_{j'k'}$ are both true, so they cannot be inconsistent (e.g., x_i and $\neg x_i$). Thus, $\{v_{jk}, v_{j'k'}\} \in E$.
  - No vertices from the same clause are in S, so intra-clause edges are irrelevant.

  Hence, S is a clique of size $m = k$.
- ($\Leftarrow$): Suppose G has a clique S with $|S| \geq k = m$. Since vertices from the same clause are not connected (no edges within C_j), S can include at most one vertex per clause. With $|S| \geq m$ and m clauses, S must include exactly one vertex per clause, so $|S| = m$. For each clause C_j, let $v_{jk} \in S$, corresponding to literal l_{jk}. Construct a truth assignment τ:
  - If $l_{jk} = x_i$, set $x_i = \text{true}$.
  - If $l_{jk} = \neg x_i$, set $x_i = \text{false}$.

  The assignment is consistent because if $x_i = \text{true}$ (from $v_{jk} = x_i$) and $x_i = \text{false}$ (from $v_{j'k'} = \neg x_i$), then l_{jk} and $l_{j'k'}$ are inconsistent, so $\{v_{jk}, v_{j'k'}\} \notin E$, contradicting that S is a clique. Thus, τ is consistent, and since each clause C_j has a true literal l_{jk}, ϕ is satisfiable.

Problem	**Definition**	**Reduction From**
3-Colorability	Graph G 3-colorable?	SAT
3D-Matching	Perfect matching in $X \times Y \times Z$?	3-SAT
Knapsack	Items fit W, value $\geq V$?	Subset Sum
Hamiltonian Path	Path visits all vertices once?	3-SAT
Hamiltonian Cycle	Cycle visits all vertices once?	Hamiltonian Path
Traveling Salesman	Tour with weight $\leq B$?	Hamiltonian Cycle

Figure 13.2: NP-Complete Problems and Reductions

3D-Matching $\leq_P$ Subset Sum

As yet another reduction, we show that the subset sum problem is harder than the 3D-Matching problem. The *Subset Sum* problem asks whether, given a set of positive integers $S = \{a_1, \ldots, a_n\}$ and a target t, there exists a subset $S' \subseteq S$ summing to t. We prove Subset Sum is NP-complete by showing it is in NP and NP-hard via a polynomial-time reduction from *3D-Matching* (3DM), using only $3n$ digits in base $m+1$ to ensure carry-free addition. It is easy to verrify that the Subset Sum problem is in NP. We reduce 3DM to Subset Sum. Given a 3DM instance with sets $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_n\}$, $Z = \{z_1, \ldots, z_n\}$, and triples $T = \{t_1, \ldots, t_m\} \subseteq X \times Y \times Z$, we construct a Subset Sum instance (S, t) with $3n$ digits in base $m + 1$, such that a perfect matching of size n exists if and only if a subset sums to t. Our construction is as follows.

- **Elements**: $3n$ elements: $x_1, \ldots, x_n$, $y_1, \ldots, y_n$, $z_1, \ldots, z_n$.
- **Base**: Use base $b = m + 1$, where $m = |T|$. Each element appears in at most m triples, so sums in each digit position are at most $n \leq m < b$, ensuring no carry.
- **Numbers**: Create a number a_j for each triple $t_j = (x_{j1}, y_{j2}, z_{j3})$, with $3n$ digits corresponding to elements. In base b:
$$a_j = b^{3n-j_1} + b^{2n-j_2} + b^{n-j_3},$$
where digits for x_{j1}, y_{j2}, z_{j3} (positions $3n - j_1 + 1$, $2n - j_2 + 1$, $n - j_3 + 1$) are 1, others 0.
- **Target**:
$$t = \sum_{k=1}^{n} (b^{3n-k} + b^{2n-k} + b^{n-k}).$$
In base b, t has 1 in each of the $3n$ digit positions.
- **Set** S: $S = \{a_1, \ldots, a_m\}$.

If 3DM has a matching: Let $M \subseteq T$ be a matching of n triples, covering each element exactly once. Then, we select $\{a_j \mid t_j \in M\}$. Each element appears in one triple, so each digit position sums to 1. The sum $\sum_{j \in M} (b^{3n-j_1} + b^{2n-j_2} + b^{n-j_3}) = t$, as each digit is 1 exactly once. Conversely, suppose that the Subset Sum has a solution. Let $S' \subseteq S$ sum to t. t has 1 in each of $3n$ digits. Since each a_j has three 1s, and sums are carry-free, exactly n numbers are chosen ($3n$ ones require n triples). Each element digit is 1, so each element is covered exactly once. The n triples form a perfect matching.

We now list other problems which have been shown to be NP-complete. Table 13.2 summarizes six NP-complete decision problems: 3-Colorability, 3D-Matching, Knapsack, Hamiltonian Path, Hamiltonian Cycle, and Traveling Salesman. For each problem, we provide its definition and identify a known NP-complete problem that can be polynomially reduced to it to prove NP-completeness. Each problem is in NP (solutions are verifiable in polynomial time), and NP-hardness is established via the specified reduction.

1. **3-Colorability**:

- *Definition*: Given a graph $G = (V, E)$, can the vertices be colored with three colors such that no adjacent vertices share the same color?
- *Reduction From*: SAT. Construct a graph with gadgets for variables (true/false colors) and clauses (ensuring at least one true literal), where a 3-coloring encodes a satisfying assignment.

2. **3D-Matching**:
   - *Definition*: Given sets X, Y, Z, each of size n, and triples $T \subseteq X \times Y \times Z$, is there a subset of n triples covering each element exactly once?
   - *Reduction From*: 3-SAT. Create variable gadgets (true/false triples) and clause gadgets (triples for true literals), where a matching corresponds to a satisfying assignment.
3. **Knapsack**:
   - *Definition*: Given items with weights w_i, values v_i, capacity W, and target V, is there a subset with weight $\leq W$, value $\geq V$?
   - *Reduction From*: Subset Sum. Set $v_i = w_i$, $W = t$, $V = t$, making Knapsack equivalent to Subset Sum.
4. **Hamiltonian Path**:
   - *Definition*: Given a graph $G = (V, E)$, is there a path visiting each vertex exactly once?
   - *Reduction From*: 3-SAT. Build a graph with variable paths (true/false) and clause nodes, where a Hamiltonian path encodes a satisfying assignment.
5. **Hamiltonian Cycle**:
   - *Definition*: Given a graph $G = (V, E)$, is there a cycle visiting each vertex exactly once?
   - *Reduction From*: Hamiltonian Path. Add a vertex connected to all others, where a Hamiltonian cycle implies a Hamiltonian path in the original graph.
6. **Traveling Salesman**:
   - *Definition*: Given a complete graph with edge weights and bound B, is there a tour of weight $\leq B$?
   - *Reduction From*: Hamiltonian Cycle. Set weights: 1 for original edges, 2 otherwise, $B = n$. A tour of weight n is a Hamiltonian cycle.

13.4 co-NP Class of Problems

In computational complexity theory, the class co-NP plays a crucial role alongside NP in understanding the structure of decision problems. The complexity class co-NP consists of all decision problems whose *complements* are in NP. Formally, a decision problem $L \subseteq \{0,1\}^*$ is in co-NP if its complement $\overline{L} = \{0,1\}^* \setminus L$ is in NP. Equivalently, a problem L is in co-NP if there exists a polynomial-time verifiable certificate for *no* instances (i.e., instances $x \notin L$). This means there is a polynomial-time Turing machine M and a polynomial p such that for every input $x \notin L$, there exists a certificate c of length at most $p(|x|)$ where $M(x, c) = 1$, and for every $x \in L$, no such certificate exists.

In contrast, a problem L is in NP if there exists a polynomial-time verifiable certificate for *yes* instances ($x \in L$). Thus, co-NP is the class of problems where the evidence for rejection is efficiently verifiable, while NP focuses on evidence for acceptance.

The relationship between co-NP and NP is symmetric due to their complementary definitions:

- If $L \in$ NP, then $\overline{L} \in$ co-NP.
- If $L \in$ co-NP, then $\overline{L} \in$ NP.

This symmetry implies that NP and co-NP are *dual* classes. Notably:

- P $\subseteq$ NP $\cap$ co-NP, since if a problem is in P, both its yes and no instances can be decided in polynomial time, providing trivial certificates.
- NP$\cap$co-NP contains problems like Integer Factorization (deciding if a number n has a factor d with $1 < d < n$), where both existence (NP) and non-existence (co-NP) of factors are verifiable with certificates (a factor or a primality certificate).

A central question is whether NP = co-NP. If NP = co-NP, then for every NP problem, its complement would also have efficiently verifiable yes certificates, implying a symmetry in verification. However, it is widely believed that NP $\neq$ co-NP, as their equality would collapse the polynomial hierarchy and have profound implications for problems like SAT and UNSAT.

The following figure illustrates the relationship between NP, co-NP, and P. P is explicitly drawn as a distinct region within the intersection of NP and co-NP, representing problems solvable in polynomial time. NP and co-NP are dual classes, where a problem is in co-NP if its complement is in NP. Their intersection, NP $\cap$ co-NP, contains P and problems like Integer Factorization, not known to be in P.

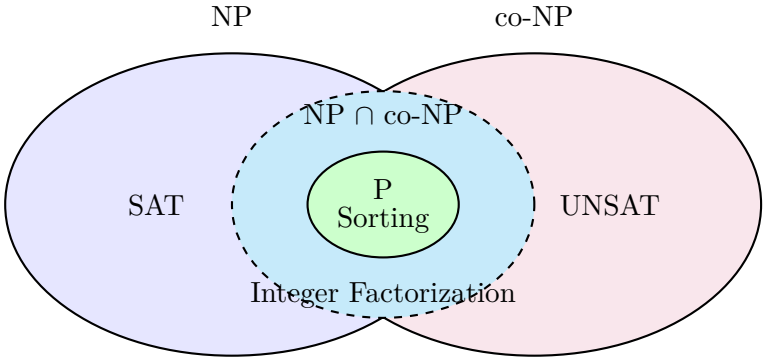


Figure 13.3: Relationship between NP, co-NP, and P. P is drawn within NP $\cap$ co-NP, which also includes problems like Integer Factorization. It is unknown if NP = co-NP.

We present several examples of problems in co-NP, focusing on their complements in NP and their verification mechanisms.

- The *Tautology* problem asks: Given a Boolean formula ϕ in disjunctive normal form (DNF), is ϕ true for all possible truth assignments (i.e., is ϕ a tautology)? The complement is: Is ϕ not a tautology? This is in NP because a no instance (non-tautology) has a certificate: a truth assignment making ϕ false, verifiable in polynomial time by evaluating ϕ. Thus, Tautology is in co-NP.
- The *Unsatisfiability*(UNSAT) problem asks: Given a Boolean formula ϕ in conjunctive normal form (CNF), is ϕ unsatisfiable (i.e., no truth assignment makes ϕ true)? The complement is SAT (Satisfiability), which is in NP: a yes instance of SAT has a certificate (a satisfying assignment) verifiable in polynomial time. Thus, UNSAT is in co-NP.
- The *Non-Isomorphism* problem for graphs asks: Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, are G_1 and G_2 not isomorphic (i.e., there exists no bijection $f : V_1 \to V_2$ preserving edges)? The complement, Graph Isomorphism, is in NP: a yes instance has a certificate (a bijection f) verifiable in polynomial time by checking edge preservation. Thus, Non-Isomorphism is in co-NP. It is not known if Non-Isomorphism is co-NP-complete.

Several open problems in complexity theory involve co-NP, reflecting its central role in understanding computational limits. We highlight key questions:

- Does NP = co-NP? The most significant open problem is whether NP = co-NP. If NP = co-NP, then for every NP-complete problem like SAT, its complement (UNSAT) would also be in NP, implying efficient certificates

for unsatisfiability. This would collapse the polynomial hierarchy, as the hierarchy relies on the distinction between NP and co-NP. Most researchers believe NP $\neq$ co-NP due to the intuitive asymmetry: verifying satisfiability (NP) seems easier than verifying unsatisfiability (co-NP).

- Which problems lie in NP$\cap$co-NP? Known examples include Primality and Integer Factorization, but identifying others (e.g., Graph Isomorphism) is open. It is conjectured that NP$\cap$co-NP $\neq$ NP and NP$\cap$co-NP $\neq$ co-NP, but the boundaries are unclear.

13.5 Approximation Algorithms

Approximation algorithms offer polynomial-time solutions for NP-complete optimization problems.

Example: Vertex Cover Approximation

Algorithm 5: ApproxVertexCover(G)

Input : Graph $G = (V, E)$
Output : Vertex cover C

$C \leftarrow \emptyset$;
$E' \leftarrow E$;
while $E' \neq \emptyset$ **do**
  Pick any edge $(u, v) \in E'$;
  $C \leftarrow C \cup \{u, v\}$;
  Remove all edges incident to u or v from E' ;
end
return C

Time: $O(m)$.
Ratio: 2-approximation.
It is easy to see that each edge requires at least one vertex in OPT. The algorithm picks two per iteration, so $|C| \leq 2 \cdot |OPT|$.

13.6 Problems

1. You are given a graph $G = (V, E)$ with non-negative lengths $w_e \geq 0$ for all edges $e \in E$. You are also given a source node s and destination node t and an integer K. Show that deciding whether there is an s-t path of length exactly K is NP-complete.

2. Consider the *Job Scheduling with Deadlines* problem: Given n jobs with processing times p_i, deadlines d_i, and profits r_i, and a single processor, select a subset of jobs to maximize total profit such that each selected job completes by its deadline (job i takes p_i time units, must finish by d_i, and jobs are non-preemptive). Prove this problem is NP-complete.

13.7 Bibliographic Remarks

The Cook-Levin Theorem (Theorem 13.1), independently established by Cook [Coo71] and Levin [Lev73], marks SAT as the first NP-complete problem, showing that every NP problem can be reduced to SAT in polynomial time. This result laid the groundwork for identifying other NP-complete problems, as detailed in the chapter's reductions (e.g., SAT $\leq_P$ 3-SAT, 3-SAT $\leq_P$ Independent Set, 3-SAT $\leq_P$ Clique). Karp [Kar72] significantly expanded this landscape by proving NP-completeness for 21 fundamental problems, including Clique, Vertex Cover, and Hamiltonian Path,

many of which are covered in the chapter's Figure 13.2. For a comprehensive treatment of NP-completeness and its reductions, Garey and Johnson [GJ79] remains a definitive reference.

Polynomial-time reductions, as illustrated in Sections 13.2 and 13.3, are central to establishing the hardness of problems like Independent Set, Vertex Cover, and 3-SAT. These reductions exploit structural relationships, such as the complementary nature of Independent Set and Vertex Cover, to demonstrate computational equivalence. The chapter's treatment of co-NP (Section 13.4) introduces the dual class of problems with efficiently verifiable no certificates, with examples like UNSAT and Tautology. Papadimitriou [Pap94] offers an authoritative resource on complexity classes, including NP, co-NP, and their intersections.

Chapter 14

The Housing Allocation Problem

14.1 Introduction

The housing market problem proposed by Shapley and Scarf [SS74] is a matching problem with one-sided preferences. There are n agents and n houses. Each agent a_i initially owns a house h_i for $i \in \{1, n\}$ and has a completely ranked list of houses. There are variations of this problem when the agents do not own any house initially. In this paper, we focus on the version with the initial endowment of houses for the agents. The list of preferences of the agents is given by $pref[i][k]$ which specifies the k^{th} preference of the agent i. Thus, $pref[i][1] = j$ means that a_i prefers h_j as his top choice. The goal is to come up with an optimal house allocation such that each agent has a house and no subset of agents can improve the satisfaction of agents in this subset by exchanging houses within the subset. It can be shown that there is a unique such matching called the *core* for any housing market. The standard algorithm for this problem is Gale's Top Trading Cycle Algorithm that takes $O(n^2)$ time. This algorithm is optimal in terms of the time complexity since the input size is $O(n^2)$. Our interest in this paper is to design parallel algorithms for this problem.

In this chapter, we focus on computing the core and give a parallel algorithm for finding the core that is nearly linear in the number of agents. Our algorithm takes expected $O(n \log^2 n)$ time and expected $O(n^2 \log n)$ work.

Section 14.2 gives Gale's Top Trading Cycle Algorithm. Section 14.3 applies LLP method to the unconstrained Housing market problem and derives a high-level parallel algorithm.

14.2 Gale's Top Trading Cycle Algorithm

Consider the housing market instance shown in Fig. 14.1. There are four agents a_1, a_2, a_3 and a_4. Initially, the agent a_i holds the house h_i. The preferences of the agents is shown in Fig. 14.1.

The Top Trading Cycle (TTC) algorithm attributed to Gale by Shapley and Scarf [SS74] works in stages. At each stage, it has the following steps:

Step 1. We construct the *top choice* directed graph $G_t = (A, E)$ on the set of agents A as follows. We add a directed edge from agent $a_i \in A$ to agent $a_j \in A$ if a_j holds the current top house of a_i. Fig. 14.2 shows the directed graph at the first stage.

Step 2. Since each node has exactly one outgoing edge in G_t, there is at least one cycle in the graph (possibly, a self-loop). All cycles are node disjoint. We find all the cycles in the top trading graph and implement the trade indicated by the cycles, i.e, each agent which is in any cycle gets its current top house.

Step 3. Remove all agents which get their current top houses and remove all houses which are assigned to some agent from the preference list of remaining agents.

$a_1 : h_2, h_3, h_1, h_4$	$a_1 : h_1$	$a_1 : h_2$
$a_2 : h_1, h_4, h_2, h_3$	$a_2 : h_2$	$a_2 : h_1$
$a_3 : h_1, h_2, h_4, h_3$	$a_3 : h_3$	$a_3 : h_4$
$a_4 : h_2, h_1, h_3, h_4$	$a_4 : h_4$	$a_4 : h_3$
Agents' Preferences	Initial Allocation	Matching returned by the TTC algorithm

Figure 14.1: Housing Market and the Matching returned by the Top Trading Cycle Algorithm

The above steps are repeated until each agent is assigned a house. At each stage, at least one agent is assigned a final house. Thus, this algorithm takes $O(n)$ stages in the worse case and needs $O(n^2)$ computational steps.

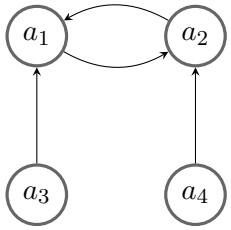


Figure 14.2: The top choice graph at the first stage.

14.3 Applying LLP Algorithm to the Housing Market Problem

We model the housing market problem as that of predicate detection in a computation. There are n agents and n houses. Each agent proposes to houses in the decreasing order of preferences. These proposals are considered as events executed by n processes representing the agents. Thus, we have n events per process. Each event is labeled as (i, h, k), which corresponds to the agent i proposing to the house h as his choice number k.

The global state corresponds to the number of proposals made by each of the agents. Let $G[i]$ be the number of proposals made by the agent i. We will assume that in the initial state every agent has made his first proposal. Thus, the initial global state $G = [1, 1, .., 1]$. We extend the notation of indexing to subsets $J \subseteq [n]$ such that $G[J]$ corresponds to the subvector given by indices in J.

We now model the possibility of reallocation of houses based on any global state. Recall that $pref[i][k]$ specifies the k^{th} preference of the agent a_i. Let $wish(G, i)$ denote the house that is proposed by a_i in the global state G, i.e.,

$$wish(G, i) = pref[i][G[i]]$$

A global state G satisfies *matching* if every agent proposes a different house, i.e.,

$$matching(G) \equiv \forall i, j : i \neq j : wish(G, i) \neq wish(G, j).$$

We generalize *matching* to refer to a subset of agents rather than the entire set.

Definition 14.1 (submatching) *Let $J \subseteq [n]$. Then, $submatching(G, J)$ iff $wish(G, J)$ is a permutation of indices in J.*

Intuitively, if $submatching(G, J)$ holds, then all agents in J can exchange houses within the subset J.

For any G, it is easy to show that

Lemma 14.2 *For all G, there always exists a nonempty J such that submatching(G, J).*

Proof: Given any G, we can create a directed graph as follows. The set of vertices is agents and there is an edge from i to j if $wish(G, i) = j$. There is exactly one outgoing edge from any vertex in $[n]$ to $[n]$ in this graph. This implies that there is at least one cycle in this graph (possibly, a self-loop). The indices of agents in the cycle gives us such a subset J.

■

We now show that

Lemma 14.3 *submatching(G, J_1) and submatching(G, J_2) implies that submatching$(G, J_1 \cup J_2)$.*

Proof: Any index $i \in J_1 \cup J_2$ is mapped to J_1 if $i \in J_1$ and J_2, otherwise.

■

Hence, there exists the biggest submatching in G. Note that $matching(G)$ is equivalent to $submatching(G, [n])$.

Definition 14.4 (Feasible Global State) *A global state G is feasible for the housing market problem iff it is a matching and for all global states $F < G$, there does not exist any submatching which is better in F than in G. Note that if there exists a submatching J which is better in F than G, then the agents in J can improve their allocation by just exchanging houses within the subset J. Formally, let*

$$B_{housing}(G) \equiv matching(G) \wedge (\forall F < G : \forall J \subseteq [n] : submatching(F, J) \Rightarrow F[J] = G[J]).$$

We show that $B_{housing}(G)$ is a lattice-linear predicate. This result will let us use the lattice-linear predicate detection algorithm for the housing market problem.

Theorem 14.5 *The predicate $B_{housing}(G)$ is lattice-linear.*

Proof: Suppose that $\neg B_{housing}(G)$. This implies that either G is not a matching or it is a matching but there exists a smaller global state F that has a submatching better than G.

First, consider the case when G is not a matching. Let J be the largest set such that $submatching(G, J)$. Consider any index $i \notin J$ such that $wish(G, i) \in J$. We claim that $forbidden(G, i, B_{housing})$. Let H be any global state greater than G such that $G[i] = H[i]$. We consider two cases.

Case 1: $H[J] > G[J]$.
Then, from the second conjunct of $B_{housing}$, we know that $\neg B_{housing}(H)$ because $submatching(G, J)$ and $H[J] \neq G[J]$.

Case 2: $H[J] = G[J]$.
Since $wish(H, i) = wish(G, i)$, $wish(G, i) \in J$, and $G[J] = H[J]$, we get that H is not a matching because the house given by $wish(G, i)$ is also in the wish list of some agent in J.

Now consider the case when G is a matching but $\neg B_{housing}(G)$. This implies

$$\exists F < G : \exists J \subseteq [n] : submatching(F, J) \wedge F[J] < G[J]).$$

However, the same F will also result in guaranteeing $\neg B_{housing}(H)$ for any $H \geq G$.

■

Algorithm LLP-Housing-Market-Algorithm: A high-level parallel algorithm to find the optimal house market

```
1 var
2     G: array[1..n] of int initially 1;// every agent starts with the top choice
3 always
4     S(G) = largest J such that submatching(G, J)
5     forbidden(G, j, B) ≡ (j ∉ S(G)) ∧ (wish(G, j) ∈ S(G))
6         advance: G[j] := G[j] + 1;
```

It is also easy to see from the proof that if an index is part of a submatching, then it will never become forbidden.

This theorem gives us the algorithm shown in Fig. LLP-Housing-Market-Algorithm. Let G be the initial global state. Let $S(G)$ be the biggest submatching in G. All agents such that they are not in $S(G)$ and wish a house which are part of $S(G)$ are forbidden and can move to their next proposal. The algorithm terminates when no agent is forbidden. This algorithm is a parallel version of the top trading cycle (TTC) mechanism attributed to Gale in [SS74].

We now show that

Theorem 14.6 *There exists at least one feasible global state G such that $B_{housing}(G)$.*

Proof: Every agent has his own house in the list of preferences. If he ever makes a proposal to his own house, he forms a submatching. That particular event is never forbidden because it is a part of a submatching. Hence, lattice-linear predicate detection algorithm will never mark that event as forbidden. Since such an event exists for all processes, we are guaranteed to never go beyond this global state.

■

The above proof also shows that agents can never be worse-off by participating in the algorithm. Each agent will either get his own house back or get a house that he prefers to his own house.

14.4 Problems

1. Give an NC algorithm to determine if the given matching is the core of the housing market.

14.5 Bibliographic Remarks

The housing market problem has been studied by many researchers [SS74, HZ79, Zho90, AS98, AS99, RP77, Rot82, Dav13]. Possible applications of the housing market problem include: assigning virtual machines to servers in cloud computers, allocating graduates to trainee positions, professors to offices, and students to roommates. This problem has also recently been studied by Zheng and Garg [ZG19] where it is shown that the problem of verifying that a matching is a core is in NC, but the problem of computing the core is CC-hard[1] The paper [ZG19] also gives a *distributed* message-passing algorithm to find the core with $O(n^2)$ messages. The parallel algorithm is taken from [Gar21].

[1]The class CC (Comparator Circuits) is the complexity class containing decision problems which can be solved by comparator circuits of polynomial size.

Chapter 15

The Assignment Problem

15.1 Introduction

In this chapter we consider the assignment problem in which we seek an assignment of items to bidders such that the total payoff is maximized. By viewing items and bidders as nodes of a bipartite graph, and the weight $v_{b,i}$ of an edge between bidder b and item i as the utility of assigning the item i to the bidder b, the problem corresponds to finding the maximum weight bipartite matching.

The assignment problem has a rich history. The most famous algorithm for this problem is due to an American scientist Kuhn who called it the Hungarian algorithm because it is inspired by the earlier works of two Hungarian mathematicians — Konig and Egervary. Another American mathematician, Munkres, observed that the algorithm is strongly polynomial and the algorithm is also known as Kuhn-Munkres algorithm. It was later discovered that the problem had earlier been solved by Jacobi in 19th century and was published in Latin in year 1890.

15.2 Problem Formulation

Let I be a set of indivisible n items, and U, a set of n bidders. Every item $i \in I$ is given a *valuation* $v_{b,i}$ by each bidder $b \in U$. The valuation of any item i is an integer between 0 and $T[i]$. Each item i is given a price $G[i]$ which is also an integer between 0 and $T[i]$.

We will assume that the number of items is equal to the number of bidders. If one of the set is smaller, say there are fewer items than bidders, then we can add virtual items such that all bidders give valuation of zero to those items. We also note that we are considering the maximum weight bipartite matching. By subtracting the valuation from the maximum valuation for each edge, and calling it the cost of that edge, we can transform this problem to the minimum cost bipartite matching.

It is important to note that it is the relative valuation of the items that is important not the absolute values. Suppose that there are three items and a bidder provides valuation of $(10, 8, 4)$ for these items, then the assigned matching will not change if his valuation is $(6, 4, 0)$ instead. The weight of the matching found for the valuation $(6, 4, 0)$ would be exactly less by 4 because in each assignment, a bidder is assigned exactly one item. Hence, one can assume that there is at least one item for each bidder that is valued as 0. Similarly and dually, suppose that an item is valued by three bidders as $(10, 8, 4)$. Then, the assignment will not change if this item is valued as $(6, 4, 0)$ instead. If we view V as the square matrix such that $V[b, i]$ equals $v_{b,i}$, then we can assume that there is a zero in every row by subtracting the minimum value in each row from every entry in the row. Similarly, we can assume that there is at least one zero in every column.

Let $x_{b,i}$ be defined as a boolean variable which is 1 iff the item i is assigned to the bidder b, and 0 otherwise. We consider the relaxation of the above problem in which $x_{b,i}$ are relaxed from being boolean variables to just being

nonnegative reals, giving us the following primal linear program.

$$\begin{aligned} \text{maximize} \quad & \sum_{(b,i)\in[n]\times[n]} v_{b,i}x_{b,i} \\ \text{subject to} \quad & \sum_{b} x_{b,i} = 1 \quad \forall i \in [n], \\ & \sum_{i} x_{b,i} = 1 \quad \forall b \in [n], \\ & x_{b,i} \geq 0 \quad \forall i \in [n], b \in [n] \end{aligned} \tag{15.1}$$

We consider the dual of the above linear program. For each constraint on the item, we define the dual variable p_i and for each constraint on the bidder, we define the dual variable q_b. The dual program is:

$$\begin{aligned} \text{minimize} \quad & \sum_{i\in[n]} p_i + q_i \\ \text{subject to} \quad & p_i + q_b \geq v_{b,i} \quad \forall i \in [n], b \in [n], \\ & p_i \geq 0 \quad \forall i \in [n], \\ & q_b \geq 0 \quad \forall b \in [n] \end{aligned} \tag{15.2}$$

The constraint $p_i + q_b \geq v_{b,i}$ can be seen as the no blocking pair constraint if the assignment problem is used to match the sellers and the buyers of the items. If $p_i + q_b < v_{b,i}$, then the assignment is not *stable* from the perspective of the seller of item i and the bidder b. Together, they can be viewed as a *blocking pair* analogous to the blocking pair in the stable matching problem. They have the motivation to leave the existing assignment and get matched so that together they improve their total payoff.

One way to solve a linear program is to find feasible solutions to primal and dual programs that satisfy complementary slackness conditions. The primary complementary slackness conditions are $\forall i, b$: $(x_{b,i} = 0) \vee (p_i + q_b = v_{b,i})$. The secondary complementary slackness conditions are trivially true for this formulation because we have only tight constraints in the primal program.

15.3 Market Clearing Price

In this section, we apply LLP technique to the problem of finding a market clearing price. The set of feasible price vectors is given by

$$\{G \mid 0 \leq G[i] \leq T, 1 \leq i \leq n\}$$

This set of price vectors forms a distributive lattice under the component-wise comparison of vectors. The minimum is given by the zero vector and the maximum is given by the vector T.

Given a price vector G, we define the bipartite graph $(I, U, E(G))$ as follows. One side of the bipartite graph is the set of items I. The other side of the graph is the set of bidders U. We now add edges between items and the bidders as follows.

$$(j, b) \in E(G) \equiv \forall i : (v_{b,j} - G[j]) \geq (v_{b,i} - G[i]).$$

Informally, an edge exists between item i and bidder b if the payoff for the bidder (the bid minus the price) is maximized with that item. Given any set $U' \subseteq U$, let $N(U', G)$ denote all the items that are adjacent to the vertices in U' in the graph $(I, U, E(G))$. A price vector G is a *market clearing price* if the bipartite graph $(I, U, E(G))$ has a perfect matching.

We construct a computation graph $(E, \rightarrow)$ for this problem as follows (see Fig. 15.1). There are n processes corresponding to n items in the computation such that each process has C events. In Fig. 15.1, we have three items and three bidders such that both prices and valuations are integers between 0 and 4. If the process i has executed k events then, then $G_i = k$. The global state G can be viewed as the price vector p where $G[i] = p[i]$. A price vector

G is a *market clearing price*, denoted by $B_{clearingPrice}(G)$ if the bipartite graph $(I, U, E(G))$ has a perfect matching. We now claim that

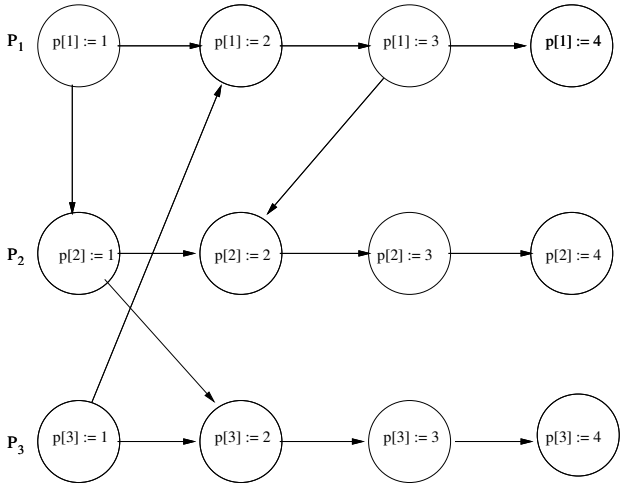


Figure 15.1: The computation graph for a market with three items and three bidders. The valuation of and price for any item is a number between 0 and 4.

Lemma 15.1 *The predicate $B_{clearingPrice}(G)$ is a lattice-linear predicate on the lattice of price vectors.*

Proof: Suppose that the price vectors G and H satisfy $B_{clearingPrice}$. Let $K = min(G, H)$. We show that K also satisfies $B_{clearingPrice}$. Suppose that $(I, U, E(K))$ does not have a perfect matching. This implies that there exists a minimal overdemanded set of items in K. Let an item i be one of the minimal overdemanded items in K. Without loss of generality assume that $K[i]$ equals $G[i]$ (the argument for the case when $K[i]$ equals $H[i]$ is identical). The item i is not an overdemanded item in G. But this is a contradiction because for all other items i', the price of the item i' has either stayed the same or has increased in going from K to G.

■

In Fig. 15.2, we have used $\alpha(G, j)$ as simply one unit of price. For any item j that is part of a minimal overdemanded set of items, we can increase its price by the minimum amount to ensure that some bidder b can switch to her second most preferred item. We now give an implementation LLP-Assignment based on this idea.

Suppose that your friend gives you an assignment and claims that the assignment is optimal. It is easy to check that the assignment is proper, i.e., every item is assigned to exactly one bidder. But, how can she quickly convince you of the optimality? Of course, one could compute the optimal assignment and check that the value of the optimal assignment is same as that given by your friend. However, our goal is to perform strictly less work than solving the problem.

Your friend only needs to provide the price vector G along with the assignment. Suppose that the item i is assigned to the bidder b, then we can set p_i to G, and q_b to $v_{b,i} - p_i$. Now, it is sufficient to check that these variables satisfy the constraints that

$$\forall i, j : p_i + q_j \geq v_{i,j}$$

Note that this check can be performed in $O(1)$ time in parallel. In essence, we have verified that both primal and dual variables are feasible and satisfy complementary slackness conditions.

Now suppose that the assignment is not optimal. Without computing the optimal assignment, how can you convince your friend that her assignment is not optimal. The answer is given by the following theorem.

Theorem 15.2 *An assignment is optimal iff there is no positive weight cycle with respect to the assignment.*

P_j: Code for thread j
shared var G: array[$1..n$] of $0..maxint$;
Market Clearing Prices: Demange-Gale-Sotomayor algorithm
input: $v[b,i]$: int for all b,i
init: $G[j] := 0$;
always:
$E = \{(k,b) \mid \forall i : (v[b,k] - G[k]) \geq (v[b,i] - G[i])$;
$demand(U') = \{k \mid \exists b \in U' : (k,b) \in E\}$;
$overDemanded(J) \equiv \exists U' \subseteq U : (demand(U') = J) \wedge (|J| < |U'|)$
forbidden(j): $(\exists minimal\ J : OverDemanded(J) \wedge (j \in J)$
advance: $G[j] := G[j] + 1$;

Figure 15.2: **Algorithm ConstrainedMarketClearingPrice** to find the minimum cost assignment vector

Algorithm LLP-Assignment: Finding the minimum clearing price vector

```
1  var
2      G: real initially ∀i : G[i] = 0;
3  while(true)
4      E := {(i,b) | ∀j : (v_{b,i} − G[i]) ≥ (v_{b,j} − G[j])};
5      if there exists a perfect matching M in E
6          then return M
7      else
8          J := minimal OverDemanded set in the bipartite graph with edges E(G);
9          forall j ∈ J in parallel do
10             δ_j = min{(v_{b,j} − G[j]) − max_{i≠j}(v_{b,i} − G[i]) | (j,b) ∈ E(G)}
11             G[j] := G[j] + δ_j;
12 end
```

Proof: A positive weight cycle is a cycle of even length of alternating matched and unmatched edges such that the weight of unmatched edges is larger than the weight of matched edges. If such a cycle exists, then by switching matched and unmatched edges we get a matching with higher weight.

Conversely, if the matching is not optimal, we consider XOR of this matching with an optimal matching. It can be shown that these edges must have a positive weight cycle.

■

15.4 Constrained Market Clearing Price

We now generalize the problem of finding a market clearing price to that of finding a constrained market clearing price. For example, constraints on the clearing prices of the form $(G[j] \geq k) \Rightarrow (G[i] \geq k')$ for $1 \leq k, k' \leq C$ are lattice-linear. The constraint says that if item j is priced at least k, then item i must be priced at least k'. The constraint $G[i] \geq G[j]$ is also lattice-linear. Observe that the constraint $G[i] \geq G[j]$ is equivalent to

$$(G[j] \geq 1 \Rightarrow G[i] \geq 1) \wedge (G[j] \geq 2 \Rightarrow G[i] \geq 2) \ldots (G[j] \geq C \Rightarrow G[i] \geq C)$$

Similarly, the constraint $(G[i] = G[j])$ can be modeled as $(G[i] \geq G[j]) \wedge (G[j] \geq G[i])$. Also, constraint of the form $G[j] \geq f(G)$ for monotone f is lattice-linear. Given any set of valuations, and a boolean predicate B that is a conjunction of lattice-linear constraints, a price vector G is a *constrained market clearing price*, denoted by $constrainedClearing(G)$ iff $clearing(G) \wedge B(G)$. Since $B(G)$ is lattice-linear, it is sufficient to give an algorithm for $clearing(G)$. It follows that the set of constrained market clearing price vectors is closed under meets. By applying the lattice-linear predicate detection, we get an algorithm to compute the least constrained market clearing price shown in Fig. 15.2. We get a generalization of Demange, Gale and Sotomayor's exact auction mechanism [DGS86] to incorporate lattice-linear constraints on the market clearing price.

We construct a computation graph $(E, \rightarrow)$ for this problem as follows (see Fig. 15.3). For any constraint, $(G[j] \geq k) \Rightarrow (G[i] \geq k')$, we put an edge from the event k' in P_i to the event k in P_j. By putting such edges, we get a computation graph and any price vector that does not satisfy constraints is not a consistent global state.

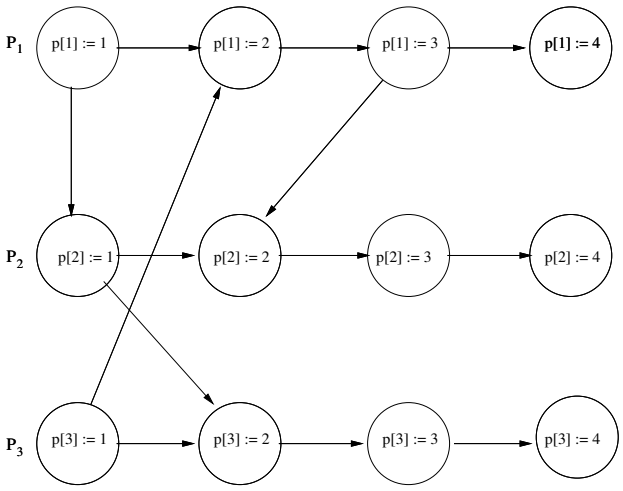


Figure 15.3: The computation graph for a market with three items and three bidders. The valuation of and price for any item is a number between 0 and 4. The computation graph models the constraint $B \equiv (p[2] \geq 2 \Rightarrow p[1] \geq 3) \wedge (p[1] \geq 2 \Rightarrow p[3] \geq 1)$

For example, suppose that the valuation for three items in Fig. 15.3 is as follows. The bidder 1 values them as $[4, 1, 0]$, the bidder 2 values them as $[4, 1, 2]$ and the bidder 3 values them as $[4, 2, 0]$. In the initial global state (with no

events on any process), the price vector is $[0, 0, 0]$. This is not a market clearing price since item 1 is overdemanded. Hence, we advance on P_1 and the new price vector is $[1, 0, 0]$. The item 1 continues to be overdemanded and we advance on P_1 again to get the price vector $[2, 0, 0]$. This is a market clearing price; however, it does not satisfy the constraint that $(p[1] \geq 2 \Rightarrow p[3] \geq 1)$. Hence, we must advance on P_3 to get the price vector $[2, 0, 1]$. This price vector satisfies the constraints but is not a clearing price. We must advance on the overdemanded item 1, to get the price vector $[3, 0, 1]$ which is both clearing and satisfies constraints. The price vector $[3, 0, 1]$ is the least price vector that is clearing and satisfies constraints. At this price vector, item 1 can be assigned to bidder 1, item 2 to bidder 3 and item 3 to bidder 2.

15.5 Problems

1. Show that the set of market clearing price vectors are also closed under the join operation.

15.6 Bibliographic Remarks

Kuhn's Hungarian method to solve the assignment problem is from [Mun57]. Demange-Gale-Sotomayor present the auction-based algorithm [DGS86] for market clearing prices.

Chapter 16

Horn and 2-SAT Satisfiability

16.1 Introduction

It is well-known than checking satisfiability of a boolean expression is a NP-complete problem. In this chapter we consider two important classes of boolean expressions for which satisfiability can be solved efficiently: Horn formulas and 2-SAT formulas.

Throughout this chapter, we assume that the predicate is given in the *conjunctive normal form* (CNF) . A boolean formula B in a conjunctive normal form is simply a conjunction of *clauses*, where each clause is a pure disjunction of *literals*. A literal is a variable in its simple form or in a complemented form. For example, suppose that we have n boolean variables $\{x_1, x_2, \ldots x_n\}$. Then, the formula $(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2)$ is in CNF with two clauses. The first clause has three literals $\{\neg x_1, \neg x_2, x_3\}$, and the second clause has two literals $\{x_1, x_2\}$.

16.2 Horn Satisfiability

A clause is a *Horn clause* if it has at most one positive literal. An example of a Horn clause is $(\neg x_1 \vee \neg x_2 \vee x_3)$. Note that this formula is also equivalent to $(x_1 \wedge x_2) \Rightarrow x_3$. A Horn clause written in this form is also called a *Horn implication* or a *definite clause*. Notice that it has only positive literals on the left hand side and a single positive literal on the right hand side. The left hand side may be empty. Thus, x_3 is also a Horn implication equivalent to $true \Rightarrow x_3$.

Another example of a Horn clause is $(\neg x_1 \vee \neg x_2 \vee \neg x_3)$. This clause does not have any positive literal. It is a *pure negative* clause.

A Horn formula is just a conjunction of Horn clauses. The problem of HornSat is to determine if the given Horn formula is satisfiable.

Consider the following Horn formula B with three variables x_1, x_2, and x_3:

$$(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_1) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2)$$

This Horn formula B is satisfiable. Consider the assignment: $x_1 =$ false, $x_2 =$ true, and $x_3 =$ false. The first and the fourth clauses are trues because x_1 is false, and the second and the third clauses are true because x_3 is false. Another assignment that satisfies B is $x_1 =$ false, $x_2 =$ false, and $x_3 =$ false. In fact, this is the *least* satisfying assignment in the Boolean lattice of three variables.

Consider the following set of Horn clauses with two variables x_1 and x_2:

$$(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge x_1$$

This set of Horn clauses is unsatisfiable because there is no assignment of truth values to x_1 and x_2 that can satisfy all the clauses simultaneously.

16.3 LLP Algorithm for HornSat

We assume that the Horn formula uses n variables $x_1, \ldots x_n$. These n boolean variables define a boolean lattice L of size 2^n that consists of boolean vectors of size n. We first show that a Horn Formula is a lattice linear predicate with respect to L.

Lemma 16.1 *Any Horn Formula is a lattice linear predicate.*

Proof: Let $B = B_1 \wedge B_2 \wedge B_m$ be the Horn formula where each B_i is a Horn clause. Since lattice linear predicates are closed under conjunction, it is sufficient to show that every Horn clause B_i is lattice linear. We consider two types of Horn clauses.

First suppose that B_i is a Horn implication, i.e., $B_i \equiv (x_{i_1} \wedge x_{i_2} \wedge x_{i_{k-1}} \Rightarrow x_{i_k})$. Consider any boolean vector G in which B_i is false. This implies that $x_{i_1}, x_{i_2}, \ldots x_{i_{k-1}}$ are true in G but x_{i_k} is false. Consider any H such that $H \geq G$ and $H[i_k] = G[i_k]$. Since $H \geq G$, we have that $(x_{i_1} \wedge x_{i_2} \wedge x_{i_{k-1}})$ holds in H. Furthermore, since $H[i_k]$ is equal to $G[i_k]$, x_{i_k} is false in H. Hence, B_i is also false in H.

Now suppose that B_i is a pure negative clause, i.e., $B_i \equiv (\neg x_{i_1} \vee \neg x_{i_2} \vee \neg x_{i_k})$. Suppose that B_i is false in G. This implies that all the literals are true. Consider any boolean vector $H \geq G$. H must also have all these literals true and therefore B_i is false in H (and any index is trivially forbidden).

■

The proof of Lemma 16.1 also shows us the appropriate advancement function. For Horn implications, we must advance on the right hand side literal. For pure negative clauses, all boolean vectors greater than G do not satisfy B. Hence, the algorithm can simply return that "no satisfying vector exists."

We can now apply LLP algorithm to find a satisfying assignment for a Horn formula.

Algorithm HornSAT-LLP: An Algorithm to find the least assignment that satisfies B

```
1 var G: vector of boolean initially ∀i : G[i] = false;
2 P_j: code for thread j
3 forbidden: ∃h : all antecedents of B_h are true, they imply x_j, and x_j is false in G
4     advance: G[j] := true;
5 forbidden: ∃h : a negative clause B_h is false, and x_j is a part of B_h
6     return null; // "no satisfying assignment exists"
7 return G; // the least assignment that satisfies B
```

We leave an efficient implementation of the algorithm HornSAT-LLP as an exercise.

Observe that we also get the following consequence of Lemma 16.1 from its lattice-linearity.

Corollary 16.2 *If G and H satisfy Horn formula, then $G \sqcap H$ also satisfies that formula.*

A direct proof of this fact is left as an exercise. Although the set of assignments satisfying a Horn formula are closed under meets, they are not closed under joins. For example, consider the Horn formula $B \equiv (\neg x_1 \vee \neg x_2)$. The bit vectors $G = [1, 0]$ and $H = [0, 1]$ satisfy B but their join $[1, 1]$ does not satisfy B.

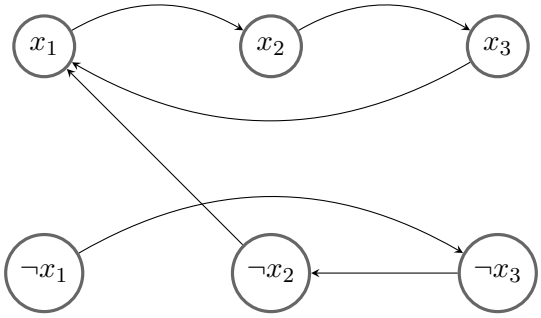


Figure 16.1: Implication Graph of the predicate $B \equiv (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$

16.4 Arithmetization of Horn Clauses

In this section, we give another proof that Horn Clauses are lattice linear by considering arithmetic expressions instead of boolean expressions. We use the natural assignment of boolean variables x_i to integer binary variables y_i. If x_i is false then y_i is assigned value 0, otherwise it is assigned 1. Given any clause, we translate it into an arithmetic predicate as follows. Every positive literal x_i in any clause is changed to y_i and every negative literal $\neg x_i$ is changed to $(1 - y_i)$. The clause is true iff the sum of all values so replaced in any clause is at least 1. For example, the clause $(\neg x_1 \vee \neg x_2 \vee x_3)$ is equivalent to $(1 - y_1) + (1 - y_2) + y_3 \geq 1$. If all y_i take values in the set $\{0, 1\}$, then it is easy to verify that the clause is true for $x's$ iff the corresponding arithmetic predicate is true for $y's$. Let us see how implication Horn clauses get translated. An implication Horn clause $(x_{i_1} \wedge x_{i_2} \wedge x_{i_{k-1}} \Rightarrow x_{i_k})$ gets translated to $(1-y_{i_1})+(1-y_{i_2})+\ldots(1-y_{i_{k-1}})+y_{i_k} \geq 1$ which is equivalent to $y_{i_k} \geq y_{i_1}+y_{i_2}+\ldots+y_{i_{k-1}}-(k-1)$. The right hand side is a monotone function of y; hence the predicate is lattice-linear. A pure negative clause $(\neg x_{i_1} \vee \neg x_{i_2} \ldots \vee \neg x_{i_k})$ gets translated to $(1 - y_{i_1}) + (1 - y_{i_2}) \ldots + (1 - y_{i_k}) \geq 1$. This predicate is equivalent to $k - 1 \geq y_{i_1} + y_{i_2} + \ldots + y_{i_k}$ which once it becomes false it stays false. Hence, predicates for negative clauses are also lattice-linear.

16.5 2-SAT

A conjunctive normal form formula is a 2-SAT formula iff every clause has at most two literals. For example, the formula $B_1 \equiv x_1 \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3)$ is a 2-SAT formula. If we consider the lattice of boolean vectors, a 2-SAT formula may not be closed under meet. For example, consider the 2-SAT formula $(x_1 \vee x_2)$. The boolean vectors 10 and 01 satisfy the formula, but their meet 00 does not. Hence, there is no *minimum* satisfying assignment and simple LLP algorithm is not applicable.

We now outline an approach to detect if there is any element in the lattice that satisfies B is a 2-SAT formula. The key idea for 2SAT algorithm is to convert the predicate to an implication graph. We assume that every clause has two literals. A unit clause x_i or $\neg x_i$ forces the value of the variable. It can be replaced in B resulting in a 2SAT formula with fewer variables. Now, we construct an implication graph $I(B)$ for B as follows. The graph has $2n$ vertices, one for each literal. Every clause $(l_i \vee l_j)$ in B is true iff $(\neg l_i \Rightarrow l_j) \wedge (\neg l_j \Rightarrow l_i)$. Therefore, we add two directed edges in the the graph — from $\neg l_i$ to l_j and from $\neg l_j$ to l_i.

Consider the following 2-SAT formula with three variables x_1, x_2, and x_3:

$$(\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$$

The implication graph on B is shown in Fig. 16.1.

We leave the following as an exercise.

Exercise 16.1 *1. There exists a satisfying assignment to 2-SAT iff there is no variable x_i such that $\neg x_i$ is reachable from x_i and x_i is reachable from $\neg x_i$ in the graph I.*

2. *Suppose B is satisfiable. Then, all satisfying assignments set x_j to true iff there exists a path from $\neg x_j$ to x_j in the implication graph.*

3. *Suppose B is satisfiable. Then, all satisfying assignments set x_j to false iff there exists a path from x_j to $\neg x_j$ in the implication graph.*

Algorithm 2SAT: An Algorithm to find a feasible assignment that satisfies B

```
1  vector function getFeasible(B: predicate)
2  var G: array[1...n] of boolean initially ∀i : G[i] = false;
3      fixed: array[1...n] of boolean initially ∀i : fixed[i] = false;
4      I(B) := implication graph of B;
5      if there exists a path from ¬x_j to x_j and from x_j to ¬x_j in I(B) then
6          return null; //"no satisfying solution"
7      while (∃j : ¬fixed[j])
8          if there exists a path from ¬x_j to x_j then G[j] := true;
9          fixed[j] := true;
10         unitPropagation(B, j);
11     endwhile;
12     return G; // a feasible solution
```

Algorithm 2SAT works as follows. The array G keeps the assignment of all the binary variables. The array $fixed$ maintains for each i, whether the bit x_i has been set to true or false. The **while** loop in the figure sets these variables as follows. It picks any variable x_j that is not fixed. If there is a path from $\neg x_j$ to x_j, then we set x_j to true; otherwise it keeps the value of x_j as false. We also set the bit $fixed[j]$ to true. Once x_j is set, other variables may also be forced. We call the method *unitPropagation* that sets the other variables that are set because x_j is set. We continue the **while** loop until all variables are set.

In the algorithm, we need to check if there exists a path from x_j to $\neg x_j$ and vice-versa. One way to accomplish this is to find strongly connected components (SCCs) in the graph. An SCC is a subgraph where every vertex is reachable from every other vertex in the same SCC. (One can use Tarjan's algorithm or Kosaraju's algorithm to find SCCs.) The expression is satisfiable if and only if for every variable x_j and $\neg x_j$ are not in the same SCC. If the expression is satisfiable, an assignment can be constructed by considering the SCCs in reverse topological order. The sequential time complexity of 2SAT problem is linear in the number of clauses.

Let us now check the time complexity for a parallel algorithm to determine if the given 2SAT expression is satisfiable. The problem reduces to checking if there exists a path from x_j to $\neg x_j$ and vice-versa using reachability. Since reachability problem is in NC, we conclude that checking satisfiability of a 2SAT formula is in NC.

16.6 Problems

1. Is the following Horn SAT formula satisfiable?
   $(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_1) \wedge (\neg x_3 \vee \neg x_1) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2)$

2. **(Efficient Satisfiability of Horn formulas)** Give an efficient implementation of the algorithm in Fig. HornSAT-LLP. In particular give efficient data structures to represent Horn implications and pure negative clauses. Your algorithm should be linear in the length of the Horn formula.

3. **(Renamable Horn formulas)** A formula is a Renamable Horn formula if by flipping the polarity of certain variables the formula can be transformed to a Horn formula. For example, $(\neg x_1 \vee \neg x_2 \vee \neg x_4) \wedge (x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee \neg x_4) \wedge (x_1 \vee \neg x_3)$ is not a Horn formula. However, by renaming x_1 as $\neg x_1$ and x_3 as $\neg x_3$, we get

a Horn formula. Give an algorithm to determine whether the given formula is a Renamable Horn formula. (Hint: Can you come up with a 2SAT formula that is true iff the given formula is renamable Horn formula).

4. **(Meet Closure of Horn Formulas)** Prove Corollary 16.2 without invoking lattice linearity of Horn Clauses.

5. **(Dual Horn formulas)** A dual Horn formula is one in which all clauses have at most one negative literal. Give an algorithm to check satisfiability of dual Horn formulas.

6. Consider a directed graph with n nodes including two distinguished nodes s and t. Show that there exists a 2-SAT formula B on n variables such that the formula is satisfiable iff t is reachable from s.

7. **(Satisfying Assignments of 2-SAT formulas)** Suppose a 2-SAT formula B is satisfiable. Then, all satisfying assignments of B must have x_j as true iff there exists a path from $\neg x_j$ to x_j in the implication graph.

8. **(Implication Graph of a 2-SAT formula)** (a) Show that there exists a satisfying assignment to a 2-SAT formula B iff there is no variable x_i such that $\neg x_i$ is reachable from x_i and x_i is reachable from $\neg x_i$ in the implication graph I constructed from B. (b) Show that every literal in any strongly connected component of the implication graph must be assigned the same value, *true* or *false*.

Chapter 17

Stable Marriage Problem with Ties

17.1 Introduction

In this chapter we consider the stable matching problem when the list of preferences may have ties. We will assume that the number of men is equal to the number of women in this chapter. The problem of stable marriage with ties is clearly more general than the standard stable matching problem discussed in Chapter 2.

We consider three versions of matching with ties. In the first version, called *weakly stable* matching M, there is no blocking pair of man and woman (m, w) who are not married in M but strictly prefer each other to their partners in M. In the second version, called *superstable* matching M, we require that there is no blocking pair of man and woman (m, w) who are not married in M but either (1) both of them prefer each other to their partners in M, or (2) one of them prefers the other over his/her partner in M and the other one is indifferent, or (3) both of them are indifferent to their spouses. The third version, called *strongly stable matching*, we require that if there is no blocking pair (m, w) such that they are not married in M but either (1) both of them prefer each other to their partners in M, or (2) one of them prefers the other over his/her partner in M and the other one is indifferent.

For the weakly stable matching, ties can be broken arbitrarily and any matching that is stable in the resulting instance is also weakly stable for the original problem. Therefore, Gale-Shapley algorithm is applicable for the weakly stable matching. We now derive LLP algorithms for the superstable and strongly stable matchings.

17.2 Superstable Matching

In many applications, agents (men and women for the stable marriage problem) may not totally order all their choices. Instead, they may be indifferent to some choices [Irv94, Man02]. We generalize $mpref[i][k]$ to a set of women instead of a single woman. Therefore, $mrank$ function is not 1-1 anymore. Multiple women may have the same rank. Similarly, $wrank$ function is not 1-1 anymore. Multiple men may have the same rank. We now define the notion of blocking pairs for a matching M with ties [Irv94]. We let $M(m)$ denote the woman matched with the man m and $M(w)$ denote the man matched with the woman w. In the version, called *weakly stable* matching M, there is no blocking pair of man and woman (m, w) who are not married in M but strictly prefer each other to their partners in M. Formally, a pair of man and woman (m, w) is *blocking for a weakly stable matching* M if they are not matched in M and
$(mrank[m][w] < mrank[m][M(m)]) \wedge$
$(wrank[w][m] < wrank[w][M(w)]$.

For the weakly stable matching, ties can be broken arbitrarily and any matching that is stable in the resulting instance is also weakly stable for the original problem. Therefore, Gale-Shapley algorithm is applicable for the weakly stable matching [Irv94]. We focus on other forms of stable matching — superstable and strongly stable matchings.

A matching M of men and women is *superstable* if there is no blocking pair (m, w) such that they are not married in M but they either prefer each other to their partners in M or are indifferent with their partners in M. Formally, a pair of man and woman (m, w) is *blocking for a super stable matching* M if they are not matched in M and
$(mrank[m][w] \leq mrank[m][M(m)]) \wedge$
$(wrank[w][m] \leq wrank[w][M(w)]$.

The algorithms for superstable marriage have been proposed in [Irv94, Man02]. Our goal is to show that LLP algorithm is applicable to this problem as well. As before, we will use $G[i]$ to denote the $mrank$ that the man i is currently considering. Initially, $G[i]$ is 1 for all i, i.e., each man proposes to all his top choices. We say that G has a superstable matching if there exist n women $w_1, w_2, \ldots w_n$ such that $\forall i : w_i \in mpref[i][G[i]]]$ and the set (m_i, w_i) is a superstable matching.

We define a bipartite graph $Y(G)$ on the set of men and women with respect to any G as follows. If a woman does not get any proposal in G, then she is unmatched. If she receives multiple proposals then there is an edge from that woman to all men in the most preferred rank. We say that $Y(G)$ is a perfect matching if every man and woman has exactly one adjacent edge in $Y(G)$,

We claim

Lemma 17.1 *If $Y(G)$ is not a perfect matching, then there is no superstable matching with G as the proposal vector.*

Proof: If there is a man with no adjacent edge in $Y(G)$ then it is clear that G cannot have a superstable matching. Now consider the case when a man has at least two adjacent edges. If all the adjacent women for this man have degree one, then exactly one of them can be matched with this man and other women will remain unmatched. Therefore, there is at least one woman w who is also adjacent to another man m'. If w is matched with m, then (m', w) is a blocking pair. If w is matched with m', then (m, w) is a blocking pair.

■

We now claim that the predicate $B(G) \equiv$ "$Y(G)$ is a perfect matching$''$ is a lattice-linear predicate.

Lemma 17.2 *If $Y(G)$ is not a perfect matching, then at least one index in G is forbidden.*

Proof: Consider any man i such that there is no edge adjacent to i in $Y(G)$. This happens when all women that man i has proposed in state G have rejected him. Consider any H such that $H[i]$ equals $G[i]$. All the women had rejected man i in G. As H is greater than G, these women can only have more choices and will reject man i in H as well.

Now suppose that every man has at least one adjacent edge. Let $Z(G)$ be the set of women with degree exactly one. If every woman is in $Z(G)$, then we have that $Y(G)$ is a perfect matching because every man has at least one adjacent edge. If not, consider any man i who is not matched to a woman in $Z(G)$. This means that all the women he is adjacent to have degrees strictly greater than one. In H all these women would have either better ranked proposals or equally ranked proposals. In either case, man i would not be matched with any of these women. Hence, i is forbidden.

■

We are now ready to present LLP-ManOptimalSuperStableMarriage. In LLP-ManOptimalSuperStableMarriage, we start with the proposal vector G with all components $G[j]$ as 1. Whenever a woman receives multiple proposals, she rejects proposals by men who are ranked lower than anyone who has proposed to her. We say that a man j is forbidden in G, if every woman z that man j proposes in G is either engaged to or proposed by someone who she prefers to j or is indifferent with respect to j. LLP-ManOptimalSuperStableMarriage is a parallel algorithm because all processes j such that forbidden(j) is true can advance in parallel. In LLP-ManOptimalSuperStableMarriage, we use $Y(j)$ to denote all women than man j has proposed in the state G, i.e., $Y(j)$ equals $mpref[j][G[j]]$.

Let us consider an example on three men and three women. The preference lists for men are:
$m_1 : (w_1 = w_2), w_3$
$m_2 : w_2, (w_1 = w_3)$

$m_3 : w_3, w_1, w_2$

The preference lists for women are:
$w_1 : (m_2 = m_3), m_1$
$w_2 : m_1, m_3, m_2$
$w_3 : (m_1 = m_3), m_2$

The algorithm starts with $G = (1, 1, 1)$. The man m_1 sends proposals to both w_1 and w_2. The man m_2 sends proposal to w_2 and m_3 sends proposal to w_3. Since w_3 receives only proposal from m_3, she accepts it. Since w_2 receives proposals from m_1 as well as m_2, she sends a reject to m_2 because she prefers m_1. Since w_1 receives proposal from only m_1, she also accepts the proposal. The man m_2 is forbidden because all his proposals are rejected and he move to the next level in his preferences. The man $_2$ now proposes to both w_1 and w_3. The woman w_1 prefers m_2 to m_1 so she rejects m_1 and accepts m_2. The woman w_3 prefers m_3 to m_2, so she rejects m_2. At this point, we have m_1 with accepted proposal from w_2, m_2 with accepted proposal from w_1 and m_3 with accepted proposal from w_3. The marriage $(m_1, w_2), (m_2, w_1), (m_3, w_3)$ is the optimal super stable marriage.

Algorithm LLP-ManOptimalSuperStableMarriage: A Parallel Algorithm for Man-Optimal Super Stable Matching

```
1 P_j: Code for thread j
2 input: mpref[i,k]: set of int for all i,k; wrank[k][i]: int for all k,i;
3 init: G[j] := 1;
4 always: Y(j) = mpref[j][G[j]];
5 forbidden(j):
6       ∀z ∈ Y(j) : ∃i ≠ j : ∃k ≤ G[i] : (z ∈ mpref[i][k]) ∧ (wrank[z][i] ≤ wrank[z][j]))
7 // all women z in the current proposals from j have been proposed by someone who either they
    prefer or are indifferent over j.
8       advance: G[j] := G[j] + 1;
```

Let us verify that this algorithm indeed generalizes the standard stable marriage algorithm. For the standard stable marriage problem, $mpref[i, k]$ is singleton for all i and k. Hence, $Y(j)$ is also singleton. Using z for the singleton value in $Y(j)$, we get the expression $\exists i \neq j : \exists k \leq G[i] : (z = mpref[i][k]) \wedge (wrank[z][i] < wrank[z][j]))$ which is identical to the stable marriage problem once we substitute $<$ for $\leq$ for comparing the $wrank$ of man i and man j.

When the preference list has a singleton element for each rank as in the classical stable marriage problem, we know that there always exists at least one stable marriage. However, in presence of ties there is no guarantee of existence of a superstable marriage. Consider the case with two men and women where each one of them does not have any strict preference. Clearly, for this case there is no superstable marriage.

By symmetry of the problem, one can also get woman-optimal superstable marriage by switching the roles of men and women. Let $mpref[i].length()$ denote the number of equivalence classes of women for man i. When all women are tied for the man i, the number of equivalence classes is equal to 1, and when there are no ties then it is equal to n. Consider the distributive lattice L defined as the cross product of $mpref[i]$ for each i. We now have the following result.

Theorem 17.3 *The set of superstable marriages, $L_{superstable}$, is a sublattice of the lattice L.*

Proof: From Lemma 17.2, the set of superstable marriages is closed under meet. By symmetry of men and women, the set is also closed under join.

■

It is already known that the set of superstable marriages forms a distributive lattice [Spi95]. The set of join-irreducible elements of the lattice $L_{superstable}$ forms a partial order (analogous to the rotation poset [GI89]) that can be used to generate all superstable marriages. Various posets to generate all superstable marriages are discussed in [Sco05, HG21].

We note that the algorithm LLP-ManOptimalSuperStableMarriage can also be used to find the constrained superstable marriage. In particular, the following predicates are lattice-linear:

1. Regret of man i is at most regret of man j.
2. The proposal vector is at least I.

17.3 Strongly Stable Matching

A matching M of men and women is *strongly stable* if there is no blocking pair (m, w) such that they are not married in M but either (1) both of them prefer each other to their partners in M, or (2) one of them prefers the other to his/her partner in M and the other one is indifferent. Formally, a pair of man and woman (m, w) is *blocking for a strongly stable matching* M if they are not matched in M and

$$\begin{array}{l} ((mrank[m][w] \leq mrank[m][M(m)]) \wedge \\ \quad (wrank[w][m] < wrank[w][M(w)])) \\ \vee((mrank[m][w] < mrank[m][M(m)]) \wedge \\ \quad (wrank[w][m] \leq wrank[w][M(w)])). \end{array}$$

As in superstable matching algorithm, we let $mpref[i][k]$ denote the set of women ranked k by man i. As before, we will use $G[i]$ to denote the $mrank$ that the man i is currently considering. Initially, $G[i]$ is 1 for all i, i.e., each man proposes to all his top choices. We define a bipartite graph $Y(G)$ on the set of men and women with respect to any G as follows. If a woman does not get any proposal in G, then she is unmatched. If she receives multiple proposals then there is an edge from that woman to all men in the most preferred rank. For superstable matching, we required $Y(G)$ to be a perfect matching. For strongly stable matching, we only require $Y(G)$ to contain a perfect matching.

We first note that a strongly stable matching may not exist. The following example is taken from [Irv94].

$m_1 : w_1, w_2$
$m_2 : w_1 = w_2$ (both choices are ties)

$w_1 : m_2, m_1$
$w_2 : m_2, m_1$

The matching $\{(m_1, w_1), (m_2, w_2)\}$ is blocked by the pair (m_2, w_1): w_1 strictly prefers m_2 and m_2 is indifferent between w_1 and w_2. The only other matching is $\{(m_1, w_2), (m_2, w_1)\}$. This matching is blocked by (m_2, w_2): w_2 strictly prefers m_2 and m_2 is indifferent between w_1 and w_2.

Consider any bipartite graph with an equal number of men and women. If there is no perfect matching in the graph, then by Hall's theorem there exists a set of men of size r who collectively are adjacent to fewer than r women. We define *deficiency* of a subset Z of men as $|Z| - N(Z)$ where $N(Z)$ is the *neighborhood* of Z (the set of vertices that are adjacent to at least one vertex in Z). The deficiency $\delta(G)$ is the maximum deficiency taken over all subsets of men. We call a subset of men Z *critical* if it is maximally deficient and does not contain any maximally deficient proper subset. Our algorithm to find a strongly stable matching is simple. We start with G as the global state vector with top choices for all men. If $Y(G)$ has a perfect matching, we are done. The perfect matching in $Y(G)$ is a strongly stable matching. Otherwise, there must be a critical subset of men with the maximum deficiency. These set of men must then advance on their proposal number, if possible. If these men cannot advance, then there does not exist a strongly stable marriage and the algorithm terminates.

LLP-ManOptimalStronglyStableMarriage is the LLP version of the algorithm proposed by Irving and the interested reader is referred to [Irv94] for the details and the proof of correctness. Similar to superstable marriages, we also get the following result.

Algorithm LLP-ManOptimalStronglyStableMarriage: A Parallel Algorithm for Man-Optimal Strongly Stable Matching

```
1 P_j: Code for thread j
2 input: mpref[i,k]: set of int for all i,k; wrank[k][i]: int for all k,i;
3 init: G[j] := 1;
4 always: Y(j) = mpref[j][G[j]];
5 forbidden(j):
6       j is a member of the critical subset of men in the graph Y(G)
7       advance: G[j] := G[j] + 1;
```

Theorem 17.4 *The set of strongly stable marriages, $L_{stronglystable}$, is a sublattice of the lattice L.*

Observe that each element in $L_{stronglystable}$ is not a single marriage but a set of marriages. This is in contrast to $L_{superstable}$, where each element corresponds to a single marriage.

17.4 Problems

1. Give all the super stable marriages for the preference lists given below. The preference lists for men are:
   $m_1 : (w_1 = w_2), w_3$
   $m_2 : w_2, (w_1 = w_3)$
   $m_3 : w_3, w_1, w_2$

   The preference lists for women are:
   $w_1 : (m_2 = m_3), m_1$
   $w_2 : m_1, m_3, m_2$
   $w_3 : (m_1 = m_3), m_2$
2. Show the correctness of the LLP-ManOptimalStronglyStableMarriage.

17.5 Bibliographic Remarks

The notion of superstable matching and strongly stable matching and the associated algorithms are from [Irv94]. The LLP versions of these algorithms are from [Gar23].

Chapter 18

The GCD Problem

18.1 Introduction

In this chapter we cover some basic algorithms in number theory. The set of all natural numbers form a distributive lattice under the usual *divides* relation, i.e., we define $x \leq y$ for two natural numbers x and y if x divides y. In this (infinite) lattice, the bottom element is 1 since it divides all numbers. The join of any two elements is given by their *least common multiple* and the meet of any two elements is given by their *greatest common divisor*.

In this chapter, we first give two algorithms for the greatest common divisor.

18.2 Parallel Greatest Common Divisor (GCD) Algorithm: First Algorithm

Let A be an array of n natural numbers. All the common divisors of A form a distributive lattice. This lattice is never empty because 1 divides all numbers. Our goal is to find the greatest common divisor of these numbers. Instead of finding just one number, we find an array of numbers G such that each number in G divides all number in A. When G is the greatest such vector then we have the required GCD.

We start with $G[i]$ equal to $A[i]$ for all i. We assume that all of them are numbers greater than or equal to 1. The algorithm terminates with all numbers identical and equal to the gcd.

The predicate

$$B \equiv (\forall i : gcd(A)\ divides\ G[i]) \wedge (\forall i, j : G[i] \geq G[j])$$

We first need to show that B is lattice-linear.

Lemma 18.1 *B is lattice-linear.*

Proof: Suppose that B is false. Since the first part is invariant of the program, we get that there exists i and j such that $G[j]$ is strictly bigger than $G[i]$. Since G can only decrease, the predicate can never become true unless $G[j]$ is advanced, i.e., decreased. This decrease must be such that the invariant is maintained.

■

Suppose that $G[i]$ is greater than $G[j]$ for some j and $G[j]$ does not divide $G[i]$, then we claim that it is safe to reduce $G[i]$ to $G[i] mod\ G[j]$. This step gives us an algorithm. If all numbers are same, we stop. Otherwise, if $G[i] > G[j]$, we replace $G[i]$ by $G[j]$ if $G[i]$ is a multiple of $G[j]$ and $G[i] mod\ G[j]$ otherwise.

The invariant is that every common divisors of A is also a common divisor of G. The invariant holds initially because G is set to A. After every advance, we only need to consider two cases. If $G[j]$ is greater than $G[i]$ and $G[i]$

```
var
  G: array[1..n] of int initially ∀i : G[i] = A[i];
forbidden(j): ∃i : G[j] > G[i]
    advance(j): if G[i] divides G[j] then G[j] := G[i];
          else G[j] := G[j] mod G[i];
```

Figure 18.1: **Algorithm GCD** to find the greatest common divisor of a set of numbers

divides $G[j]$, then it is sufficient to set $G[j]$ to $G[i]$ because just the divisors of $G[i]$ are common divisors to both $G[i]$ and $G[j]$. If $G[i]$ does not divide $G[j]$, then the only common divisors are those that divide $G[i]$ and $G[j] mod\ G[i]$ (prove it!).

It is easy to see that the algorithm terminates. All values are initially non-zero positive integers. They stay non-zero and integral and always decrease; therefore, the algorithm must terminate. Initially G is same as the given vector A. It decreases as the algorithm executes, and the least possible value is the vector with all 1's.

18.3 Parallel Greatest Common Divisor (GCD) Algorithm: Second Algorithm

Let A be an array of n natural numbers. Our goal is to find the GCD of these numbers. When all elements of A are divided by the GCD, we get a quotient vector which is also a vector of natural numbers. Finding GCD of A is equivalent to finding the minimum vector G such that there exists a number d such that for all i, $A[i] = d * G[i]$. The problem can be formulated as finding minimum G such that $\forall i : G[i] = A[i]/d$. Equivalently, our goal is to find minimum G such that $B_{gcd}(G) \equiv \forall i, j : A[i]/G[i] = A[j]/G[j]$, This feasibility predicate, B_{gcd}, is equivalent to $\forall j : G[j] \geq \max_i\{A[j]/A[i] * G[i]\}$. Since the right hand side is a monotone function, we know that the predicate is lattice-linear. We can apply LLP algorithm with $forbidden(G, j)$ as $G[j] < \max_i\{A[j]/A[i] * G[i]\}$ and $\alpha(G, j) = \lceil \max_i\{A[j]/A[i] * G[i]\} \rceil$.

We now give a parallel algorithm to compute gcd of an array of numbers based on this idea.

```
var
  G: array[1..n] of int initially ∀i : G[i] = 1;
      for all j such that G[j] < max_i{A[j]/A[i] * G[i]};
          G[j] := ⌈max_i{A[j]/A[i] * G[i]}⌉;
      endfor;
  return A[1]/G[1];
```

Figure 18.2: **Algorithm GCD** to find the greatest common divisor of a set of numbers

The above algorithm was mechanically derived from LLP algorithm. Let us see how it relates to Euclid's algorithm to compute gcd of two numbers: $A[1]$ and $A[2]$.

Suppose that $A[1]$ and $A[2]$ are 10 and 15 respectively. We get the following steps. Initially, $G[1] = 1$ and $G[2] = 1$. Since $G[2] * A[1] < G[1] * A[2]$, we update $G[2]$ to 2, getting $G = [1, 2]$. Now $G[1] * A[2] < G[2] * A[1]$, we get $G = [2, 2]$. Now $G[2] * A[1] < G[1] * A[2]$, we update $G[2]$ to $\lceil A[2]/(A[1] * G[1]\rceil = 15/10 * 2 = 3$. We now have $G = [2, 3]$ and the algorithm terminates. It returns $A[1]/G[1]$ which equals 5.

```
var
  G: int initially ∀i : G[i] = 1;
      while(A[1]/G[2] ≠ A[2]/G[1]) do
          if G[1] * A[2] < G[2] * A[1] then
              G[1] := ⌈A[1]/A[2] * G[2]⌉;
          else
              G[2] := ⌈A[2]/A[1] * G[1]⌉;
      endwhile;
  return A[1]/G[1];
```

Figure 18.3: **Algorithm GCD** to find the greatest common divisor of a set of numbers

18.4 Chinese Remainder Theorem

We have r coprime numbers $m_1, m_2, \ldots m_r$ and r integers b_i such that $0 \leq b_i < m_i$. Our goal is to find the least number x such that $x \equiv b_i (mod\ m_i)$. There are r equations. We think of this as r processes such that process i is solving equation i. Furthermore, we want each process to get the same value of x. Initially, we start with $G[i]$ equal to $b[i]$ for all i. We are searching for the least G such that

$$B \equiv (\forall i, j : G[i] \leq G[j]) \wedge (\forall i : G[i] \equiv b_i (mod\ m_i))$$

Since each process P_i can only increase the value of $G[i]$, the predicate $(G[i] \leq G[j])$ is lattice-linear. Furthermore, $(G_i \equiv b_i (mod\ m_i))$ is a local predicate and therefore also lattice-linear. Thus, B is a lattice-linear predicate and we can use LLP algorithm to solve the problem. The simplest algorithm will advance on the process P_i by 1 if either $G[i] < G[j]$ for some j or if $\neg(G[i] \equiv b_i (mod\ m_i))$. Furthermore, the only way the second part would become true is if $G[i]$ is advanced to a state in which $G[i] mod\ m_i$ equals b_i. To simplify matters further, we initialize $G[i]$ to b_i since that is the smallest state in P_i satisfying the local predicate. We maintain the invariant that $\forall i : G[i] = b[i] mod\ m[i]$. With this observation, we get the Figure Par-CRT.

Algorithm Par-CRT: A Parallel Algorithm for Chinese Remainder Problem

```
1 input: m : array[1..n] of int;
2 b : array[1..n] of int;
3 var: G: array[1..n] of int ;
4 init: ∀j : G[j] := b[i];
5 forbidden(j): (∃i : G[j] < G[i]) ∨ (G[j] mod m[j] ≠ b[j])
6 advance(j):
7       G[j] := G[j] + min_k{G[j] + k * m[j] ≥ G[i]};
```

Observe that process P_i does not need access to any $b[j]$ or $m[j]$ for $j \neq i$. It does need access to $G[j]$. Hence, this algorithm can be applied in a distributed fashion if G is available publicly even though b and m are private. Applying the Chinese Remainder Theorem to solve this problem requires public access of m.

So far we have not shown that the algorithm terminates but we know that if the algorithm terminates it gives us the smallest number that satisfies B. For this particular example, we do know that the program terminates due to Chinese Remainder Theorem. However, if instead of linear equations we had arbitrary equations, then one could still apply this program except that the program may not terminate.

Instead of searching for the solution in increasing order, one can also search it in the decreasing order provided we know where to start the search from. In the above example, let M be the product of all $m[i]$, i.e., $M = m[1]m[2]...m[n]$.

Then, from Chinese Remainder Theorem we know that the solution exists between $0..M-1$. It is easy to see that B is also dual lattice-linear. Hence, one can use the following algorithm to find the largest solution that is at most M.

Algorithm Par-CRT2: A Parallel Algorithm for Chinese Remainder Problem

```
1 input: m : array[1..n] of int;
2 b : array[1..n] of int;
3 var: G: array[1..n] of int ;
4 init: ∀j : G[j] := greatest number at most M − 1 that is congruent to b[j] mod m[j];
5 forbidden(j): (∃i : G[j] > G[i]) ∨ (G[j]mod m[j] ≠ b[j])
6 advance(j):
7       G[j] := G[j] − min_k{G[j] − k ∗ m[j] ≤ G[i]};
```

18.5 Viewing Numbers as Distributive Lattice

The set of all natural numbers forms a distributive lattice under the divides relation. This distributive lattice (infinite) has the following representation. For every prime p, we have a chain of elements $p^0, p^1, p^2, \ldots$. This infinite poset has as many chains as the number of distinct primes. There is a 1-1 correspondence between the lattice of natural numbers and the cross product of all chains in the poset (by the unique factorization of numbers) We can go from any element n in the number lattice to an element in the croos-product by choosing $k+1$ elements from the chain for the prime p if k is the largest integer such that p^k divides n.

Let us define some arithmetical functions on the set of natural numbers. We are interested in functions that can be defined on the poset representation of natural numbers, i.e., the function is defined only for the prime powers. The value of the function on any number is simply the product of its value on each of the prime powers.

Definition 18.2 (Multiplicative Functions) *We call an arithmetical function f* multiplicative *if it is not a zero function and $f(mn)$ equals $f(m)f(n)$ whenever $gcd(m,n)$ equals 1.*

Note that we require $f(mn)$ equals $f(m)f(n)$ only when $gcd(m,n)$ equals 1. If we drop the requirement of gcd to be 1, then the function is called *completely* multiplicative.

We now introduce some multiplicative functions.

Definition 18.3 (Mobius function) *The Mobius function μ is defined on any prime power p^k as 1 if $k = 0$, -1 if $k = -1$ and 0, otherwise.*

Since μ is defined to be multiplicative it follows that if $n = p_1^{a_1} p_2^{a_2}, \ldots p_m^{a_k}$. Then, $\mu(n)$ equals $(-1)^k$ if $\forall i \in [k]$: $a_i = 1$ and $\mu(n)$ equals 0, otherwise.

We now show a simple formula for the divisor sum of $\mu(n)$.

Theorem 18.4 *For any $n > 1$, $\sum_{d|n} \mu(d) = 0$.*

Proof: Let $n = p_1^{a_1} p_2^{a_2}, \ldots p_k^{a_k}$. It is easy to see that d divides n iff $d = p_1^{b_1} p_2^{b_2}, \ldots p_m^{b_k}$, where for all $i \in [m] : b_i \leq a_i$. Consider the poset $P(n)$. It is easy to verify that $\sum_{d|n} \mu(d)$ is the sum of all cross-products of chains in $P(n)$. This sum is also equal to the product of sum of $\mu(d)$ where d is a prime power. However, $\sum_{d_i|n, d \text{ is a prime power of } p_i} \mu(d)$ is zero because for any prime p that divides n, $\mu(p^0) = 1, \mu(p^1) = -1$, and $\mu(p^k) = 0$ for $k \geq 2$. Hence, $\sum_{d|n} \mu(d) = 0$.

■

We now show another multiplicative arithmetical function defined on the set of natural numbers.

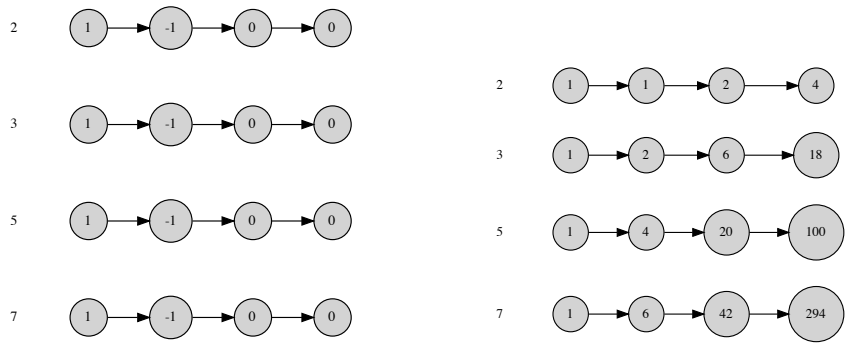


Figure 18.4: Poset representations for μ and ϕ. The value for $\phi(p^k)$ is $p^k - p^{k-1}$. Thus, the third entry for prime 3 is $3^2 - 3^1 = 6$.

Definition 18.5 (Euler's Totient function) *The Euler's totient function ϕ is defined on any prime power p^k for $k > 0$ as $p^k - p^{k-1}$. Also, $\phi(1)$ is defined as 1.*

For example, $\phi(3^2) = 3^2 - 3^1 = 9 - 3 = 6$. The function ϕ can be extended to any natural number n by using the multiplicative property. We now show that

Lemma 18.6 *For any $n > 1$, $\phi(n)$ equals the number of positive integers less than or equal to n which are relatively prime to n.*

Proof: Let $n = p_1^{a_1} p_2^{a_2}, \ldots p_k^{a_k}$ for some $k \geq 1$. When k equals 1, $n = p_1^{a_1}$. There are $p_1^{a_1}$ positive integers less than or equal to n. Out of these, $p_1^{a_1 - 1}$ are less than n and divide n. Therefore, $\phi(n)$ equals the number of positive integers less than or equal to n which are relatively prime to n. Let A_i denote the numbers that are relatively prime to $p_i^{a_i}$. Now, suppose that $k > 1$. From the multiplicative property, $\phi(n) = \phi(p_1^{a_1})\phi(p_2^{a_2}), \ldots \phi(p_k^{a_k})$. It can be easily verified that $\phi(n)$ consists of numbers relatively prime to n.

■

We now show that

Theorem 18.7 *For any $n > 1$, $\sum_{d|n} \phi(d) = n$.*

Proof: Let $n = p_1^{a_1} p_2^{a_2}, \ldots p_k^{a_k}$ for some $k \geq 1$. The poset representation of $\phi(n)$ is shown in Figure. Clearly, $\sum_{d|n} \phi(d) = \Pi_{i=1}^{k} \sum_{j=0}^{j=k} \phi(p_i^j)$. Since, $\phi(p_i^j) = p^j - p^{j-1}$, by the telescoping argument, we get $\sum_{j=0}^{j=k} \phi(p_i^j) = p_i^k$. Hence, $\sum_{d|n} \phi(d) = \Pi_{i=1}^{k} p_i^k = n$.

■

Some additional multiplicative functions are defined below.

1. The arithmetical function N is defined as $N(p^k) = p^k$.
2. The arithmetical function *Identity*, I is defined as $I(p^k) = 1$ if $k = 0$ and 0 otherwise.
3. The arithmetical function *Unit*, u is defined as $u(p^k) = 1$ for all p, k.

4. The arithmetical function λ is defined as $\lambda(p^k) = (-1)^k$ for all k.

The following Lemma gives the summation results for these functions.

Lemma 18.8 *For any $n > 1$,*
(a) $\sum_{d|n} N(d) = \Pi_{p^k|n}(p^{k+1} - 1)/(p - 1)$.
(b) $\sum_{d|n} I(d) = 1$.
(c) $\sum_{d|n} u(d) =$ the number of divisors of n.
(d) $\sum_{d|n} \lambda(d) = 1$ if n is a square and 0, otherwise.

Proof: Since all these functions are multiplicative, we will simply compute them for $n = p^k$.
(a) $\sum_{d|p^k} N(d) = (p^{k+1} - 1)/(p - 1)$ since we have a geometric series. From multiplicative property, the results follows.
(b) $\sum_{d|p^k} I(d) = 1$ since only $I(1) = 1$. It is 0 for all $k > 1$. By multiplicative property, we get that $\sum_{d|n} I(n) = 1$.
(c) $\sum_{d|p^k} u(d) = k + 1$. By invoking the multiplicative property, we get that $\sum_{d|n} u(d) =$ the number of divisors of n.
(d) $\sum_{d|p^k} \lambda(d) = 1$ if k is even and 0 otherwise. Therefore, $\sum_{d|n} \lambda(d) = 1$ if for each prime the corresponding exponent is even and 0 otherwise. Thus, $\sum_{d|n} \lambda(d) = 1$ if n is a square and 0, otherwise.

■

Chapter 19

Linear Programming

19.1 Introduction

In this chapter, we consider linear programming that is a general technique to solve many problems.

We note here that linear programming has also served as a unifying tool for many combinatorial optimization problems. In particular, the man-optimal stable marriage problem, the shortest path problem, and the least market clearing price can all be modeled as integer linear programs with total unimodularity. However, it is not known whether linear programming is strongly polynomial. In contrast, our method results in algorithms that are close to best-known strongly polynomial algorithms for all these problems. There are other differences in these two approaches. Algorithms for linear programming typically stay inside the feasible space and improve the objective function with every iteration. Our approach keeps the objective function as extremal as possible while making progress towards a feasible solution. Linear programming method uses real variables with linear objective function and linear constraints. Our method assumes that the search is for the infimum element of a feasible space in a distributive lattice such that the feasible space satisfies a lattice-linear predicate.

Linear constraints for linear programming are defined using conjunction of half-spaces. Lattice-Linear predicates are incomparable to half-spaces. There are half-spaces that are not lattice-linear and there are lattice-linear predicates that are not half-spaces.

Linear programming is a every general technique to solve combinatorial optimization problems. Many optimization problems can be modeled using linear or integer linear programs.

Suppose that there is an objective function which we are trying to minimize,

$7x_1 + x_2 + 5x_3$

subject to constraints,

x_1 - $x_2 + 3x_3 \geq 10$

$5x_1 + 2x_2$ - $x_3 \geq 6$

where x_1, x_2, x_3 are the decision variables and assumption is that,

$x_1, x_2, x_3 \geq 0$

Suppose we have solution, $x = (2, 1, 3)$. On substituting in the objective function, we get the objective calue to be $14 + 1 + 15 = 30$. The reader can verify that this solution is also feasible as it satisfies the constraints,

2 - 1 + 3(3) = 10

5(2) + 2(1) - 3 = 9

Trying to find a bound on primal:

$7x_1 + x_2 + 5x_3 \geq x_1$ - $x_2 + 3x_3 \geq 10$ by comparing coefficients point-wise and $x_i \geq 0$

solution has to be ≥ 10 but we are trying to get a better or greater lower bound

Add up the constraints equations,

$6x_1 + x_2 + 2x_3 \geq 16$

The objective function is still greater than the above equation. Hence new lower bound is 16.

The goal of primal was to minimize and so now the goal of dual will be to maximize. We have to figure out what combination of each constraint should we add such that it gives the tightest bound on the objective function.

We have n decision variables and m constraints in our primal. So dual will have m variables (one variable for each constraint in primal) and n constraints (since n decision variables in primal).

Objective function is $10y_1 + 6y_2$ subject to constraints,

$y_1 + 5y_2 \leq 7$

$-y_1 + 2y_2 \leq 1$

$3y_1 - y_2 \leq 5$

In a generalized representation of the primal, objective function can be written as

$$\sum_{i=1}^{n} c_i x_i$$

constraints can be written as

$$\sum_{j} A_{ij} x_j \geq b_j$$

where A is a matrix of dimension m*n, c and b are vectors of size n and m respectively.

For dual, b is translated to c, c is translated to b and matrix A becomes A^T.

19.2 Strong Duality

Theorem 19.1 *The primal program has finite optimum if and only if its dual has finite optimum. Moreover, if $x*$ and $y*$ are optimal solutions for (P) and (D) then,*

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*$$

This theorem translates into all many min-max theorems such as max flow = min cut, max matching = min vertex cover and min number of chains to cover a poset = max sized anti-chain.

19.3 Weak Duality

Theorem 19.2 *If x and y are feasible solutions for (P) and (D) then,*

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{i=1}^{m} b_i y_i$$

Proof:

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{j=1}^{n} (\sum_{i=1}^{m} a_{ij} y_i) x_j$$

This claim is true because y is feasible and $x \geq 0$

$$\sum_{i=1}^{m} (\sum_{j=1}^{n} a_{ij} x_j) y_i \geq \sum_{i=1}^{m} b_i y_i$$

This claim is true because x is feasible and $y \geq 0$

Hence proved,

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{i=1}^{m} b_i y_i$$

■

19.4 Complementary Slackness Conditions

Theorem 19.3 *Given that x,y are (P) and (D) feasible solutions. Then x and y are both optimal if and only if the following complementary slackness conditions are satisfied:*

$$\forall j \quad either(x_j = 0) \quad or \quad \sum_{i=1}^{m} a_{ij} y_i = c_j$$

$$\forall i \quad either(y_i = 0) \quad or \quad \sum_{j=1}^{n} a_{ij} x_j = b_i$$

19.5 Maximum Matching Problem As Linear Program

Primal:

$$max \sum_{i,j} W_{ij} x_{ij}$$

subject to:

$$\sum_{j} x_{ij} \leq 1$$

and

$$\sum_{i} x_{ij} \leq 1$$

where x_{ij} is an edge between A and B, i is an index from A and j is an index from B. For each edge, there is a weight W_{ij}. We need to pick the edges such that the sum of their weights is maximum.

Replacing integral constraint by linear constraint,

$$x_{ij} \in \{0, 1\} \Rightarrow x_{ij} \geq 0$$

The above transformation is called LP-Relaxation. This can be used only when we can prove that the optimal solution is always going to be integral.

When $|A| = |B| = n$

Number of decision variables = n^2

Number of constraints = 2n

Dual:

Number of decision variables = 2n

Number of constraints = n^2

Objective function (minimizing vertex labels):

$$min \sum u_i + \sum v_j$$

subject to: $u_i + v_j \geq w_{ij}$

$u_i \geq 0 \quad and \quad v_j \geq 0$

where u_i is constraints for side A and v_j is constraints for side B

19.6 The Transportation Problem

We have n supply nodes and n demand nodes. The supply at node i is given by a_i and the demand at node j is given be b_j. We assume that the total supply is equal to the total demand. If the supply and demand does not match, we can create fake nodes to ensure the equality.

Let w_{ij} be the cost function when an item is shipped from i to node j. Let the variable x_{ij} be the number of items i shipped from the source node i to the demand node j. Our goal is to choose x_{ij} such that the cost of transportation is minimized. $\sum_{i,j} w_{ij}x_{ij}$ is maximized subject to:
$\sum_j x_{ij} = a_i$ for all i
$\sum_i x_{ij} = b_j$ for all j
and $x_{ij} \geq 0$.

The dual of the relaxed linear program is:

maximize $\sum_i a_i u_i + \sum_j b_j v_j$ subject to
$u_i + v_j \leq w_{ij}$ for all i, j,
$u_i \geq 0$ for all i,
$w_j \geq 0$ for all j.

19.7 The Stable Marriage Problem

Let $x_{i,j}$ denote that man i is matched to woman j. We require that $x_{ij} \in \{0, 1\}$. The constraint that every man and every woman is matched to a single partner is given as follows. For each man i,

$$\sum_j x_{ij} = 1.$$

Similarly, for each woman j,

$$\sum_i x_{ij} = 1.$$

Furthermore, we would like this assignment to not have any blocking pair. For any man i and woman j, they do not form a blocking pair

This condition can be expressed as follows. For any man i and woman j, if j is matched to a man who is less preferred than i, then man i must be matched to a woman who he prefers to i. Otherwise, (i, j) will form a blocking pair for the matching. Let $Q(j, i)$ be the set of men who are less preferable than i to woman j, i.e.,

$$Q(j, i) = \{i' \mid wrank[j][i'] > rank[j][i]\}$$

Then, the expression $\sum_{i' \in Q(j,i)} x_{i',j}$ is one iff j is matched to someone who is less preferable than i. Similarly, let $R(i, j)$ be the set of women who are more preferable than woman j to man i. Then, the expression $\sum_{j' \in R(i,j)} x_{i,j'}$ is one iff i is matched to someone who is more preferable than woman j. Given these two expressions, we can write the condition that (i, j) is not a blocking pair as

$$\sum_{i' \in Q(j,i)} x_{i',j} - \sum_{j' \in R(i,j)} x_{i,j'} \leq 0.$$

This we have $O(n^2)$ constraints.

This constraint can be modeled as follows.

19.8 The Shortest Path Problem

Let x_i denote the distance from the source vertex x_0 to x_i. Let t be the destination vertex. We assume that there are no incoming edges to x_0 or outgoing edges from x_t.

We would like to maximize $x_t - x_0$ such that for all edge $(i, j) \in E$: $x_j \leq x_i + w[i, j]$. Hence, the LP formulation is as follows.

$$\begin{aligned} \text{maximize} \quad & x_t - x_s \\ \text{subject to} \quad & x_j \leq x_i + w[i,j] \quad \forall (i,j) \in E, \\ & x_i \geq 0 \quad \forall i \end{aligned} \tag{19.1}$$

This formulation is identical to that we used in designing an LLP based algorithm (or equivalently, Bellman-Ford algorithm). We assume that there is no incoming edge to x_0. In this linear program, there are n variables, one for each node, and there are m constraints, one for each edge. The variables x_i are nonnegative.

The dual of this linear program is as follows.

We have m variables, $y_{i,j}$ for each edge $(v_i, v_j) \in E$. The variable $y_{i,j}$ can be viewed as the flow on the edge (v_i, v_j).

We would like to minimize $\sum_{(i,j)\in E} w[i, j] * y[i, j]$ as follows.

minimize $\sum_{(i,j)\in E} w[i, j] * y[i, j]$

subject to
- for each node v_i other than v_0 and v_t: $\sum_j y_{j,i} - \sum_k y_{i,k} = 0$.
- for v_t, $\sum_j y_{j,t} = 1$.
- for v_0, -$\sum_j y_{0,j} = -1$.
- for all i, j: $y[i, j] \geq 0$.

Linear Programming Formulation of the Shortest Path Problem

Given a directed graph $G = (V, E)$ with vertices V and edges E, and a cost c_{ij} associated with each edge $(i, j) \in E$. The goal is to find the shortest path from a source vertex s to a destination vertex t.

Primal Problem

Variables:

- Let x_{ij} be a binary variable for each edge $(i, j) \in E$, indicating whether the edge is part of the shortest path.

Objective Function:

$$\text{Minimize} \sum_{(i,j)\in E} c_{ij} x_{ij}$$

Constraints:

- For each vertex $v \in V \setminus \{s, t\}$:

$$\sum_{(i,v)\in E} x_{iv} - \sum_{(v,j)\in E} x_{vj} = 0$$

- For the source vertex s:

$$\sum_{(s,j)\in E} x_{sj} = 1$$

- For the destination vertex t:

$$\sum_{(i,t)\in E} x_{it} = 1$$

- For all x_{ij}:

$$x_{ij} \geq 0$$

Dual Problem

Variables:

- Let y_v be a dual variable for each vertex $v \in V$.

Objective Function:

$$\text{Maximize } y_t - y_s$$

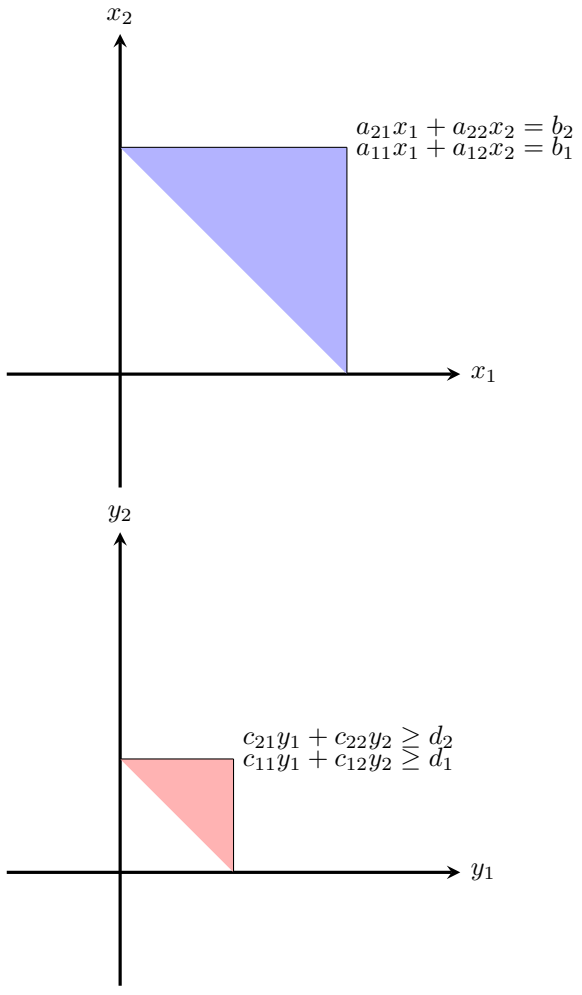
Constraints:

- For each edge $(i, j) \in E$:

$$y_i - y_j \leq c_{ij}$$

- The dual variables y_v are unrestricted in sign.

The primal problem minimizes the total cost of the path from the source to the destination. The dual problem can be seen as assigning potentials to the vertices such that the potential difference across each edge does not exceed its cost. The objective is to maximize the potential difference between the destination and the source.



19.9 Comparison of Linear Programming with Lattice-Linear Predicates

Linear programming can also be viewed as a search for an optimal feasible solution. However, there are many important differences.

- **Vector Space vs Distributive Lattice**: First, the underlying space in linear programming is the set of real valued vectors whereas the underlying space in the lattice-linear predicate detection method is a distributive lattice.

  In the domain of distributive lattices, we do not have addition or the scalar multiplication as in vector spaces. All of lattice-linear predicate algorithms use the following two operations: meet of the underlying lattice and the "advance" operation. The advance operation maps an element of the lattice to a bigger element in the lattice.

- **Polyhedron vs Meet-Closed Predicates**: In linear programming, the feasible space is characterized by a polyhedron (or the set of vectors x such that $Ax \leq 0$ for some matrix A). There is no lattice structure required on the feasible space. It is not guaranteed that if two vectors are feasible, then their component-wise minimum vector is also feasible. Lattice-linear predicate detection requires the feasible space to be closed under meets.

  Linear programming has its optimization objective as minimization or maximization of a linear cost function. Lattice linear predicate detection simply uses the underlying order operation to define optimization. Since feasible space is closed under meets, the infimum of the feasible space is well-defined.

- **Efficiency of the General Algorithm**: Even though many problems studied in this book can also be solved via linear programming, the algorithms derived in that manner are not as efficient as the LLP algorithm. The following list gives the algorithms that are used for Linear Programming.

  1. **Simplex Method:** Developed by George Dantzig, it is widely used for solving linear programming problems by moving from one vertex of the feasible region to an adjacent one with a higher objective value until the optimum is reached. The *dual Simplex method* is a variant of the simplex method, used when the initial solution is not feasible. It modifies the constraints to move towards feasibility and optimality simultaneously. The worst-case time complexity is exponential, but in practice, it is often efficient.
  2. **Interior Point Methods:** These methods, including Karmarkar's algorithm, approach the optimal solution from within the feasible region. They can be faster than the simplex method for large-scale problems.
  3. **Ellipsoid Method:** Introduced by Shor and later improved by Khachiyan, this was the first polynomial-time algorithm for linear programming. It's more of theoretical interest as it's outperformed by other methods in practice.
  4. **Cutting Plane Method:** Adds additional linear constraints (cuts) to the problem to exclude regions that do not contain the optimal solution, gradually cutting closer to the optimal point.
  5. **Branch and Bound Method:** Often used for mixed-integer linear programming, this method explores branches of possible solutions and bounds them to identify the optimal solution.

- **Parallelism of the General Algorithm**: The LLP algorithm determines which component in G are forbidden and advances on all of those components in parallel. The worst case complexity is given be the height of the lattice.

19.10 Modeling Linear Programming Using Lattices

Suppose that we have m linear constraints. Our goal is to find a point that satisfies all linear constraints and minimizes a linear objective function. We assume that the constraints are in n dimensional space.

We model it as a lattice linear program.

Chapter 20

The Predicate Detection Problem

20.1 Introduction

Debugging and testing multithreaded software is widely acknowledged to be a hard task. Sometimes it takes a programmer days to locate a single bug, especially when the bug appears in one thread schedule but not in others. The current debugging and testing method for multithreaded programs is as follows. The programmer tries the program with multiple inputs in the hope of finding a faulty execution. However, the behavior of a multithreaded program depends not only on the external user input, but also the thread schedule and the order in which locks are obtained by the program. It is easy for the testing process to miss a bug that arises with an alternate schedule. One of the fundamental problems in debugging these systems is to check if the user-specified condition exists in *any* global state of the system that can be reached by an alternative thread schedule. This problem, called *predicate detection*, takes a concurrent computation (in an online or offline fashion) and a condition that denotes a bug (for example, violation of a safety constraint), and outputs a schedule of threads that exhibits the bug if possible. Predicate detection is predictive because it generates inferred reachable global states from the computation; an inferred reachable global state might not be observed during the execution of the program, but is possible if the program is executed in a different thread interleaving.

20.2 Detecting Conjunctive Predicates

In this section we describe two parallel algorithms to detect a conjunctive predicate $B = l_1 \wedge l_2 \wedge \cdots \wedge l_n$. To detect B, we need to determine if there exists a consistent global state G such that B is true in G. Note that given a computation on n processes each with m states, there can be as many as m^n possible consistent global states. Therefore, enumerating and checking the condition B for all consistent global states is not feasible. Since B is conjunctive, it is easy to show [GW92] that B is true iff there exists a set of states $s_1, s_2, ..., s_n$ such that (1) for all i, s_i is a state on P_i, (2) for all i, l_i is true on s_i and (3) for all i, j: $s_i \| s_j$. Our detection algorithm will either output such local states or guarantee that it is not possible to find them in the computation. When the global predicate B is true, there may be multiple G such that B holds in G. We are interested in algorithms that return the minimum G that satisfies B. The minimum G corresponds to the smallest counter-example to a programmer's understanding becuase B typically represents the violation of a safety constraint.

Our parallel algorithm is based on the setting where the execution traces for all processes have been collected at one process. For example, in the centralized algorithm for conjunctive predicate detection, one process serves as a checker and collects the traces. All other processes involved in detecting the conjunctive predicate, referred to as application processes, check for local predicates during the computation. Each process P_i also maintains the vector clock algorithm. Whenever the local predicate of a process becomes true for the *first* time since the most recently

sent message (or the beginning of the trace), it generates a debug message containing its local timestamp vector and sends it to the checker process [GW92].

The checker process uses queues of incoming messages to hold incoming local snapshots from application processes. We require that messages from an individual process be received in FIFO order. If the underlying system is non-FIFO, then sequence numbers can be attached with messages to ensure FIFO delivery. At the end of the computation, the checker process has a sequence of local states from each process where its local predicate is true. We now describe a sequential and a parallel algorithm to detect B on these traces. The sequential algorithm is an adaptation of the algorithm from [GW92]. We include it here because it is instrumental in understanding the parallel algorithm. Moreover, the correctness of the parallel algorithm is shown by assuming the correctness of the sequential algorithm.

20.3 A Work-Efficient Parallel Algorithm

The algorithm in Fig. 20.2 takes as input n traces each of size m as shown in Fig. 20.1. Since conjunctive predicates are lattice-linear, we simply use LLP algorithm to detect them. We use G for the consistent global state. A local state $G[j]$ is forbidden if it is less than $G[i]$ for some i. Since we are interested only in global states where the local predicate is true for all $G[i]$, we assume that for any i, we only consider $G[i]$ such that the local predicate is true in $G[i]$.

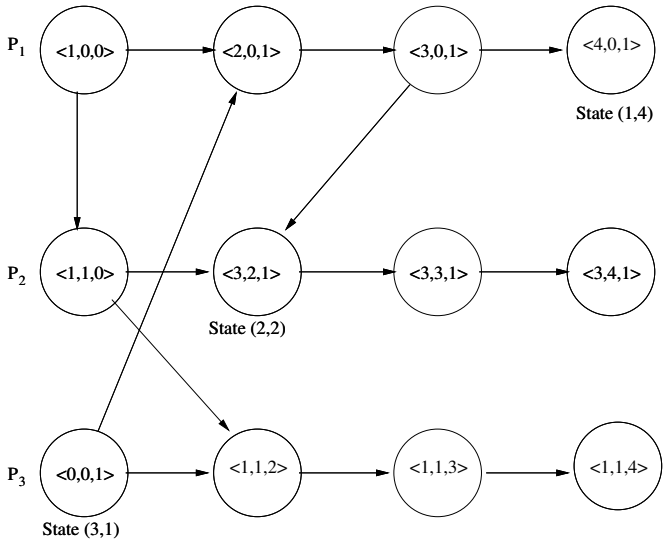


Figure 20.1: State-Based Model of a Distributed Computation

20.4 An NC Algorithm for Conjunctive Predicate Detection

We now give another parallel algorithm that has lower time complexity but higher work complexity. Our approach is based on computing a *state rejection graph* for the trace. The state rejection graph is a directed graph with all local states as vertices of the graph. Let state j on process i be denoted as (i, j). The state rejection graph puts a rejection edge from the state (i, j) to (i', j') if the rejection of state (i, j) as a possible component of the consistent cut implies that the state (i', j') will also be rejected.

In Fig. 20.4 we show the state rejection graph of the computation in Fig. 20.1. The first state in P_1 given by the vector clock $\langle 1, 0, 0\rangle$ will be rejected by the sequential algorithm because it happened before $\langle 1, 1, 0\rangle$. The sequential algorithm will then move P_1 to the second state $\langle 2, 0, 1\rangle$ (successor of the first state in P_1). However, this would

```
function ConjunctiveAlgorithm()
var
    G: array[1..n] of int initially 1;
    T = (m_1, m_2, ..., m_n); //maximum number of proposals at P_i
always
    forbidden(G, j, B) ≡ ∃i : G[j] → G[i]
while ∃j : forbidden(G, j, B) do
    for all j such that forbidden(G, j, B) in parallel:
        if (G[j] = T[j]) then return null;
        else G[j] := G[j] + 1;
endwhile;
return G; // satisfying global state
```

Figure 20.2: Conjunctive Predicate Detection Algorithm.

imply that the state $\langle 0,0,1\rangle$ will be rejected because $\langle 0,0,1\rangle$ happened before $\langle 2,0,1\rangle$. Hence, there is a rejection edge from $\langle 1,0,0\rangle$ to $\langle 0,0,1\rangle$. Similarly, the rejection of $\langle 0,0,1\rangle$ implies that P_3 will move to $\langle 1,1,2\rangle$. However, that move will result in rejection of the state $\langle 1,1,0\rangle$. Therefore, we put a rejection edge from $\langle 0,0,1\rangle$ to $\langle 1,1,0\rangle$. Finally, the rejection of $\langle 1,1,0\rangle$ will result in P_2 moving to $\langle 3,2,1\rangle$ which will result in the rejection of $\langle 2,0,1\rangle$ and $\langle 3,0,1\rangle$. All the rejection edges are shown in dashed arrows in Fig. 20.4. We show how such a graph can be constructed efficiently in parallel. The next step in the algorithm is to compute the transitive closure of this graph. Finally, the algorithm determines the least local state at each process which has not been rejected. In this example, it is the fourth state on P_1, the second state on P_2 and the third state on P_3.

Our parallel algorithm is presented in Fig. 20.3. The input to the algorithm is the same as that of the LLP algorithm: a two-dimensional array of vector clocks so that we can determine the happened-before order between states.

We now explain the steps of the algorithm.

Step 1: We first create F, the set of all initially rejected states. Let I be the global state consisting of each processor's first local state, i.e., $I = \{(i,1) \mid i \in 1..n\}$. If there are no dependencies between any of these states, we have already reached the first consistent global state. Else, if there is a dependency from one of these states to another, we reject whichever state happened-before the other and add it to F. We represent the set F by a boolean bit array of size n that is indexed by processor. F is initially empty. Then, we set $F[i]$ to 1 whenever there exists a state $(j,1)$ such that $(j,1) \rightarrow (i,1)$.

This step can be done in $O(1)$ time in parallel with $O(n^2)$ work by using a separate processor for each value of i and j.

Step 2: In step two, we create the state rejection graph represented as an adjacency matrix. We define a new two-dimensional array called R, which is of size $mn \times mn$, where each row and each column represents a different state. In this directed graph, there is an edge from state (i,j) to another state (i',j') only if we know that once state (i,j) is rejected, state (i',j') will also be rejected. In the adjacency matrix R, this is represented as $R[(i,j),(i',j')] = 1$. Additionally, we make the diagonal of the matrix all 1's. We show that creating this boolean matrix can be done in constant time. First, setting the diagonal to all 1's in R takes constant time. We now discuss how off-diagonal entries are set. This is a crucial step in our algorithm. Suppose that a state (i,j) is rejected. We know that the processor will advance to the next state. Then, the next choice for that processor is $(i,j+1)$. Thus, the rejection of (i,j) would lead to the rejection of all states (i',j') where $(i',j') \rightarrow (i,j+1)$. Formally,

$$R[(i,j),(i',j')] = 1 \equiv (i',j') \rightarrow (i,j+1)$$

By using a separate processor for each tuple (i,j,i',j'), we can set R in $O(1)$ time and $O(m^2n^2)$ work. We

```
function ParallelCut()
Input: states : array[1...n][1...m] of vectorClock // Sequence of local states at each process
Output: Consistent Global State as array cut[1...n]

Step 1: Create F: set of states rejected in the first round
 var F : array[1...n] of 0...1 initially 0;
 for all (i ∈ 1...n, j ∈ 1...n) in parallel do
        if ((i,1) → (j,1)) then
           F[i] := 1;

Step 2: Create R: State Rejection Graph // Represented as an Adjacency Matrix
 var R : [(1...n,1...m),(1...n,1...m)] of 0...1;
 for all (i ∈ 1...n, j ∈ 1...m) in parallel do
    R[(i,j),(i,j)] := 1;
 for all (i ∈ 1...n, j ∈ 1...m−1, i' ∈ 1...n, j' ∈ 1...m)
        such that i ≠ i' in parallel do
        if ((i',j') → (i,j+1)) then
           R[(i,j),(i',j')] := 1;
        else
           R[(i,j),(i',j')] := 0;

Step 3: Create RT: transitive closure of R
 var RT : array[(1...n,1...m),(1...n,1...m)] of 0...1;
 RT := TransitiveClosure(R);

Step 4: Create valid: replace invalid states by 0
 var valid : array[[1...n][1...m] of 0...1;
 for all (i ∈ 1...n, j ∈ 1...m) in parallel do
    valid[i][j] := 1;
 for all (i ∈ 1...n, i' ∈ 1...n, j' ∈ 1...m) in parallel do
    if (F[i] = 1) ∧ (RT[(i,1),(i',j')] = 1) then
        valid[i'][j'] := 0;


Step 5: Create cut: First Consistent Global State
 var cut : array[1...n] of 0...m initially 0;
 for all (i ∈ 1...n, j ∈ 1...m) in parallel do
        if (valid[i][j] ≠ 0) then
           if (j = 1) ∨ ((j > 1) ∧ (valid[i][j−1] = 0) then
              cut[i] := j;
 for all (i ∈ 1...n) in parallel do
        if (cut[i] = 0) then
           output("No satisfying Consistent Cut");

 return ConsistentCut := cut;
```

Figure 20.3: The ParallelCut algorithm to find the first consistent cut.

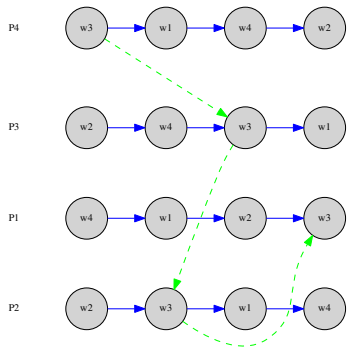


Figure 20.4: State Rejection Graph of a computation shown in dashed arrows

represent the state rejection graph as a boolean matrix so that we can compute the transitive closure of this graph by doing matrix multiplications.

Step 3: In step three, we take the transitive closure of R. The transitive closure of R is also represented as an adjacency matrix RT. It is well known that the transitive closure for any directed graph with $|V|$ vertices can be computed in $O(\log |V|)$ time using $O(|V|^3 \log |V|)$ work on the common CRCW PRAM [JáJ92]. Since our graph has $O(mn)$ vertices, this step takes $O(\log mn)$ time using $O(n^3m^3 \log mn)$ operations on the CRCW PRAM. This is the only step in our algorithm that takes more than $O(1)$ time.

Step 4: In step four, we use both F and RT to determine which states will not be part of the first consistent global state. To do this, we create a new two-dimensional array called $valid[1 \ldots n][1 \ldots m]$ where each entry is a value of $0 \ldots 1$. We initialize every entry in $valid$ to 1 in $O(1)$ time with $O(mn)$ work. The algorithm sets rejected states in $valid$ with a 0. We use F and RT for this purpose. For all possible values of i, j, i', and j', in parallel, we check to see if a state (i, j) is an element of F and if there is an edge from (i, j) to another state (i', j'). If the two states (i, j) and (i', j') fit this criteria, then we set $valid[i'][j']$ to 0. In other words, if a state (i, j) is initially rejected, and there is an edge from (i, j) to (i', j') in RT, then we know the state (i', j') will also be rejected.

This step can also be done in $O(1)$ time and $O(m^2n^2)$ work. In our example, we compute the states reachable by $\langle 1, 0, 0\rangle$. In our example, states $\{\langle 0, 0, 1\rangle, \langle 1, 1, 0\rangle, \langle 2, 0, 1\rangle, \langle 3, 0, 1\rangle\}$ are all reachable by $\langle 1, 0, 0\rangle$ by following one or more rejection edges. Thus, we mark the states $(1, 1), (2, 1), (3, 1), (1, 2), (1, 3)$ with $0's$ in $valid$.

Step 5: In step five, we traverse $valid$ and construct the set of states that form the consistent global state in a new array called cut where $cut[i] = j$ signifies that the tuple (i, j) is a part of the consistent global state. To do this, for every process, in parallel, we simply search for the first entry which is nonzero and either it is the first entry or the entry prior to it is zero.

In our example, we can easily see that the first consistent global state is the set $cut = \{(1, 4), (2, 2), (3, 2)\}$. This step can be done in $O(1)$ time and $O(nm)$ work.

Thus, the entire algorithm takes $O(\log mn)$ time and $O(m^3n^3 \log mn)$ work on a common CRCW PRAM.

Remark: Note that the algorithm to detect a conjunctive predicate can be used to detect a global predicate in Disjunctive Normal Form. A predicate is in Disjunctive Normal Form (DNF) if it is expressed as a disjunction of k pure conjunctions. To detect a predicate in this form it is sufficient to detect each conjunction in parallel.

20.5 Conjunctive Predicate at the Given Level

We now show that, in general, asking for a conjunctive predicate on a particular level is **NP**-complete.

Theorem 20.1 *Given a distributed computation, deciding whether there exists a global state with k events satisfying a given conjunctive predicate is* **NP***-complete.*

Proof: We first show that the problem is in **NP**. The global state itself provides a succinct certificate. We can check that all local predicates are true in that global state and that the global state is at level k.

For hardness, we use the subset sum problem. Given a subset problem on n positive integers, $x_1, x_2, \ldots, x_n$ with the requirement to choose a subset that adds up to k, we create a computation on n processes as follows. Each process P_i has x_i events. The local predicate on P_i is true initially and also after it has executed x_i events. Thus, the local predicate is true at each process exactly twice. The problem asks us if there is a global state with k events in which all local predicates are true. Such a global state, if it exists, would choose for every process either the initial local state or the final local state. All the final states that are chosen correspond to the numbers that have been chosen.

To avoid the expansion of the numbers in binary to unary construction, we encode the representation of events on a process as follows: Since the conjunctive predicates can only be true when the local predicates are true, we keep only local states which satisfy their corresponding local predicate and store the number of local events executed so far with them. This leaves two local states at each process: The initial state with zero events executed until that point, and the final state of the process with the number of events equal to x_i for the i^{th} process. Now, the construction of the computation from the subset sum problem is polynomial in the size of the input.

■

20.6 Predicate Detection Problem

In this section, we explore the complexity-theoretic results for predicate detection. It is not surprising that given a parallel program computation and a boolean predicate, it is computationally hard to determine if the execution went through a global state in which the predicate became true. We also show that given a boolean predicate b and an execution, determining whether b is lattice-linear for that execution is co-NP-complete.

Since there are N processes, the total number of global states possible is m^N, where m is the number of state intervals at any process. Consider a boolean predicate B. Even when B is a boolean expression, and processes do not communicate, the problem of detecting *possibly*: B is NP-complete.

We show in this section that the problem of global predicate detection is NP-complete. In fact, we show that it is NP-complete even in the absence of messages between processes.

The global predicate detection problem is a decision problem. It can be written as:

> **Input instance**: a poset S of N sequences, a set of variables X partitioned into N subsets $X_1, \ldots, X_N$, and a predicate B defined on X.
> **Problem:** Determine whether there exists a consistent cut $G \in S$ such that $B(G)$ has the value true.

We now show:

Theorem 20.2 *The global predicate detection problem is NP-complete.*

Proof: First note that the problem is in NP. The verification that the cut is consistent can easily be done in polynomial time (for example, using vector clocks and examining all pairs of states from the cut). Therefore, if the predicate itself can be evaluated in polynomial time, then the detection of that predicate belongs to the set NP.

We show NP-completeness of the simplified predicate detection problem where all program variables are restricted to taking the values "true" or "false", and at most one variable from each X_i can appear in B. We reduce the satisfiability problem of a boolean expression (SAT) to the global predicate detection problem by constructing an appropriate poset.

The poset is constructed as shown in Figure 20.5. For each variable $u_i \in U$, we define a process P_i that hosts variable x_i (i.e., $X_i = \{x_i\}$). Let the sequence S_i consist of exactly two states. In the first state, x_i has the value false. In the second state, x_i has the value true.

It is easily verified that the predicate B is true for some cut in S if and only if the expression is satisfiable.

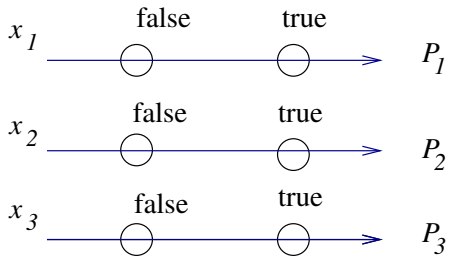


Figure 20.5: Transformation from SAT to global predicate detection

The above result shows that detection of a general global predicate is intractable even for simple distributed computations.

20.7 Recognizing Lattice-Linear and Regular Predicates

As discussed earlier, efficient detection algorithms exist for various classes of predicates. Thus, given a boolean expression B, one would like to determine if it belongs to a tractable subclass, in which case detection of the predicate may be performed efficiently. We first consider the classes of Lattice-Linear and regular predicates. In this section, we show that determining whether a given boolean expression is Lattice-Linear with respect to a given distributed computation is a co-NP-complete problem. We also show that this problem is co-NP-complete for regular predicates, as well as post-Lattice-Linear predicates.

We define the decision problems of predicate recognition for Lattice-Linear and regular predicates as follows.

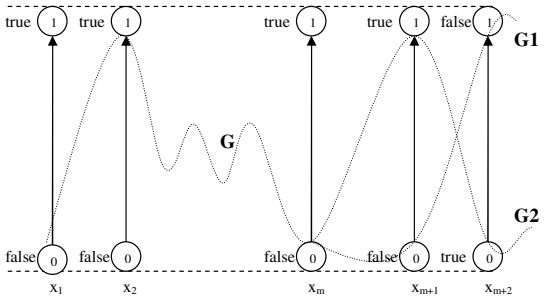


Figure 20.6: Transformation for Theorems 20.3 and 20.4.

Lattice-Linearity: Given a boolean expression b and a program computation, is b a Lattice-Linear predicate?
Regularity: Given a boolean expression b and a program computation, is b a regular predicate?

Theorem 20.3 Lattice-Linearity *is co-NP-complete.*

Proof: *Lattice-Linearity is in co-NP*: Given a pair of candidate global states G and H in which the predicate is true, it can be easily verified in polynomial time that the predicate is false in the global state $G \cap H$. Thus, Lattice-Linearity is in co-NP.

Lattice-Linearity is co-NP-hard: To show co-NP-hardness, we transform an arbitrary instance of TAUTOLOGY to an instance of Lattice-Linearity.

Let b be a boolean expression involving variables $x_1, x_2, ..., x_m . \forall i = 1..m$, we place each x_i on a separate process, P_i. Each of these m processes has two local states, a *true* state and a *false* state, which corresponds to the value taken by the variable x_i in that local state. We also define two new variables, x_{m+1} and x_{m+2}, and place them on processes P_{m+1} and P_{m+2} respectively. Process P_{m+1} has two local states: an initial *false* state and a final *true* state, and process P_{m+2} has two local states: an initial *true* state and a final *false* state. Figure 20.6 shows this transformation. It is evident that this transformation can be achieved in polynomial time.

We define

$$B = b \vee x_{m+1}x_{m+2} \vee \overline{x}_{m+1}\overline{x}_{m+2}$$

We claim that B is Lattice-Linear iff b is a tautology. If b is a tautology, then B is trivially Lattice-Linear, since B will be true for all global states. Conversely, if b is not a tautology, then there exists a subcut involving processes $P_1 ... P_m$ in which b evaluates to false. Let us call this subcut G. We can now extend the subcut G to form two cuts, $G1$ and $G2$, in which the predicate B is true, as shown in Figure 20.6.

$$G1 = (G, 0, 1)$$

and

$$G2 = (G, 1, 0)$$

However,

$$G1 \cap G2 = (G, 0, 0)$$

in which the predicate B is false. Thus, B is not Lattice-Linear.

■

Theorem 20.4 Regularity *is co-NP-complete.*

Proof: *Regularity is in co-NP*: Given two candidate global states in which the predicate is true, and their union and intersection such that the predicate is false in either the union or intersection, it can be easily verified in polynomial time that the predicate is not regular. Thus, Regularity is in co-NP.

Regularity is co-NP-hard: The transformation in Theorem 20.3 holds for Regularity as well. That is, b is a tautology iff $B = b \vee x_{m+1}x_{m+2} \vee \overline{x}_{m+1}\overline{x}_{m+2}$ is regular. If b is a tautology, then B is trivially regular since B is true for all global states. Conversely, if b is not a tautology, then B is not regular since both the intersection and union of $G1$ and $G2$ result in a global state in which B is false. Thus, Regularity is co-NP-complete.

■

Note that the above transformation can also be used to show that the problem of deciding whether a given boolean predicate is dual-Lattice-Linear is co-NP-complete.

20.8 Efficient Advancement Property

We stated earlier that efficient detection of Lattice-Linear predicates relies on the assumption that the given predicate satisfies the efficient advancement property, that is, the forbidden state can be identified in polynomial time. The question that arises is, do all Lattice-Linear predicates satisfy the efficient advancement property? If not, then is it possible to efficiently detect Lattice-Linear predicates that do not satisfy this property? We show here that, unless RP=NP, polynomial-time detection cannot be performed for all Lattice-Linear predicates.

We use a result by Valiant and Vazirani [VV85], which states that satisfiability is NP-hard under randomized reductions even for instances that have at most one satisfying assignment (USAT). Valiant and Vazirani's proof uses a randomized polynomial-time algorithm that reduces a given instance of SAT to an instance of USAT.

Theorem 20.5 (Valiant-Vazirani) *If there exists a randomized polynomial-time algorithm for solving instances of SAT having at most one satisfying assignment, then NP=RP.*

We know that a predicate having at most one satisfying assignment is Lattice-Linear, so every instance of USAT is a Lattice-Linear predicate. Given any instance B of USAT, involving variables $x_1, x_2, ..., x_m$, one can create a distributed computation as depicted in Figure 20.5, such that detecting $possibly : B$ is equivalent to solving USAT for B. Since we know that Lattice-Linear predicates that satisfy efficient advancement can be detected in polynomial-time, this indicates that Lattice-Linear predicates that do not exhibit efficient advancement may not be detected in polynomial-time, even by a randomized algorithm. Furthermore, since every instance of USAT is also a regular predicate, detection of regular predicates is also NP-hard under randomized reductions.

20.9 Problems

20.10 Bibliographic Remarks

The detection of conjunctive predicates was discussed by Garg and Waldecker in [GW92]. Distributed on-line algorithms for detecting conjunctive predicates were presented in Garg and Chase [GC95]. Observer-independent predicates were introduced by Charron-Bost, Delporte-Gallet, and Fauconnier [CBDGF95]. Hurfin, Mizuno, Raynal and Singhal [HMRS95] gave a distributed algorithm that does not use any additional messages for predicate detection. Distributed algorithms for offline evaluation of global predicates are also discussed in Venkatesan and Dathan [VD92]. Stoller and Schneider [SS95] have shown how Cooper and Marzullo's algorithm can be integrated with that of Garg and Waldecker's to detect a conjunction of global predicates.

The second algorithm for detecting conjunctive predicates is from [GG19]. The **NP**-completeness of detecting a conjunctive predicate at the level k is shown in [GS24].

The complexity of detecting a boolean predicate is taken from [CG98]. The complexity of checking whether a predicate is lattice-linear (or regular) is from [KG05].

Chapter 21

Enumeration Algorithms

21.1 Introduction

So far, we have been concerned with finding the least feasible elements of the set satisfying the feasibility predicate. What if we wanted to enumerate all feasible elements. For example, we may want to enumerate all satisfying assignments for Horn formulas, all stable marriages, all integral market clearing prices etc. In general the feasible set may be large. For most applications, we consider the set of feasible elements is exponentially larger than the input size. Our goal would be to reduce the complexity of enumeration per element.

21.2 Birkhoff's Theorem

We first define *join-irreducible* elements as follows.

Definition 21.1 (Join-Irreducible Elements) *An element $x \in L$ is join-irreducible if*

1. $x \neq 0$, *and*
2. $\forall a, b \in L : x = a \sqcup b \Rightarrow (x = a) \vee (x = b)$.

Pictorially, in a finite lattice, an element is join-irreducible iff it has exactly one lower cover, that is, there is exactly one edge coming into the element. Figure 21.1(a) shows a distributive lattice with its join-irreducible elements. Intuitively, the set of join-irreducible elements of a distributive lattice are analogous to basis elements of a linear vector space in the sense that the lattice can be generated using join-irreducible elements in the same way as the vector space can be generated by linear combination of basis elements. We denote the set of consistent cuts of any distributed computation $(E, \rightarrow)$ by $C(E)$ ($\rightarrow$ is implicit). We exploit the property that the lattice is distributive to derive the notion of a computation slice. Let $J(L)$ denote the set of join-irreducible elements in L. Now we can state Birkhoff's theorem for finite distributive lattices.

Theorem 21.2 (Birkhoff's Representation Theorem) *Let L be a finite distributive lattice. Then the map $f : L \rightarrow C(J(L))$ defined by*

$$f(a) = \{x \in J(L) \mid x \leq a\}$$

is an isomorphism of L onto $C(J(L))$. Dually, let P be a finite poset. Then the map $g : P \rightarrow J(C(P))$ defined by

$$g(a) = \{x \in P \mid x \leq a\}$$

is an isomorphism of P onto $J(C(P))$.

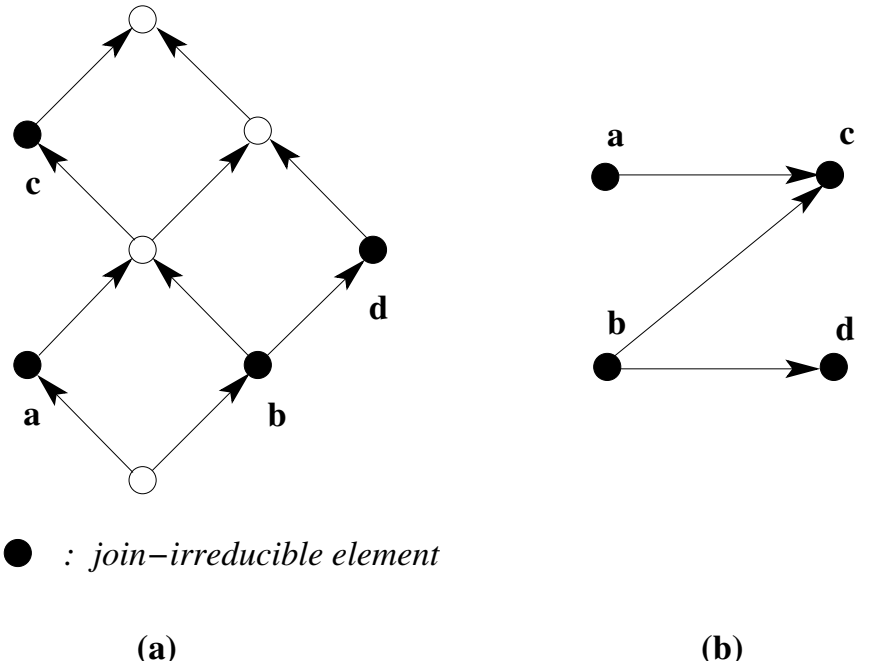


Figure 21.1: (a) An example of a distributive lattice (b) its partial order representation.

The above theorem implies one-to-one correspondence between a finite poset and a finite distributive lattice. Given a finite poset, we get the finite distributive lattice by considering its set of down-sets. Given a finite distributive lattice, we can recover the poset by focusing on its join-irreducible elements. Informally, any element of a lattice can be written as a join of a subset of join-irreducible elements of the lattice. For example, Figure 21.1(b) gives the poset corresponding to the lattice in Figure 21.1(a).

From the above discussion it is clear that given any finite distributed computation, the structure *finite distributive lattice* completely characterizes its execution graph.

21.3 Slicing

The notion of consistent global states or ideals of a poset can be extended to graphs in a straightforward manner. A subset of vertices, H, of a directed graph, P, is an *ideal* if it satisfies the following condition: if H contains a vertex v and (u, v) is an edge in the graph, then H also contains u. Observe that an ideal of P either contains all vertices in a strongly connected component or none of them. Let L denote the set of ideals of a directed graph P. Observe that the empty set and the set of all vertices trivially belong to L. We call them *trivial* ideals. It is easy to show that given a directed graph P, $(L; \subseteq)$ forms a distributive lattice.

Since trivial ideals are always part of L, it is more convenient to deal only with nontrivial ideals of a graph. It is easy to convert a graph P to P' such that there is one-to-one correspondence between all ideals of P and all nontrivial ideals of P'. We construct P' from P by adding two additional vertices $\perp$ and $\top$ such that $\perp$ is the smallest vertex and $\top$ is the largest vertex (i.e., there is a path from $\perp$ to every vertex and a path from every vertex to $\top$). It is easy to see that any nontrivial ideal will contain $\perp$ and not contain $\top$. As a result, every ideal of P is a nontrivial ideal of the graph P' and vice versa. We will deal with only nontrivial ideals from now on and an ideal would mean nontrivial ideal unless specified otherwise.

We will deal with only nontrivial ideals from now on and an ideal would mean nontrivial ideal unless specified otherwise. The directed graph representation of Fig. 21.2(a) is shown in Fig. 21.2(c).

The slice of a directed graph P with respect to a predicate B (denoted by $slice(P, B)$) is a graph derived from

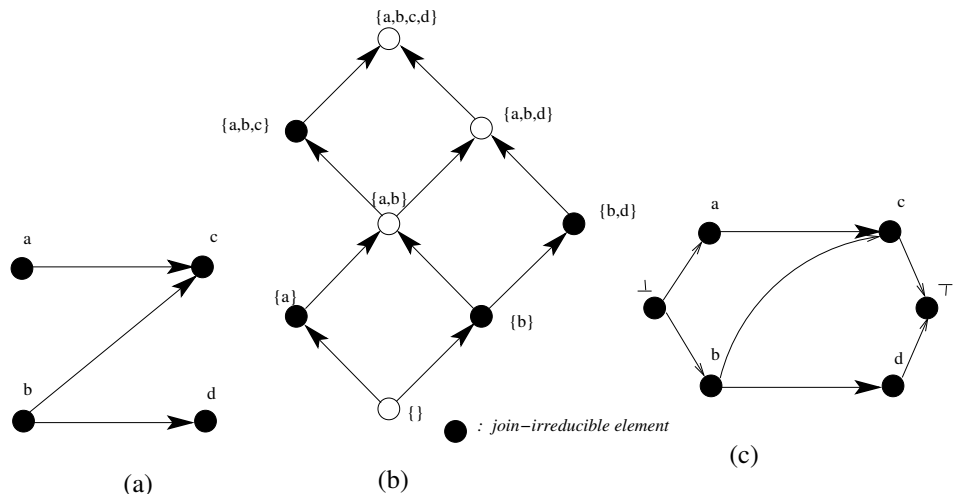


Figure 21.2: (a) P: A partial order (b) the lattice of ideals. (c) the directed graph P'

P such that all the ideals in L that satisfy B are included in the ideals of the slice with respect to B. The slice may include some additional ideals which do not satisfy the predicate. Formally,

Definition 21.3 (Slice) *A* slice of a graph P with respect to a predicate B is the directed graph obtained from P by adding edges such that:
(1) it contains all the ideals of P that satisfy B and
(2) of all the graphs that satisfy (1), it has the least number of ideals.

Let L be a finite distributive lattice generated by the graph P. Let L' be any sublattice of L. Then, there exists a graph P' that can be obtained by adding edges to P that generates L'.

21.4 Computing All Stable Matchings

We now consider the problem of computing all stable matchings. Since the number of stable matchings may be exponential in n, instead of keeping all matchings in explicit form, we would like a concise representation of polynomial size that can be used to enumerate all stable matchings. In SMP literature, rotation posets are used to capture all stable matchings. We will use the notion of computation slicing introduced in [MG01b] for this purpose. In particular, we give an efficient algorithm to compute the slice for the SMP computation. A rotation poset [GI89] is a special case of the slice when the set of external constraints is empty.

We first show that the set of stable matchings is a sublattice of the lattice of all men assignments. We have already shown that the predicate "the assignment is a stable matching" is a lattice-linear predicate and therefore closed under meets. We now show that the predicate is also closed under joins. To that end, we use the dual of a lattice-linear predicate called dual-linear predicate. Given any predicate B, we define

$$\text{reverse-forbidden}(G, i) \equiv \forall H \in L : H \subseteq G : (G[i] \neq H[i]) \vee \neg B(H)$$

We define a predicate B to be dual-linear with respect to poset S if for any global state G in the poset, B is false in G implies that G contains a *reverse-forbidden state*. Formally, A boolean predicate B is *dual-linear* with respect to a poset S iff:

$$\forall G \in L : \neg B(G) \Rightarrow \exists i : \text{reverse-forbidden}(G, i)$$

Note that although the concepts of lattice-linear predicate and dual-linear predicates are dual, the proof of dual-linearity is different from that of lattice-linearity. This is because we are computing assignments with respect to men and not women thereby bringing in asymmetry.

Lemma 21.4 *Let G be any assignment such that it is not a stable matching. Then, there exists i such that $G[i]$ is reverse-forbidden.*

Proof: First assume that G is not a matching. We know that there is at least one woman q who is missing in this assignment. Let i be the most preferred man for that woman in the pre-frontier events that correspond to proposal to q. If there is no man in pre-frontier events for that woman who proposes to q, then there is no matching possible because q is not proposed in either a frontier or a pre-frontier event. So, we can assume that i exists. We claim that $G[i]$ is reverse-forbidden. Consider any H such that $H \subseteq G$ and $H[i] = G[i]$. We need to show that H cannot be a stable matching. For H to be a matching, for some k different from i, $H[k].w$ equals q. However, by our choice of i, q prefers man i and man i prefers q to $H[i].w$.

Now, assume that G is a matching, but not stable. Suppose that (j, k) is a blocking pair. Let q be the woman corresponding to $G[j].w$. Of all the men who propose to that woman in pre-frontier events, choose the one who is most preferred for that woman. We know that there exists at least one because (j, k) is a blocking pair. Let that man be i. We claim that $G[i]$ is reverse-forbidden. Consider any H such that $H \subseteq G$ and $H[i] = G[i]$. We need to show that H cannot be a stable matching. Since G is a matching, $G[i].w$ is different from $q = G[j].w$. For H to be a matching, for some l, $H[l].w$ equals $G[j].w$. However, by our choice of i, we know that $H[l].w$ prefers man i to l and man i prefers $H[l].w$ to $G[i].w$.

■

The above lemma allows us to find the man-pessimal stable matching. We start with G such that $G[i]$ equals the last choice proposal for P_i. If G is a stable matching, we are done. Otherwise, from the proof of Lemma 21.4, we can find i such that unless $G[i]$ goes backward, there cannot be any stable matching. By repeating this procedure, we get the men-pessimal stable matching.

Since stable matching is dual-linear, it follows that the set of assignments is closed under joins. Predicates that are both meet-closed and join-closed are called regular predicates [GM01b]. The set of ideals that satisfy a regular predicate forms a sublattice of the lattice of all ideals. Since a sublattice of a distributive lattice is also distributive, the set of ideals that satisfy a regular predicate forms a finite distributive lattice. From Birkhoff's theorem [Bir67] we know that a finite distributive lattice can be equivalently represented using the poset of its join-irreducible elements.

For any regular predicate B, let L_B be the sublattice of consistent global states that satisfy B. Since L_B can have a number of elements that is exponential in n, we would like to have a compact representation of L_B. A slice of a poset P with respect to a predicate B is a graph such that the consistent global states of the graph includes all consistent global states that satisfy B and when B is regular it includes only those. The computation of a slice is shown in Fig. 21.3. The algorithm requires computation of $J(B, e)$ for all $e \in E$ where $J(B, e)$ be the least consistent global state that satisfies B and includes e.

```
(1)  graph function computeSlice(B:regular_predicate, P: graph)
(2)  var R: graph initialized to P;
(3)  begin
(4)      for every element e in P do
(5)          let J(B,e) be the least global state of P that satisfies B and includes e;
(6)          for every f ∈ J(B,e) do;
(7)              add edge (f,e) to R;
(8)      return R;
(9)  end;
```

Figure 21.3: An efficient algorithm to compute the slice for a regular predicate B

Hence, to compute a slice it is sufficient to give a procedure to compute $J(B, e)$. To determine $J(B, e)$ it is sufficient to use the algorithm for detecting lattice-linear predicate by using the following predicate for every e:

$B_e(G) \equiv B(G) \wedge (e \in G)$. Since B is a lattice-linear predicate, and the predicate $e \in G$ is also lattice-linear, $B_e(G)$ is also lattice-linear.

21.5 An Algorithm to Determine Join-Irreducible Min Cuts

To identify join-irreducible min cuts:

1. Construct the lattice of all min cuts, ordered by inclusion of S-sets.
2. For each min cut $C = (S, T)$, find its immediate predecessors: min cuts $C' = (S', T')$ where $S' \subset S$ and no S'' exists such that $S' \subset S'' \subset S$.
3. A min cut is join-irreducible if it has exactly one immediate predecessor.

Consider the directed graph in Fig. 21.4 with 4 vertices.

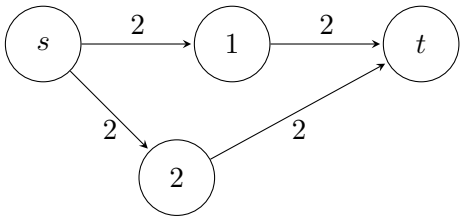


Figure 21.4: Directed graph with capacities (acyclic).

The graph has a minimum cut capacity of 4. The min cuts are: $\{s\}|\{1,2,t\}$, $\{s,1\}|\{2,t\}$, $\{s,2\}|\{1,t\}$, and $\{s,1,2\}|\{t\}$. These mincuts form a distributive lattice shown in Fig. 21.5

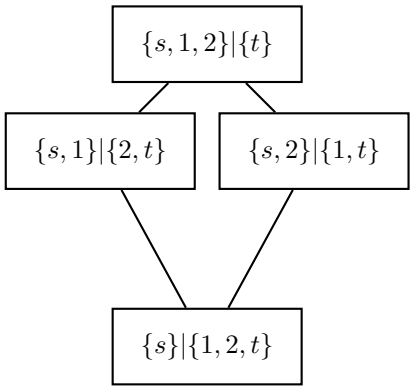


Figure 21.5: Lattice of minimum cuts for the acyclic directed graph.

As mentioned earlier, in the worst case, the lattice may be exponential in the size of the graph. We construct the poset of join-irreducible mincuts which generates the lattice of mincurs. This poset has at most $n - 2$ elements.

- $\{s\}|\{1,2,t\}$: No predecessors (bottom element), so join-irreducible.
- $\{s,1\}|\{2,t\}$: One predecessor ($\{s\}$), so join-irreducible.
- $\{s,2\}|\{1,t\}$: One predecessor ($\{s\}$), so join-irreducible.
- $\{s,1,2\}|\{t\}$: Two predecessors ($\{s,1\}$, $\{s,2\}$), so not join-irreducible.

Thus, the join-irreducible min cuts are $\{s\}|\{1,2,t\}$, $\{s,1\}|\{2,t\}$, and $\{s,2\}|\{1,t\}$.

We now give an efficient algorithm to generate all the join-irreducible elements of the mincut lattice. For all vertices $v \in V$, we let $J(v)$ be the least mincut that includes the vertex v, if it exists. We first show that $J(v)$ is a join-irreducible mincut for all $v \in V - \{t\}$.

Lemma 21.5 *Let $v \in V - \{t\}$ and $J(v)$ be the least mincut that includes the vertex v. Then, $J(v)$ is a join-irreducible mincut.*

Proof: Let $J(v)$ be join of two mincuts X and Y such that $J(v)$ is different from both X and Y. Since the join corresponds to union, and $v \in J(v)$, either $v \in X$ or $v \in Y$. Without loss of generality, suppose that $v \in X$. Since $J(v)$ is the least minimal that includes v, we get $J(v) \subseteq X$. But, $J(v) = X \cup Y$ implies that $X \subseteq J(v)$.

■

We can compute $J(v)$ with two max-flow computations as shown in Fig. LeastJoinIrreducible. We first find the max-flow in the graph (line 1). Let μ be the max-flow. Now, we construct another max-flow graph G' as follows. We add an edge from the source vertex s to v with capacity equal to infinity (line 4). We again find the max-flow μ' (line 5). If μ' equals μ, then the mincut obtained is the least mincut that includes the vertex v (lines 7-9). If there is no mincut that includes v, then we return null.

Algorithm LeastJoinIrreducible: Generate Least Join-Irreducible Min Cut including v

Data: Directed graph $G = (V, E)$, capacities $c(e)$, source s, sink t
Result: Join-irreducible min cut $J(v)$

```
1  μ ← min cut capacity of G ;                          // One max-flow
2  J(v) ← null ;                                        //
3  G' ← G ;                                             // Copy graph
4  Add edge s → v with capacity ∞ in G' ;               // Force v with s
5  (S_v, T_v) ← min cut in G' (via max-flow)
6  μ' ← capacity of cut (S_v, T_v) in original G
7  if μ' = μ then
8  |   J(v) ← (S_v, T_v)}
9  end
10 return J(v)
```

We now extend this algorithm to generate J, the list of all join-irreducible mincuts. We observe here that the algorithm is polynomial in the input size.

Algorithm AllJoinIrreducible1: Generate Join-Irreducible Min Cuts

Data: Directed graph $G = (V, E)$, capacities $c(e)$, source s, sink t
Result: Set J of join-irreducible min cuts

$J \leftarrow \emptyset$
$Covered \leftarrow \emptyset$; // Vertices in some S-set
$\mu \leftarrow$ min cut capacity of G (via max-flow) $S_0 \leftarrow \{s\}$, $T_0 \leftarrow V \setminus \{s\}$
$C_0 \leftarrow$ capacity of cut (S_0, T_0)
if $C_0 = \mu$ **then** $J \leftarrow J \cup \{(S_0, T_0)\}$ $Covered \leftarrow Covered \cup S_0$;
for $v \in V \setminus \{s, t\}$ **do**
    if $v \in Covered$ **then continue**;
    $G' \leftarrow G$; // Copy graph
    Add edge $s \rightarrow v$ with capacity ∞ in G' ; // Force v with s
    $(S_v, T_v) \leftarrow$ min cut in G' (via max-flow)
    $C_v \leftarrow$ capacity of cut (S_v, T_v) in original G
    if $C_v = \mu$ **then**
        $J \leftarrow J \cup \{(S_v, T_v)\}$
        $Covered \leftarrow Covered \cup S_v$
    end
end
return J

The algorithm works as follows.

- Compute μ once with a max-flow algorithm.
- Check the minimal cut $\{s\}|V \setminus \{s\}$; add it to J if it is a min cut and mark its vertices as covered.
- For each $v \in V \setminus \{s, t\}$, skip if already covered. Otherwise, compute the least min cut containing v using max-flow.
- Add the cut to J if its capacity is μ, updating *Covered*.
- The number of max-flow computations is $O(k)$, where k is the number of join-irreducible min cuts ($k \leq |V| - 2$).
- Total time complexity: $O(kMF(n, m)$, where $MF(n, m)$ is the time complexity of computing maxflow in a graph with n vertices and m edges. The worst case for k is $O(n)$.

We now show how the computation complexity of the above algorithm can be improved. We show an algorithm that avoids the second computation of max-flow. We assume that we have access to the reduced graph from the first computation of max-flow. Based on the first max-flow, the algorithm computes a reduced graph of *super-vertices*. Each super-vertex corresponds to a subset of vertices such that if u is in the super-vertex and (u, v) is not a staurated edge, then v is also in the supervertex. The reduced graph can be computed in time proportional to the number of edges. The algorithm is inspired from [PQ80]. A difference is that they do not explicitly generate *join-irreducible* mincuts. Furthermore, we later extend this algorithm to find a mincut that satisfies the given predicate B.

Algorithm AllJoinIrreducible2: Generate Join-Irreducible Min Cuts (Directed Graphs)

Data: Directed graph $G = (V, E)$, capacities $c(e)$, source s, sink t
Result: Set J of join-irreducible min cuts

$J \leftarrow \emptyset$
$Covered \leftarrow \emptyset$; // Vertices in some S-set
$\mu \leftarrow$ min cut capacity of G (via max-flow) ; // max-flow
$S_0 \leftarrow \{s\}$, $T_0 \leftarrow V \setminus \{s\}$
$C_0 \leftarrow$ capacity of cut (S_0, T_0)
if $C_0 = \mu$ **then** $J \leftarrow J \cup \{(S_0, T_0)\}$ $Covered \leftarrow Covered \cup S_0$;
for $v \in V \setminus \{s, t\}$ **do**
  if $v \in Covered$ **then continue**;
  find strongly connected component W that includes v with all edges in the residual graph;
  if $t \notin W$ **then**
    $S_v \leftarrow S_0$ union with all vertices reachable from W using edges in the residual graph
    $T_v \leftarrow V \setminus S_v$
    $J \leftarrow J \cup \{(S_v, T_v)\}$
    $Covered \leftarrow Covered \cup S_v$
  end
end
return J

21.6 Problems

21.1. Show that $x \in L$ is join-irreducible iff
(i) $x \neq 0$
(ii) $\forall a, b : (a < x) \wedge (b < x) \Rightarrow ((a \sqcup b) < x)$

21.2. The dual notion of join-irreducible elements is *meet-irreducible* elements. Let $M(L)$ denote the ordered set of meet-irreducible elements of L. Show that the ordered set $J(L)$ is isomorphic to the set $M(L)$.

21.7 Bibliographic Remarks

The notion of a slice of a computation was proposed in Garg and Mittal [GM01a] and later generalized in Mittal and Garg [MG01a].