THIS IS THE PARALLEL BOOK VERSION

A Systematic Approach to Parallel 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

1.1 Introduction

This book is on parallel algorithms. 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

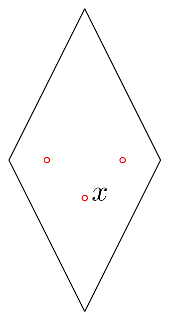


Figure 1.1: Search space with feasible solutions. The feasible solutions are shown in red. The solution x is the least feasible solution.

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 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.

1.2 Computing Performance

Parallel algorithms are a crucial aspect of today's technology. At one time, computer processing power grew faster every year. However, this is no longer the case. Increased performance is now typically achieved by adding processors. Even smartphones are multi-core. There are two main styles of using multiple processors to accomplish computing goals: distributed and parallel computing systems.

Distributed computing systems contain multiple processors, with their own memory spaces, that are connected by a communication network. Typically, a single task is divided amongst individual computers which communicate and collaborate with each other by passing messages via the network.

On the other hand, parallel computing systems consist of multiple processors, referred to as processing elements (PE), which communicate using a shared memory space. This book focuses on algorithms that use and optimize parallel computing systems. The actual implementation of parallel computing systems is more detailed and complex than the diagram in Fig. 1.2. For instance, each PE typically has its own cache memory. We use the simplified model for this book.

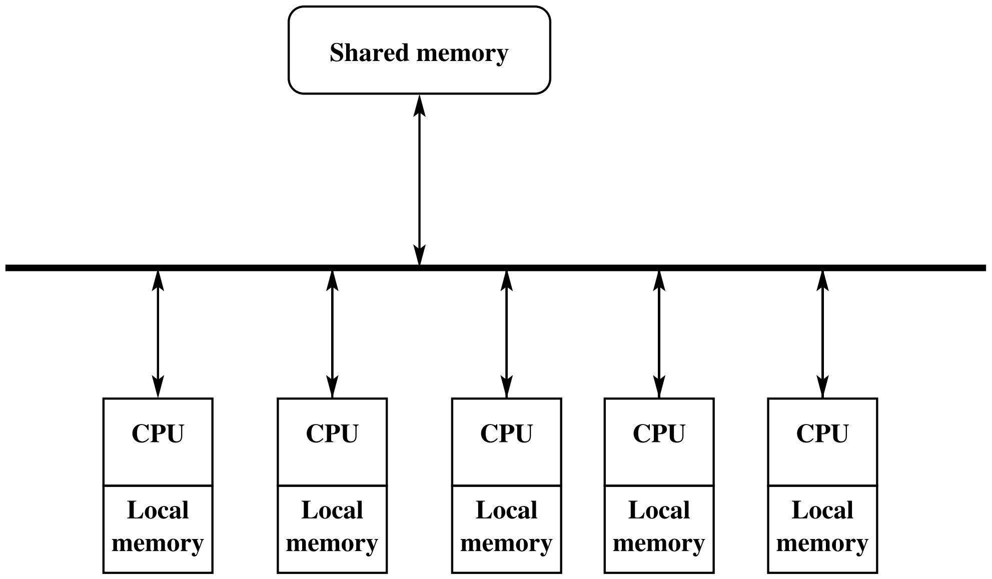


Figure 1.2: A parallel system

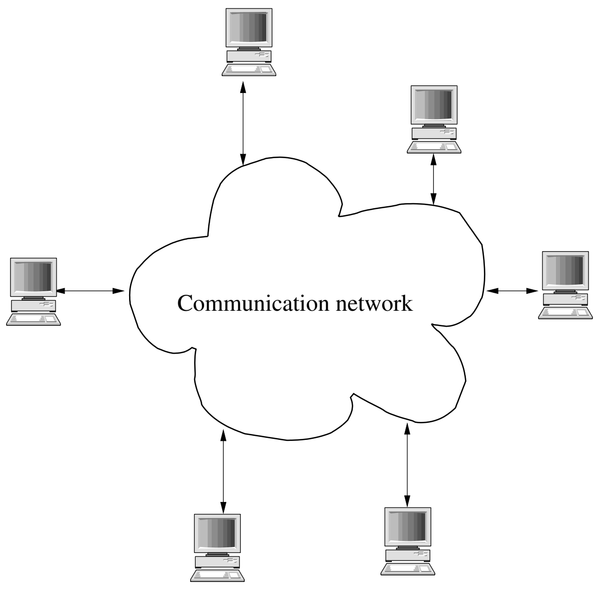


Figure 1.3: A distributed system

1.3 The Parallel Random Access Machine

When multiple processing elements (PE) are accessing the shared memory space, it is straightforward to understand when they are accessing different memory locations at each point in time. However, what happens when multiple PEs try to read and write to the same memory location?

To describe simultaneous memory access between two or more PE, we use the following letters to describe that access:

- **E** = Exclusive
- **C** = Concurrent
- **R** = Read
- **W** = Write

Parallel computing machines will be referred to as **PRAM** (Parallel Random Access Machine). PRAMs can be group into four main categories describing the model's behavior for simultaneous access to the same memory location:

- **EREW** (Exclusive Read Exclusive Write) - This is the most restrictive model in which only one PE can read or write to a memory location at a time.
- **CREW** (Concurrent Read Exclusive Write) - This is the most frequently used model in which multiple PEs can read a memory location at the same time, but only one can write at a time.
- **ERCW** (Exclusive Read Concurrent Write) - In this model, multiple PEs can write on a memory location but the read is exclusive. This model is generally not useful.
- **CRCW** (Concurrent Read Concurrent Write) - Multiple PEs can both read and write to a memory location simultaneously.

It is understandable in modern computing that multiple PEs reading from the same memory location simultaneously should not cause an issue, but concurrent writes could. Therefore concurrent writes are further defined as:

- **Common** - All values being written by the multiple PEs are all the same (common), and therefore allowed.
- **Priority** - A predetermined priority of the PEs is used to determine which value gets written.
- **Arbitrary** - A random value is picked to be written.

We use CREW and CRCW type PRAM models in this book.

1.4 Reduce

Let us now take a look at how parallel algorithms can be used to solve a simple problem. Consider an array A of size $n = 8$, $A = [8, 2, 9, 1, 3, 5, 11, 7]$. To find the maximum entry in the array A using a sequential algorithm, one starts at the beginning of the array and compares each entry keeping track of the maximum

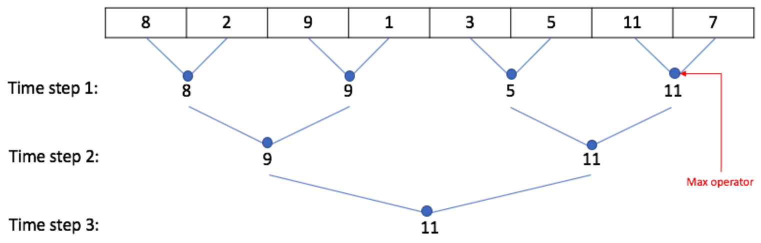


Figure 1.4: Computing the maximum of an array using the *Reduce* operation.

value along the way, until the end of the array is reached. The time complexity for this sequential algorithm is $O(n)$.

This problem can be solved in parallel by allowing each PE to compute a pairwise maximum value as shown in Fig. 1.4. In this example, the first time step entails four PEs computing the pairwise maximum on the values of the original array. The result is that the problem has now been reduced from eight elements in the array to only four elements. During the second time step, two PEs compute the pairwise maximum of the values computed in the previous step. Finally, the overall maximum value is calculated from the two values from the previous step. This algorithm is called the **Reduce**. The time complexity for this parallel algorithm is $O(\log n)$.

Notice that the *max* operator is associative, meaning that the order in which the operators are performed does not matter. It turns out that this same Reduce algorithm can be used with any operator that is associative including the minimum, the sum, the multiplication, the logical AND, and the logical OR operators. There are no concurrent writes in this algorithm and can be easily implemented on a CREW PRAM.

In addition to the time complexity, it is also useful to analyze the amount of work being done by our algorithms. The work complexity can be thought of as the *sequential* time complexity of the algorithm. In our example of computing the maximum of an array, we do $O(n)$ work even though the parallel time complexity is $O(\log n)$. Notice that, in comparison to the sequential algorithm, the *Reduce* algorithm reduces the time complexity (from $O(n)$ to $O(\log_2 n)$), and adds no additional work complexity. We define such a parallel algorithm as **work optimal**; its work complexity matches that of the best known sequential algorithm for the problem.

We now show a trick that can frequently be used for parallel algorithms. This trick cascades the sequential algorithm with the parallel algorithm as follows. We first consider the array of n divided into $n/\log_2 n$ subarrays, each of size $\log_2 n$. For each subarray of size $\log_2 n$, we use a core to find the maximum sequentially. We have used $n/\log_2 n$ cores and $O(\log_2 n)$ time to get the maximum of all subarrays. In the second step, we use the parallel algorithm to compute the maximum of $n/\log_2 n$ numbers. The second step requires at most $O(\log n)$ time. The key advantage of this approach is that we have used only $n/\log_2 n$ cores. The total time complexity of the *cascaded* algorithm is still $O(\log n)$.

1.5 Brent's Scheduling Principle

So far our simplified analysis of parallel algorithms has assumed that there are an unlimited number of PEs available for the parallel algorithm. However, this may not always be realistic, so it is useful to analyze the time performance considering hardware limitations.

To accomplish this, in addition to n (the problem size), we introduce another variable p as the number of available PEs (or cores). We define $T(n)$ as the parallel time complexity of problem size n with any number of cores, $W(n)$ as the work complexity of n, and $T(n,p)$ as the bounded time complexity due to both problem size n and a certain number of available cores p. Brent's Scheduling Principle states:

$$T(n,p) = O(\frac{W(n)}{p} + T(n)) \tag{1.1}$$

This can be derived by summing the work performed at each parallel time step:

$$\begin{aligned} T(n,p) &= \sum_{i=1}^{T(n)} (\frac{W^i(n)}{p} + 1) \\ &= \frac{1}{p} \sum_{i=1}^{T(n)} W^i(n) + \sum_{i=1}^{T(n)} 1 \\ &= \frac{W(n)}{p} + T(n) \end{aligned} \tag{1.2}$$

Now let us apply this principle to the cascaded reduce algorithm. We observed that the time complexity $T(n) = O(\log_2 n)$, and work $W(n) = O(n)$. We used $p = n/\log_2 n$ cores, so let us apply Brent's Scheduling Principle using this number of cores:

$$\begin{aligned} T(n,p) &= O(\frac{W(n)}{p} + T(n)) \\ &= O(\frac{n}{n/\log_2 n} + \log_2 n) \\ &= O(\log_2 n) \end{aligned} \tag{1.3}$$

We see here that Brent's Scheduling principle gives the same answer as the time complexity previously computed when the number of cores is set to $n/\log_2 n$.

1.6 Parallel Complexity: Class *NC*

We now turn our attention to the parallel complexity of the problems. The class NC is the set of problems that can be solved *efficiently* on a parallel computer. By *efficiently* on a parallel computer, we mean that the parallel time it takes to solve a problem of size n is $O(\log^k(n))$ for a constant k. Observe that for the sequential complexity, we use *efficiently* to mean that the problem can be solved in time equal to $O(n^k)$ for a constant k. We require a significant speedup on a parallel computer for the problem to be in the class NC.

For the formal definition, we first define a *language* L to be a subset of strings over the alphabet $\Sigma = \{0,1\}$. The language-recognition problem is a binary decision problem that takes an arbitrary string s and decides whether the string belongs to L.

Definition 1.1 *A langauge L is in NC if there exists a PRAM algorithm that can decide for any input string s of size n whether s belongs to L in $O(\log^k n)$ time for a fixed k using $O(n^c)$ processors for some fixed c.*

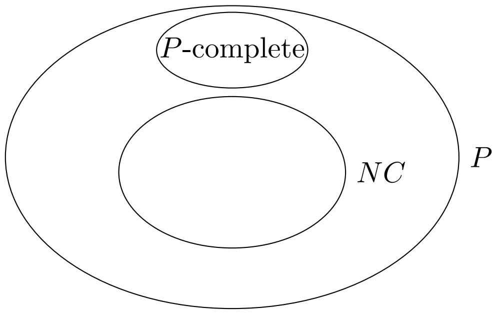


Figure 1.5: Complexity classes P, NC, and P-complete.

Since a single processor can simulate behavior of multiple processors, it is clear that $NC \subseteq P$. Is the converse true? This is a major open problem in computation complexity. The current belief is that the converse is false and there are problems for which there do not exist any efficient parallel algorithms. In a manner analogous to P vs NP question, we define a subclass of problems in P, called *P-complete* that we think are the hardest to parallelize.

We first define the notion of *NC-reductions*. A language L_1 is *NC-Reducible* to L_2, if there exists an NC algorithm to transform an input string s_1 of L_1 to a string s_2 such that $s_1 \in L_1$ iff $s_2 \in L_2$. Clearly, if L_1 is *NC-reducible* to L_2, then any NC algorithm for L_2 can be converted to an NC algorithm for L_1. Hence, L_2 is at least as hard as L_1. It is clear that the notion of *NC-reducible* is transitive.

We can now define the notion of *P-completeness*. Fig. 1.5 shows a pictorial representation of these complexity classes.

Formally,

Definition 1.2 *A language L is P-complete if it is in P and every language in P is NC-reducible to L.*

It follows that if L is *P-complete*, then $L \in NC$ implies $NC = P$.

Hence, it is unlikely that there exists any NC algorithm for any *P-complete* problem (similar to the belief that it is unlikely that there exists any polynomial time algorithm for any NP-complete problem).

We now list some *P-complete* problems.

- *Circuit Value Problem (CVP)*: Given a boolean circuit consisting of $\neg$, $\vee$ and $\wedge$ gates, and the input x_1, x_2, x_n, determine if the value of the circuit is 1.

- *Ordered depth-first-search*: Given a directed graph and three vertices s, u and v, determine if u is visited before v when a depth-first-search is started at s.

- *Maxflow*: Given a directed graph with two distinguished vertices s and t and the maximum capacity of each edge, determine if the maxflow in the graph is odd.

- *Linear Inequalities*: Given an $n \times m$ matrix and a n-dimensional vector b, determine if there exists a vector x such that $Ax \leq b$.

A proof of *P-completeness* of these problems is available in [JáJ92].

1.7 Problems

1. Model the problem of finding a satisfying assignment of a boolean expression as searching for an element in an appropriate distributive lattice.

2. Model the problem of finding the cost of reaching a vertex from a given vertex in a weighted graph as a search for a feasible element in a distributive lattice.

1.8 Bibliographic Remarks

There are many books on sequential and parallel algorithms. For sequential algorithms, the reader is referred to [CLRS01, KT06, HSR01]. For parallel algorithms, the reader is referred to [JáJ92, LB00, SMDD19]

Chapter 2

Lattice Linear Predicate Detection

2.1 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 2.2 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 2.3 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 2.4 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 *parallel*. It can advance on multiple components in parallel. 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.

2.2 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. Lattice-linearity is first defined in [CG98] in the context of detecting global conditions in a distributed system where it is simply called linearity. We use the term *lattice-linearity* to avoid confusion with the standard usage of linearity.

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 $G \le H$, if $H[i]$ equals $G[i]$, then B is false for H. Formally,

Definition 2.1 (Forbidden State [CG98]) *Given any distributive lattice L of n-dimensional vectors of $\mathbf{R}_{\ge 0}$, and a predicate B, we define $forbidden(G, i, B) \equiv \forall H \in L : G \le H : (G[i] = H[i]) \Rightarrow \neg B(H)$.*

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 2.2 (lattice-linear Predicate [CG98]) *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 $B_{jobs}(G) \equiv \forall j : (G[j] \ge t_j) \wedge (\forall i \in pre(j) : G[j] \ge G[i] + t_j)$. 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 $H \ge G$ 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. **Prefix Sum Problem**: Our second example relates to (exclusive) prefix sum of an array A with non-negative reals. We are required to output an array G such that $G[j]$ equals sum of all entries in A from 0 to $j-1$. We define G to be feasible iff B_{prefix} holds where $B_{prefix} \equiv (\forall j > 0) : (G[j] \ge G[j-1] + A[j-1])$. Again, it is easy to verify that B_{prefix} is lattice-linear. The minimum vector G that satisfies B_{prefix} corresponds to the exclusive prefix sum of the array A.

3. **Continuous Optimization Problem**: We are required to find minimum nonnegative x and y such that $B \equiv (x \ge y^2/4 + 5) \wedge (y \ge x - 4)$. 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 $B \equiv (x \ge 2y^2 + 5) \wedge (y \ge x - 4)$, 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).

4. **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 2.3 *Let B be any boolean predicate defined on a lattice L of vectors.*
(a) Let $f : L \to \mathbf{R}_{\geq 0}$ 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.

Proof: (a) Suppose B is false for G. This implies that $G[i] < f(G)$. Consider any vector $H \geq G$ such that $H[i]$ is equal to $G[i]$. Since $G[i] < f(G)$, we get that $H[i] < f(G)$. The monotonicity of f implies that $H[i] < f(H)$ which shows that $\neg B(H)$.
(b) This is shown in [CG98]. Assume that B is not lattice-linear. This implies that there exists a global state G such that $\neg B(G)$, and $\forall i : \exists H_i \geq G : (G[i] = H_i[i])$ and $B(H_i)$. Consider $Y = \cup_i \{H_i\}$. All elements of $Y \in L_B$. However, $inf\ Y$ which is in G is not an element of L_B. This implies that L_B is not closed under the infimum operation. Conversely, let $Y = \{H_1, H_2, \ldots, H_k\}$ be any subset of L_B such that its infimum G does not belong to L_B. Since G is the infimum of Y, for any i, there exists $j \in \{1 \ldots k\}$ such that $G[i] = H_j[i]$. Since $B(H_j)$ is true for all j, it follows that there exists a G for which lattice-linearity does not hold.

(c) Follows from the equivalence of meet-closed predicates with lattice-linearity and that meet-closed predicates are closed under conjunction. For a more direct proof, suppose that $\neg(B_1 \wedge B_2)$. This implies that one of the conjuncts is false and therefore from the lattice-linearity of that conjunct, a forbidden state exists.

■

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 2.3(a) that B_j is lattice-linear. The predicate $B_{jobs} \equiv \forall j : B_j$ is lattice-linear due to Lemma 2.3(c). Also note that the problem of finding the minimum vector that satisfies B_{jobs} is well-defined due to Lemma 2.3(b).

We now discuss detection of lattice-linear predicates which requires an additional assumption called the *efficient advancement property* [CG98] — there exists an efficient (polynomial time) algorithm to determine the forbidden state. This property holds for all the problems considered in this book. Once we determine j such that $forbidden(G, j, B)$, we also need to determine how to advance along index j. To that end, we extend the definition of forbidden as follows.

Definition 2.4 (α-forbidden) *Let B be any boolean predicate on the lattice L of all assignment vectors. For any G, j and positive real $\alpha > G[j]$, we define $forbidden(G, j, B, \alpha)$ iff*

$$\forall H \in L : H \geq G : (H[j] < \alpha) \Rightarrow \neg B(H).$$

Given any lattice-linear predicate B, suppose $\neg B(G)$. This means that G must be advanced on all indices j such that $forbidden(G, j, B)$. We use a function $\alpha(G, j, B)$ such that $forbidden(G, j, B, \alpha(G, j, B))$ holds whenever $forbidden(G, j, B)$ is true. With the notion of $\alpha(G, j, B)$, we have the algorithm LLP shown in Fig. LLP. The algorithm LLP has two inputs — the predicate B and the top element of the lattice T. It returns the least vector G which is less than or equal to T and satisfies B (if it exists). Whenever B is not true in the current vector G, the algorithm advances on all forbidden indices j in parallel. This simple parallel algorithm can be used to solve a large variety of combinatorial optimization problems by instantiating different $forbidden(G, j, B)$ and $\alpha(G, j, B)$.

Algorithm LLP: Algorithm LLP to find the minimum vector at most T that satisfies B

```
vector **function** getLeastFeasible($T$: vector, $B$: predicate)
**var** $G$: vector of reals initially $\forall i : G[i] = 0$;
**while** $\exists j : forbidden(G, j, B)$ **do**
    **for all** $j$ such that $forbidden(G, j, B)$ **in parallel**:
        **if** $(\alpha(G, j, B) > T[j])$ then return null;
        **else** $G[j] := \alpha(G, j, B)$;
**endwhile**;
**return** $G$; // the optimal solution
```

Theorem 2.5 *Suppose there exists a fixed constant $\delta > 0$ such that $\alpha(G, j, B) - G[j] \geq \delta$ whenever $forbidden(G, j, B)$. Then, the parallel algorithm LLP finds the least vector $G \leq T$ that satisfies B, if one exists.*

Proof: Since $G[j]$ increases by at least δ for at least one forbidden j in every iteration of the *while* loop, the algorithm terminates in at most $\sum_i \lceil T[i]/\delta \rceil$ number of steps.

We show that the algorithm maintains the invariant $(I1)$ that for all indices j, any vector V such that $V[j]$ is less than $G[j]$ cannot satisfy B. Formally, the invariant $(I1)$ is

$$\forall j : (\forall V \in L : (V[j] < G[j]) \Rightarrow \neg B(V)).$$

Initially, the invariant holds trivially because G is initialized to 0. Suppose $forbidden(G, j, B)$. Then, we increase $G[j]$ to $\alpha(G, j, B)$. We need to show that this change maintains the invariant. Pick any V such that $V[j] < \alpha(G, j, B)$. We now do a case analysis. If $V \geq G$, then $\neg B(V)$ holds from the definition of $\alpha(G, j, B)$. Otherwise, there exists some k such that $V[k] < G[k]$. In this case $\neg B(V)$ holds due to $(I1)$.

We can now show Theorem 2.5 using the invariant. First, suppose that the algorithm LLP terminates because $\alpha(G, j, B) > T[j]$. In this case, there is no feasible vector in L due to the invariant (because the predicate B is false for all values of $G[j]$). Now suppose that the algorithm terminates because there does not exist any j such that $forbidden(G, j, B)$. This implies that G satisfies B due to lattice-linearity of B. It is also the least vector that satisfies B due to the invariant $(I1)$.

■

For the job scheduling example, we get a parallel algorithm to find the minimum completion time by using $forbidden(G, j, B_{jobs}) \equiv (G[j] < t_j) \vee (\exists i \in pre(j) : G[j] < G[i] + t_j)$, and $\alpha(G, j, B_{jobs}) = \max\{t_j, \max\{G[i] + t_j | i \in pre(j)\}\}$.

For the prefix sum example, we get a parallel algorithm by using $forbidden(G, j, B_{prefix}) \equiv (G[j] < G[j-1] + A[j-1])$ and $\alpha(G, j, B_{prefix}) = G[j-1] + A[j-1]$ for all $j > 0$.

We now show, on account of Lemma 2.3(c), that if we have a parallel algorithm for a problem, then we also have one for the constrained version of that problem.

Lemma 2.6 *Let LLP be the parallel algorithm to find the least vector G that satisfies B_1 if one exists. Then, LLP can be adapted to find the least vector G that satisfies $B_1 \wedge B_2$ for any lattice-linear predicate B_2.*

Proof: The algorithm LLP can be used with the following changes:
$forbidden(G, j, B_1 \wedge B_2) \equiv forbidden(G, j, B_1) \vee forbidden(G, j, B_2)$, and
$\alpha(G, j, B_1 \wedge B_2) = \max\{\alpha(G, j, B_1), \alpha(G, j, B_2)\}$.

■

For example, suppose that we want the minimum completion time of jobs with the additional lattice-linear constraint that $B_2(G) \equiv (G[1] = G[2])$. B_2 is lattice-linear with $forbidden(G, 1, B_2) \equiv (G[1] < G[2])$ and $forbidden(G, 2, B_2) \equiv (G[2] < G[1])$. By applying, Lemma 2.6, we get a parallel algorithm for the constrained version.

Note that the straightforward application of LLP may not give the most time-efficient parallel algorithm. The efficiency of the algorithm may depend upon $\alpha(G, j, B)$ chosen for the predicate B. We will later show such optimizations for many problems such as the shortest path algorithm.

2.3 Notation

We first go over the notation used in description of our parallel algorithms. Fig. 2.1 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. 2.1.

All other variables are derived directly or indirectly from G. G is an array of objects such that $G[j]$ is the state of thread j for a parallel program. 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. 2.1 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.

```
$P_j$: 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 2.1: LLP Parallel Program for (a) job scheduling problem using forbidden predicate (b) job scheduling problem using ensure clause and (c) the shortest path problems

2.4 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 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.

2.5 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 $B \equiv \forall (i,j) \in \mathit{pred} : G[j] \geq G[i] + 1$ is a lattice-linear predicate.

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

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

   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.

2.6 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 3

The Stable Marriage Problem

3.1 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. 3.1 shows an instance of the stable matching problem.

In this chapter, we first give the Gale-Shapley (GS) algorithm in Section 3.3. Section 3.4 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. We then give a parallel LLP version of the Gale-Shapley algorithm in Section 3.5. It also discusses the constrained stable matching problem, where in addition to men's preferences and women's preferences, there may be a set of additional *lattice-linear* constraints. For example, we may state 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. As another example, we may require the matching must contain some pairs called *forced pairs*, or must not contain some pairs called *forbidden pairs* [DdFdFS03].

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

3.2 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.

3.3 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.

```
input: A stable marriage instance: $mpref$, $rank$
output: man-optimal stable marriage
var
    $mList$: list of $1..n$; // list of men that are free initially includes all men;
    $partner$: array[$1..n$] of $0..n$ initially $partner[i] = 0$ for all $i$ //current fiance for woman $i$
    $G$: array[$1..n$] of $0..n$ initially $G[i] = 0$ for all $i$ //number of proposals made by man $i$
while ($mList$ is nonempty)
    remove a man $i$ from $mList$
    $G[i] := G[i] + 1$ //move to the next available top choice for man $i$
    $z := mpref[i][G[i]]$ // woman corresponding to that choice
    if ($partner[z] = 0$) // the woman was not engaged
        $partner[z] := i$
    else if ($rank[z][i] < rank[z][partner[z]]$)
        add $partner[z]$ to $mList$
        $partner[z] := i$
    else add $i$ to $mList$
endwhile;
return $G$;
```

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

Lemma 3.1 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.*

3.4 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 3.2 *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 3.3 *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 3.3. 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.

```
input: A stable marriage instance, initial proposal vector $I$
output: smallest stable marriage greater than or equal to $I$ (if one exists)
$forbidden(G, i)$ 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) $(j, \rho(G, i))$ is a blocking pair for $G$.
if $numw(I) > m$ then return null; // no stable matching exists
    else $G := I$;
while there exists a man $i$ such that $forbidden(G, i)$
    find the next woman $q$ in the list of man $i$ s.t. $i$ has the most preferred proposal to $q$ until $G$,
    if no such choice after $G[i]$ or the number of women proposed including $q$ exceeds $m$ then
        return null; // "no stable matching exists"
    else $G[i]$ := choice that corresponds to woman $q$;
endwhile;
return $G$;
```

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: $(\exists i : \exists k \leq G[i] : (z = mpref[i][k]) \wedge (rank[z][i] < rank[z][j]))$
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. Due to Lemma 2.6, we can use LLP algorithm to find the least stable marriage satisfying these 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 Summary

The following table lists all the algorithms discussed in this chapter.

Problem	**Algorithm**	**Parallel Time**	**Work**
Stable Marriage	Gale-Shapley	$O(n^2)$	$O(n^2)$
Stable Marriage	α	critical path length	$O(m^2)$
Stable Marriage	LLP	critical path length	$O(n^2)$

3.7 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. Give an instance of the stable marriage problem in which the maximally parallel LLP version of the algorithm is n times faster than the sequential Gale-Shapley algorithm.
4. 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.
5. Give an algorithm in which all men start with their least favored choice and improve their choices in the search of the stable marriage.

3.8 Bibliographic Remarks

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

Chapter 4

Parallel Reduce Algorithms

4.1 Introduction

In this chapter we illustrate some basic ideas in designing parallel algorithms by computing the maximum of an array of size n. These ideas would also be applicable to other operators such as computing the *sum*, or the *product* of an array. Similarly, we can compute the *OR*, and the *AND* of a boolean array using these ideas.

A sequential algorithm can solve this problem in $O(n)$ time and $O(n)$ work. We now show various parallel algorithms to solve the same problem. For simplicity, we assume that n is a power of 2, i.e., $n = 2^k$. We also assume that all elements in the array are distinct. Exercise 2 requires you to extend these algorithms when entries may not be distinct.

In Section 4.2, we first present a simple algorithm called ParReduce that takes $O(\log n)$ time and $O(n)$ work on an EREW PRAM. Section 4.3 describes a technique to combine the sequential algorithm with a parallel algorithm to reduce the total number of processors. Section 4.4 gives the LLP algorithm for the reduce operation. The LLP algorithm does not have the synchronization requirements of the Algorithm ParReduce. In Section 4.5, we consider a CRCW PRAM and present an algorithm that takes $O(\log \log n)$ time and $O(n)$ work.

4.2 Reduce

Suppose that we have an array A of size n and we want to compute the sum of all numbers. A simple *for* loop achieves this in $O(n)$ time. We now show a parallel algorithm that computes the sum in $O(\log n)$ time assuming that we have n cores. The algorithm is best viewed as constructing a binary tree such that leaves of the tree consists of the elements of the array and the internal nodes correspond to the sum of its children. Fig. 4.1 shows such a tree. Clearly, the root of the tree has the sum of the entire array. Since the tree is binary, the height of the tree is $O(\log n)$. Therefore, if the parallel algorithm can compute a level of the tree in parallel in $O(1)$ time, then the entire algorithm also has the time complexity of $O(\log n)$.

Algorithm ParReduce uses h to indicate the level of the tree. In the first iteration h equals 1 and the partial sums are stored in the first half of the array A. The entry $A[i]$ stores the sum of entries $A[2i-1]$ and $A[2i]$. In the second iteration, we reduce the size of the array that stores partial sum by an additional factor of 2. After $\log n$ iterations, the total sum is stored in $A[1]$. We note here that in the description of the algorithm we have the step $A[i] := A[2i-1] + A[2i]$ which is executed by all eligible i in parallel. This step may entail both reads and writes to the same memory location. For all such steps in the parallel algorithms, we follow the interpretation that all processes first do all the **reads** of the shared memory (into the private registers of the processors). When all reads are finished, then all writes are performed.

Table 4.1 shows values of A array after various values of h.

Algorithm ParReduce: A Parallel Algorithm for Reduce

```
1 input: $A$: array[1..$n$] of int // Assume $n$ is a power of 2 for simplicity
2 for $h := 1$ to $\log n$ do
3     forall $i$ in parallel do
4         if ($i \le n/2^h$) then
5             $A[i] := A[2i-1] + A[2i]$
6 return $A[1]$
```

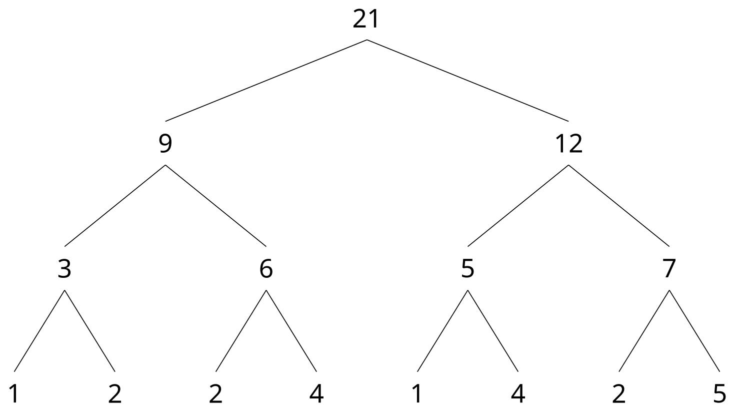


Figure 4.1: A Summation Tree

i	1	2	3	4	5	6	7	8
A	1	2	2	4	1	4	2	5
$h = 1$	**3**	**6**	**5**	**7**	1	4	2	5
$h = 2$	**9**	**12**	5	7	1	4	2	5
$h = 3$	**21**	12	5	7	1	4	2	5

Table 4.1: Entries during Execution of the Parallel Reduce Algorithm

The total time taken by this algorithm is $O(\log n)$ and the total number of comparisons made by this algorithm is $O(n)$. Ignoring for now the question of how various threads determine which comparison to make at what time step, we see that the total amount of work done by all threads is $O(n)$.

Whenever we design parallel algorithms, we should check if we require concurrent reads or concurrent writes by multiple processors in any step. Assuming that each of the processor knows when to read the labels of its children and when to write its own label, there is no conflict in computation and the algorithm can be implemented on a EREW PRAM.

It is also important to consider the *cost* of a PRAM algorithm. The cost of an algorithm is simply the product of the number of processors used by the algorithm and its time complexity. In this case, we are using n processors; therefore, the cost of this algorithm is $O(n \log n)$.

4.3 Accelerated Cascading Technique

We now show a technique that combines two different algorithms to reduce the cost (and sometimes the work) of many PRAM algorithms. Instead of applying the binary tree algorithm on the entire array of size n, we first divide the array into $O(n/\log n)$ blocks of $O(\log n)$. Also, we use only $O(n/\log n)$ processors so that each block can be assigned to a single processor. Now, each processor can compute the maximum of its block using a sequential algorithm. Since this step can be performed in parallel for different blocks, this step takes $O(\log n)$ time with the total cost of $O(n/\log n * n) = O(n)$. Now we have $O(n/\log n)$ numbers and we need to compute the maximum of these numbers. At this point, we apply the binary tree based algorithm. It will take us $O(\log n)$ time with $O(n/\log n)$ processors. Hence, the total cost of the second step would only be $O(n)$. Combining the time, the work and the cost of each of the steps, we get $T(n) = O(\log n)$, $W(n) = O(n)$, and $C(n) = O(n)$.

It can be shown that $O(\log n)$ time is required on EREW PRAM. Also, since the work matches that of a sequential algorithm, we have an optimal work-time algorithm on EREW PRAM.

4.4 LLP-Reduce Algorithm

The parallel reduce algorithm has two disadvantages: it destroys the original input array and it requires that all threads synchronize after each iteration of the outer for loop. We can take care of the first problem by making a copy of the original array. This step can be carried out in $O(1)$ time with the number of cores equal to the size of the array. We now show a LLP based algorithm to compute the sum that takes care of both of these problems. We again assume that the array size of A is a power of 2 for simplicity. We would like to compute an array G of size $n-1$ such that $G[1]$ contains the sum of the entire array, $G[2]$ contains the sum of the first half of the array, $G[3]$ contains the sum of the second half of the array, and so on.

Fig. 4.2 shows the arrays G and A for n equal to 8. Note that for $1 \le j < n/2$, the children of node $G[j]$ are the nodes $G[2j]$ and $G[2j+1]$. For $n/2 \le j < n$, the children of nodes $G[j]$ are the nodes $A[2j-n+1]$ and $A[2j-n+2]$. Then, the least vector G that satisfies

$$B_{sum}(G) \equiv (\forall j : 1 \le j < n/2 : G[j] \ge G[2j] + G[2j+1]) \wedge (\forall j : n/2 \le j \le n-1 : G[j] \ge A[2j-n+1] + A[2j-n+2])$$

is equal to the recursive sum. The array $G[1..n/2-1]$ contains the sum of the left and the right children which are in G itself. The array range $G[n/2..n-1]$ contains the sum of elements of A viewed as the leaves of the tree.

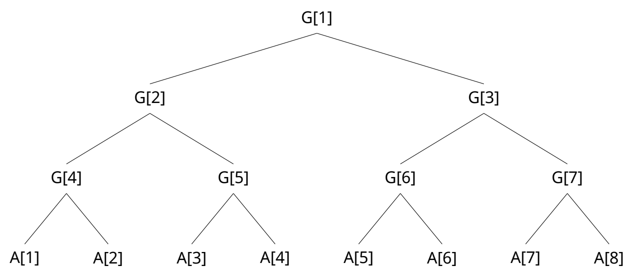


Figure 4.2: A Summation Tree for LLP Reduce

The parallel algorithm LLP computes G in $O(\log n)$ time. Instead of using $+$, we can use any other associative operator such as *min* or *max*.

Algorithm LLP-Reduce: A Parallel LLP Algorithm for Reduce

```
1 $P_j$:
2 **input**: $A$: array[1..$n$] of int // Assume $n \geq 2$ is a power of 2 for simplicity
3 **shared var**: $G$: array[1..$n-1$] of int
4 **init**: $G[j] := -\infty$
5 **ensure**: $G[j] \geq G[2j] + G[2j+1]$ if $1 \leq j < n/2$
6 **ensure**: $G[j] \geq A[2j-n+1] + A[2j-n+2]$ if $n/2 \leq j < n$
```

Table 4.2 shows values of the array G after various values of t, the iteration number. For simplicity, we consider a maximally parallel execution in which any entry that can change does so at any iteration. However, the final answer remains the same even if the execution is not maximally parallel. There is no synchronization required for this algorithm.

i	1	2	3	4	5	6	7
G	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$t=1$	$-\infty$	$-\infty$	$-\infty$	3	6	5	7
$t=2$	$-\infty$	9	12	3	6	5	7
$t=3$	21	9	12	3	6	5	7

Table 4.2: Entries during a maximally Parallel Execution of the LLP-Reduce Algorithm

Although our discussion was for the sum, it is clear that any associative operation such as the product, the min, the max, or the boolean operator such as the AND, the OR, or the XOR can be used.

4.5 A Fast Parallel Common CRCW Algorithm

We now give a faster algorithm on common CRCW PRAM. This algorithm requires $O(n^2)$ processors. We have a boolean array *isBiggest* of size n which is initialized to true for all i. We use one processor for every distinct i and

j which allows us to compare every distinct $A[i]$ and $A[j]$ in one time step. The processor assigned to (i, j) writes *false* on $isBiggest[i]$ if the entry for $A[j]$ is bigger than $A[i]$. Note that multiple writers may access $isBiggest[i]$ in one time step. However, all of them would be writing the common value *false* to that memory location. This step assumes common CRCW PRAM.

Algorithm FindingBiggest: A Common CRCW Parallel Algorithm for Finding the Maximum

```
1 var $isBiggest$: array[1..$n$] of boolean;
2 forall $i$ in parallel do:
3       $isBiggest[i]$ := true;
4 forall $i, j$: $i \neq j$: in parallel do:
5       if ($A[j] > A[i]$) then $isBiggest[i]$ := false;
6 forall $i$: in parallel do:
7       if $isBiggest[i]$ then output (max = $A[i]$);
```

We have: $T(n) = O(1)$, and $W(n) = O(n^2)$.

How can we reduce the work complexity? We now show a technique that allows us to reduce the work complexity on a common CRCW machine. In binary trees we were comparing two labels using one processor and one time step. From the fast-parallel algorithm, we know that we can find the maximum of $\sqrt{n}$ nodes in $O(1)$ time using n processors. Can we use this fast parallel algorithm to increase the branching but reduce the height of the tree? For this section assume that $n = 2^k$ and $k = 2^m$. As a concrete example, let $n = 256 = 2^8$. Here $k = 8 = 2^3$. At the first level, we will have a branching of 16, i.e., the root would have 16 subtrees each of size 16. If we knew the maximum of each of the subtrees, then in $O(1)$ step we can compute the overall maximum in $O(1)$ steps. This step will require $16^2 = 256$ processors. Let us now determine the way to compute the maximum of the subtree with 16 labels. Each of these subtrees would have 4 subtrees of size 4. Using 16 processors, we can compute maximum of this tree. Since there are 16 subtrees, we need 16 times 16 processors at the second level. Since the height of the tree is $\log\log n$, the total number of time steps is $O(\log\log n)$. At each step, we use n processors. This gives us: $T(n) = O(\log\log n)$, and $W(n) = O(n \log\log n)$.

We now show that the previous algorithm that is not work optimal can be made work-optimal by the technique of "cascading" multiple algorithms. Note that we have an additional factor of $O(\log\log n)$ in the work. To eliminate this factor, we first partition our array of size n into $O(n/\log\log n)$ blocks each of size $O(\log\log n)$. If our array was only of size $O(n/\log\log n)$ and we apply the doubly logarithmic height tree algorithm, we would have the desired work complexity of $O(n)$. So now we consider how to compute the maximum of a block of $\log\log n$ numbers. Since this block is small in size, we can compute the maximum using a single processor in $O(\log\log n)$ time. Hence, the entire algorithm is as follows:

Step 1: run sequential algorithm on each of the blocks of size $O(\log\log n)$. This takes time equal to $O(\log\log n)$, and work equal to $O(n)$. We now have an array of size $n/\log\log n$.

Step 2: We now run the doubly logarithmic height tree algorithm. The total work is $O(n)$, and the total time is $O(\log\log n)$.

When we combine the time complexity of the work complexity of these two steps, we get the desired time complexity of $O(\log\log n)$ and the work complexity of $O(n)$.

4.6 Summary

Table 4.3 summarizes all the algorithms discussed in this chapter.

Algorithm	Work	Time	PRAM
Sequential	O(n)	O(n)	
Binary Tree Based	O(n)	O($\log(n)$)	EREW
All-pair Comparison Algorithm	O(n^2)	O(1)	CRCW
Doubly Logarithmic Tree Algorithm	O($n\log(\log(n))$)	O($\log(\log(n))$)	CRCW
Cascaded Algorithm	O(n)	O($\log(\log(n))$)	CRCW

Table 4.3: Time Complexity and Work Complexity of Computing Maximum

4.7 Problems

1. Generalize the Algorithm LLP-Reduce for all values of n (i.e., n may not be a power of 2).
2. Suppose that an array does not have all elements that are distinct. Show how you can use any algorithm that assumes distinct elements for computing the maximum to solve the problem when elements are not distinct.
3. Give work-optimal algorithms to compute OR of a boolean array on EREW and common CRCW PRAM.
4. Suppose that instead of PRAM you have a network of processors connected by a two dimensional square mesh. Assuming that every hop on the network takes one unit of time, give an algorithm with $O(\sqrt{n})$ time complexity to compute the maximum in the network.
5. Repeat the previous problem with the topology of a d-dimensional hypercube (i.e. $n = 2^d$ for some integer d).

4.8 Bibliographic Remarks

Parallel algorithms for computing maximum or minimum of an array are presented in most books on parallel computation (e.g. [JáJ92]).

Chapter 5

Parallel Prefix Algorithms

5.1 Introduction

In this chapter we discuss the *parallel-prefix* operation. The parallel-prefix operation is also called *scan* operation and we will use these terms interchangeably. The reduce operation applies an *associative* operation, such as the sum, the product, the min or the max, to an array of size n to get an answer that corresponds to the sum, the product, the min or the max of the entire array. The parallel-prefix operation computes the answer for all prefixes of the array.

This chapter is organized as follows. Section 5.2 gives an algorithm for the *parallel-prefix* operation that requires $O(n \log n)$ work. Section 5.3 presents work-optimal algorithms for the parallel-prefix operation. It first gives a simple recursive parallel algorithm for the scan operation. It then presents an iterative version due to Blelloch [Ble90]. Section 5.4 gives an LLP version of the parallel-prefix operation. Finally, Section 5.5 presents some applications of the parallel-prefix operation.

5.2 A Parallel Prefix Sum algorithm with $O(n \log n)$ work

Suppose that we have an array A of size n. We want to compute C, the sum of all possible prefixes of A, formally defined as follows:

$$C[k] = \sum_{i=0}^{i \le k} A[i]$$

For example, suppose $A = [1, 4, 9, 16]$. Then, $C = [1, 5, 14, 30]$. This operation called **parallel prefix** or **scan** operation is a fundamental building block for many parallel algorithms.

There is an easy linear time sequential algorithm for this problem.

Algorithm SeqPrefix: A Sequential Algorithm for Prefix-Sum

```
1 for (int $i = 0; i < n; i++$)
2       $C[i] := A[i]$
3 for (int $i = 1; i < n; i++$)
4       $C[i] := C[i-1] + A[i]$
```

This algorithm taken $O(n)$ time and requires $O(n)$ work.

The following parallel algorithm reduces the overall time by computing the sum in a binary tree like fashion rather than one at a time as done by the sequential algorithm.

Algorithm ParPrefix: A Parallel Algorithm for Prefix-Sum with $O(n \log n)$ work

```
forall $i \in [0, n-1]$: in parallel do
    $C[i] := A[i]$;
for (int $d = 1; d < n; d = 2d$)
    forall $i \in [1, n-1]$: in parallel do
        if $(i - d \geq 0)$
            $C[i] := C[i] + C[i-d]$;
```

After every iteration of the second parallel for loop, $C[i]$ holds sum of all $A[k]$ such that $i - 2d + 1 \leq k \leq i$ for any d. Table 5.1 shows values of C array after various values of d.

A	3	2	4	2	1	3	5	2
C, $d = 1$	3	**5**	**6**	**6**	**3**	**4**	**8**	**7**
C, $d = 2$	3	5	**9**	**11**	**9**	**10**	**11**	**11**
C, $d = 4$	3	5	9	11	**12**	**15**	**20**	**22**

Table 5.1: Entries during Execution of the Parallel Prefix Algorithm

One needs to be careful in implementing the above high-level algorithm. There needs to be proper synchronization of threads for reading and writing in-place. For example, the pseudo-code for implementing parallel-prefix on a GPU is as follows. Observe that the *for* loop with index i is not in the pseudo-code. This is because the code is executed by all processes P_i.

Algorithm ParPrefixGPU: A Parallel Prefix Algorithm on GPU with $O(n \log n)$ work

```
$P_i$::
$C[i] := A[i]$;
for (int $d = 1; d < n; d = d * 2$) // in sequence
    if $(i \geq d)$ $val = C[i-d]$;
    barrier();
    if $(i \geq d)$ $C[i] = C[i] + val$;
    barrier();
endfor;
```

This algorithm requires $O(\log n)$ time but is not work-optimal. It requires $O(n \log n)$ work whereas the sequential algorithm requires only $O(n)$ work. We now present a work-optimal algorithm for this problem.

5.3 A Work-Optimal Parallel Prefix Sum Algorithm

We now show an algorithm that computes parallel prefix with $O(n)$ work. We give two versions of this algorithm. First, we give a recursive version. We will use arrays indexed starting from 1 and assume that the input array has size n that is a power of 2, i.e., let $n = 2^k$ for some integral k. An example of the recursive algorithm Recursive-ParPrefix is shown in Table 5.2. We first take care of the base case when n equals 1. Otherwise, we copy the array A into B such that the array B has the size that is half of the size of A and every two elements of A are added to one element of B. We make the recursive call to parallel prefix of B, getting the result in array D. Once we have recursive solution in D, we can compute the solution in C by doing a case analysis on index i.

Algorithm Recursive-ParPrefix: A Recursive Parallel Prefix Algorithm with $O(n)$ work

```
1 if $(n = 1)$ then $C[1] = A[1]$ else
2     for $i := 1$ to $n/2$ in parallel do
3         $B[i] := A[2i - 1] + A[2i]$;
4     $D :=$ recursive-ParPrefix$(B[1], B[2], ..., B[n/2])$;
5     for $i := 1$ to $n$ in parallel do
6         if $(i = 1)$ then $C[1] := A[1]$;
7         else if $even(i)$ then $C[i] := D[i/2]$
8         else if $odd(i)$ and $(i > 1)$ then $C[i] := D[i/2] + A[i]$
```

A	1	2	3	4	5	6	7	8
B	3	7	11	15				
D	**3**	**10**	**21**	**36**				
C	1	**3**	6	**10**	15	**21**	28	**36**

Table 5.2: Example of a Recursive Scan

.

Let us compute the time required by recursive parallel-prefix. In one parallel time step, the input size is reduced by a factor of 2. Hence, the algorithm takes $O(\log n)$ time. To compute the total work required by the above algorithm, we use the following recurrence relations.

$$W(n) = W(n/2) + O(n).$$

$$W(1) = O(1).$$

Therefore, $W(n) = O(n)$.

We now show a non-recursive version of the algorithm which computes the *exclusive* scan of an array. The inclusive scan of an array A returns for every index i, the sum of all entries before i including i, whereas the exclusive scan returns the sum of all entries in prefix excluding itself. For example, Table 5.3 shows a simple inclusive and exclusive scan of an array A.

Going from exclusive scan to inclusive parallel prefix sum can be performed in one parallel step.

The work-optimal algorithm can be viewed in a tree like fashion except that we require two sweeps of the tree — one in the upward direction and one in the downward direction.

An example is shown below. Suppose that the array A equals $[01, 02, 03, 04, 05, 06, 07, 08]$. For simplicity, we have used $A[i] = i$. Now, the upward sweep is shown in Fig. 5.1.

The upward sweep can be understood in terms of the tree as follows. For any vertex v, we compute:

$$sum[v] = sum[L[v]] + sum[R[v]]$$

Input Array A	01 02 03 04 05 06 07 08
Inclusive Scan of A	01 03 06 10 15 21 28 36
Exclusive Scan of A	00 01 03 06 10 15 21 28

Table 5.3: Example of an Inclusive and Exclusive Scan

.

Algorithm Blelloch-Scan: Blelloch's Parallel Prefix Algorithm with $O(n)$ work

```
**forall** $i$ in $[0, n-1]$: in **parallel** do
    $B[i] := A[i]$;
**for** $h := 0$ to $\log n - 1$ do // upward sweep: reduce operation
    **forall** $i$ from 0 to $n-1$ in steps of $2^{h+1}$ in **parallel** do
        $B[i + 2^{h+1} - 1] := B[i + 2^h - 1] + B[i + 2^{h+1} - 1]$;
$B[n-1] := 0$;
**for** $h := \log n - 1$ down to 0 do // downward sweep
    **forall** $i$ from 0 to $n-1$ in steps of $2^{h+1}$ in **parallel** do
        $LeftValue = B[i + 2^h - 1]$; // store the value of the left child
        $B[i + 2^h - 1] = B[i + 2^{h+1} - 1]$; // Left child gets the value of this node
        $B[i + 2^{h+1} - 1] = LeftValue + B[i + 2^{h+1} - 1]$; // Right child gets the sum
```

```
01  02  03  04  05  06  07  08
    03      07      11      15
            10              26
                            36
```

Figure 5.1: Upward Sweep of Blelloch Scan

where $L[v]$ and $R[v]$ are the left and right children of v.

We now replace the root by 0 and start the downward sweep. The downward sweep can be understood in terms of the tree as follows.

$$scan[L[v]] = scan[v]$$

$$scan[R[v]] = sum[L[v]] + scan[v]$$

The upward sweep followed by the downward sweep is presented in Fig. 5.2.

Figure 5.3 shows the values computed using upward scan and the downward scan in a binary tree. The upward scan values are shown inside circles and the downward scan values are shown outside circles.

We now show the correctness of the above procedure. We say that vertex x *precedes* vertex y if x appears before y in the preorder traversal of the tree.

```
01  02  03  04  05  06  07  08
    03      07      11      15
            10              26
                            36
                            00
            00              10
     00     03      10      21
 00  01  03 06  10  15  21  28
```

Figure 5.2: Blelloch Scan

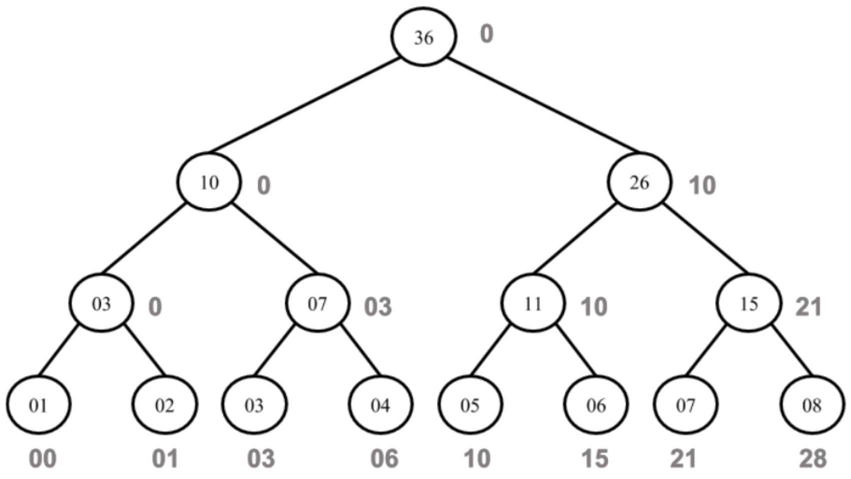


Figure 5.3: Blelloch Scan Algorithm applied to Parallel Prefix Puzzle

Theorem 5.1 *After a complete down-sweep, each vertex of the tree contains the sum of all the leaf values that precede it.*

Proof: The proof is inductive from the root: we show that if a parent has the correct sum, both children must have the correct sum. The root has no elements preceding it, so its value is the identity element which is zero for sum.

The left child of any vertex has exactly the same leaves preceding it as the vertex itself. This is because the preorder traversal always visits the left child of a vertex immediately after the vertex. By the induction hypothesis, the parent has the correct sum, so it need only copy this sum to the left child. The right child of any vertex has two sets of leaves preceding it, the leaves preceding the parent, and the leaves at or below the left child. Therefore, by adding the parent's down-sweep value, which is correct by the induction hypothesis, and the left-child's up-sweep value, the right-child will contain the sum of all the leaves preceding it.

■

5.4 LLP based Parallel Prefix-Sum

Let A be an array of n numbers. Assume that n is a power of two for simplicity. We would like to compute parallel (exclusive) prefix sum of A. We compute the parallel prefix sum in the array G using ideas in Blelloch scan. Our LLP based implementation will use additional space but has less synchronization. Let the input array be A of size n. When n is a power of 2, the summation tree of the reduce operation has $n-1$ nodes. We create an additional array S of size $n-1$ that simulates the upward scan using Algorithm LLP-Reduce. We now use another LLP algorithm to simulate the downward scan. We model the downward scan using an array G of size $2n-1$ as follows. The first $n-1$ elements of the array compute the scan values for the intermediate nodes. The last n entries give us the required values for exclusive scan. Similar to LLP Algorithm for Reduce, we use the indexing scheme of a perfect binary tree such that for any node with index i, its left child is given by $2i$, its right child by $2i+1$, and its parent by $i/2$ (for non-root nodes).

The LLP algorithm initializes every element of G to $-\infty$. Each element $G[j]$ is updated exactly once when the forbidden predicate is true for index j. Since $G[1]$ corresponds to the root of the tree for the downward scan, we set $G[1]$ to 0 as required by the first ensure clause of the LLP. The left child of any node is even and is updated by the second ensure clause by copying the value of the parent. All the left children are characterized by their indices being even and thus setting $G[j]$ to $G[j/2]$ copies the value of the parent. The right child of any node is updated by the third and the fourth ensure clauses as the sum of the parent value and the S value for its left sibling. The fourth clause is used for the last level of the tree when the sum of the subtree at that node is simply equal to A value.

Algorithm LLPScan: A Parallel LLP Algorithm for Scan

```
$P_j$: Code for thread $j$
// Assume $n$ is a power of 2 for simplicity
input: $A$: array[1..$n$] of int
    $S$: array[1..$n-1$] of int // summation tree
output:$G$:array[1..$2n-1$] of int such that $G[i] = \sum_1^{i-n} A[i-n]$ for $i \geq n$.
init: $G[j] := -\infty$
ensure: $G[j] \geq 0$ if $j = 1$
ensure: $G[j] \geq G[j/2]$ if $j$ is even
ensure: $G[j] \geq S[j-1] + G[j/2]$ if $j$ is odd and $j < n$
ensure: $G[j] \geq A[j-n] + G[j/2]$ if $j$ is odd and $j > n$
```

5.5 Applications

Although our description of the parallel prefix operation was based on the sum operator, it is applicable to any *associative* operation such as the product, min, or max. Many problems on arrays can be solved using the parallel prefix operation one or more times. We give some examples below.

- *Array Compaction*: Suppose we are given an array A of integers. Our problem is to create another array B consisting of only those elements that specify certain given *filter* predicate. As a simple example, we may want the array B to contain only those entries in A that are divisible by 5. If for every entry in A that is divisible by 5, we could determine the number of entries that appear in A before this entry that are also divisible by 5, we are done. We can store that entry at the right index in B. To solve this problem, In $O(1)$ time we create an additional array C such that $C[j]$ equals 1 if $A[j]$ satisfies the filter predicate and 0 otherwise. The parallel prefix of the array C gives us the index where the entries of A should go in the array B.

- *Sparse Dot Product*: Suppose that we have a vector A of size n such that most entries are zero. Instead of storing the entire vector, we can store a *sparse* vector B such that $B[i]$ corresponds to i^{th} nonzero entry and it stores $(j, A[j])$ where j is the index of i^{th} nonzero entry. Suppose that we want to compute the dot product of A with a dense vector v. With n processors, we can easily compute the dot product in $O(\log n)$ time. If we had a sparse matrix A, and we wanted to computes Av, then by computing dot product of different rows of A with v in parallel, we can compute Av also in $O(\log n)$ time.

- *Maximum Subsequence Problem*: Suppose that you are given a nonempty array A of integers of size n. The array consists of both positive and negative integers. We give an algorithm that gives a nonempty subsequence in the array with the largest value. Formally, we are required to find

  $$max_{i,j \geq i} \sum_{k=i}^{j} A[k]$$

  The algorithm is:

  - First reduce-max and return this if the output is negative (this handles the edge case when all elements are negative).

- Compute parallel prefix sum, and call the resulting array B.
- Compute parallel postfix max on B (this can be done with Blelloch scan on a reversed B and call the resulting array C.
- Create a new array D, and for each index i in parallel write $D[i] := C[i] - B[i] + A[i]$.
- Finally output the result from reduce-max on D.

This procedure has time complexity $O(\log n)$ and work complexity $O(n)$ in the CREW PRAM model.

Below is the psuedo-code:

Algorithm Maximum-Subsequence: Parallel Maximum Subsequence

```
1 $m$ := reduce_max($A$)
2 if $m < 0$ then output m
3 $B$ := prefix_sum($A$)
4 $C$ := postfix_max($B$)
5 $D$ := allocate_new_array($n$)
6 foreach $i \in \{1, 2, ..., n\}$ in parallel do
7     $D[i] := C[i] - B[i] + A[i]$
8 end
9 output reduce_max($D$)
```

5.6 Summary

The following table lists all the algorithms discussed in this chapter.

Problem	**Algorithm**	**Parallel Time**	**Work**
Parallel Prefix	Hillis-Steele	$O(\log n)$	$O(n \log n)$
Parallel Prefix	Recursive	$O(\log n)$	$O(n)$
Parallel Prefix	Blelloch	$O(\log n)$	$O(n)$
Parallel Prefix	LLP	$O(\log n)$	$O(n)$

5.7 Problems

1. Suppose that we have a mesh of size $\sqrt{n}$ rows and $\sqrt{n}$ columns. An array of size n is distributed on this mesh in a row-major form. Design an algorithm such that every node has the parallel prefix at the end of the algorithm. Your algorithm should not take more than $O(\sqrt{n})$ time and $O(n^{3/2})$ cost.

2. Consider a mesh of size $n^{1/3}$ rows and $n^{1/3}$ columns. An array of size n is distributed on this mesh in a row-major form such that each node has $n^{1/3}$ elements. Give a work optimal algorithm that takes $O(n^{1/3})$ time.

3. Give an algorithm to compute prefix-sum on a hypercube of n nodes with $O(\log n)$ time complexity. What is the work complexity of your algorithm?

4. Give a parallel-prefix algorithm to divide the array into two parts: the part with odd entries followed by the part with even entries.

5.8 Bibliographic Remarks

Parallel Prefix algorithms are presented in most books on parallel computation (e.g. [JáJ92]). The reader will find many applications of the parallel prefix computation in [Ble90].

Chapter 6

Sorting Algorithms

6.1 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.

It is easy to generalize this divide-and-conquer strategy for p processors by dividing the range of indices into contiguous $p+1$ ranges. In the second time step the range of indices is reduced by a factor of $p+1$. Therefore, we have $\log n/\log(p+1)$ parallel algorithm on p processors. In an unsorted array, search would require n/p time in the worst case. This difference in search time motivates sorting of arrays used in software systems.

This chapter is organized as follows. Section 6.2 discusses sorting algorithms that are 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 and $O(n)$ parallel time in the worst case. Section 6.3 discusses sequential and parallel versions of *mergesort*. Section 6.4 discusses sequential and parallel versions of *quicksort*. Since Merge Sort and Quick Sort are based on divide-and-conquer paradigm, they are revisited in Chapter 12. Section 6.5 discusses bitonic sort. All these sorting algorithms are based on comparison of two entries in the array. Finally, Section 6.6 describes a sorting algorithm that is not based on comparison of two entries.

6.2 Sorting Algorithms based on Swapping Consecutive Entries

Let A be an array of distinct integers. Our goal is to sort the array. This problem at face 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 we can 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: A High-Level LLP Sorting algorithm based on immediate swaps

```
1 $P_j$: Code for thread $j$
2 **var** $A$: array[1..$n$] of int; // input array
3 $G$: array[1..$n$] of 0..$n-1$ initially $G[i] = 0$ for all $i$// abstract variable: just for correctness
4     **forbidden(j)**: $A[j] > A[j+1]$ // $G[j]$ is missing at least one inversion
5         **advance(j)**: $swap(A[j], A[j+1])$ // increment $G[j]$
```

Observe that in this algorithm, in a parallel setting, it is important that the statement $swap(A[j], A[j+1])$ is executed atomically whenever $A[j] > A[j+1]$, for $j = 1..n-1$. 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: Bubble Sort B

```
1 **var** $A$: array[1..$n$] of int; // input array
2 $found$: boolean;
3 **repeat**
4     $found$ := false;
5     **for** $j := 1$ to $n-1$ do
6         **if** $A[j] > A[j+1]$
7             $found := true$;
8             $swap(A[j], A[j+1])$
9 **until** ($found$ = false);
```

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.

Algorithm Insertion-Sort: Insertion Sort B

```
var $A$: array[1..$n$] of int; // input array
for $i := 1$ to $n$ do
// Assuming that $A[1..i-1]$ is sorted, ensure that $A[1..i]$ is sorted
    for $j := i-1$ to 1 do
        if $A[j] > A[j+1]$
            $swap(A[j], A[j+1])$
```

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 adapt Algorithm LLP-Sort1 for parallel execution. In particular, we saw that the algorithm required atomicity of checking for the forbidden predicate and swaps. We can avoid this expensive synchronization by scheduling the forbidden indices as follows. In the first phase only odd indices check for the forbidden predicate and swap with the right entry if necessary. In the second phase only even indices carry out this operation. These phases are executed alternatively until there is no forbidden index left. This algorithm, called Odd-Even Sort, is similar to the sequential bubble sort, in that it is based on swapping out of order entries. It is shown in Algorithm OddEven-Sort.

Algorithm OddEven-Sort: Odd-Even Sort B

```
var $A$: array[1..$n$] of int; // input array
repeat
    $found$ := false;
    forall $odd(j), j \in [1..n-1]$ in parallel do
        if $(A[j] > A[j+1])$
            $found := true$;
            $swap(A[j], A[j+1])$
    forall $even(j), j \in [1..n-1$ in parallel do
        if $(A[j] > A[j+1])$
            $found := true$;
            $swap(A[j], A[j+1])$
until $(\neg found)$;
```

It is easy to see that the repeat loop is executed at most n times. Hence, the algorithm takes $O(n)$ time and $O(n^2)$ work. An example is shown in Fig. 6.1.

6.3 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

8 1 4 6 9 5 2 7; original unsorted array
1 8 4 6 5 9 2 7; all odd entries are swapped in parallel with next entries, if inverted
1 4 8 5 6 2 9 7; even entries are swapped, if inverted
1 4 5 8 2 6 7 9; odd entries are swapped
1 4 5 2 8 6 7 9; ; even entries are swapped, if inverted
1 4 2 5 6 8 7 9; odd entries are swapped, if inverted
1 2 4 5 6 7 8 9; even entries are swapped, if inverted

Figure 6.1: An Execution of Odd-Even Sort

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: MergeSort

```
MergeSort($A, low, high$)
**if** $low < high$ then
    $mid = \lfloor (lo + high)/2 \rfloor$ ;
    $B :=$ MergeSort($A, low, mid$) **in parallel** with $C :=$ MergeSort($A, mid + 1, high$)
    return $MergeTwo(B, C)$;
```

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 arrays

```
**var**
    $B$: array[1..$m$] of int; // input array
    $C$: array[1..$n$] of int; // input array
    $D$: array[1..$m + n$] of int; // output array
int $i, j, k := 0, 0, 0$;
**while** $(i < m)$ and $(j < n)$ do
    **if** $(B[i] < C[j])$ then $\{D[k] = B[i]; i++;\}$
    **else** $\{D[k] = C[i]; j++;\}$
    $k++$;
**while** $(i < m)$ $\{ D[k] = B[i]; i++; k++; \}$
**while** $(j < n)$ $\{ D[k] = C[j]; j++; k++; \}$
```

We now devise a parallel algorithm to merge two sorted arrays. For simplicity, we assume that all elements are unique. This algorithm is based on finding the location in D where each element of B and C will appear. If every element in B and C determines its rank in parallel, it can write that value at the correct location in D.

Define the rank of any element x as the number of elements in B and C that are less than x, i.e., $rank(B[i], D)$ = the number of elements in B less than $B[i]$ + $rank(B[i], C)$. The first number is simply i (for arrays that start with index 0). The second number can be determined using binary search in $O(\log(n))$ time. So the overall time is:

$T(n) = O(1) + O(\log(n))$. This algorithm requires concurrent reads but no concurrent writes, so a *CREW* PRAM is sufficient.

6.4 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: QuickSort

```
1 procedure QuickSort($A, low, high$)
2 if $low < high$ then
3     $pivot := choosePivot()$
4     $(p, q) := partition(A, pivot, low, high)$
5     QuickSort($A, low, p$) in parallel with QuickSort($A, q, high$)
```

We describe a slight variant of the Quicksort algorithm in which we partition the array into three parts. Algorithm QuickSort 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.

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 12.

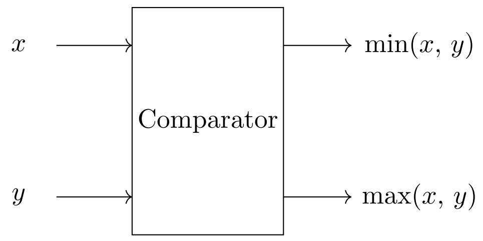
6.5 Bitonic Sort

Bitonic sort is a parallel sorting algorithm. One of its nice properties is that it is an *oblivious* sorting algorithm. A sorting algorithm is oblivious if its control flow is independent of the actual data. For example, an oblivious sorting

Algorithm Three-Way-Partition: Three-Way-Partition

```
function Partition($A, pivot, low, high$) returns (int, int)
    $k := high$;
    if $k > low$ then
    $p := low$
    $q := low$
    while $q < k$ do
        if $A[q] < pivot$ then
            swap( $A[p], A[q]$)
            $p := p + 1$; $q := q + 1$;
        else if ($A[q] > pivot$) then
            $k := k - 1$
            swap( $A[q], A[k]$)
        else
            $q := q + 1$;
    return $(p, q)$
```

algorithm will take the same steps if the input array is already sorted! Independence of the control flow from the actual data is nice because there is the execution of each thread is deterministic and hence it is easy to map the control flow onto a GPU (and to build hardware for sorting). Oblivious sorting algorithms can be built using a network of *comparators*. A comparator can be considered as a block with two inputs x and y and two outputs lo and hi such that lo is minimum of x and y and hi is the maximum of x and y. Let us denote the comparator as a function that returns a pair (lo, hi). For example, $comparator(3, 5)$ will return $(3, 5)$ and $comparator(5, 3)$ will also return $(3, 5)$. Pictorially, a comparator can be shown as



By combining comparators, we can create a *sorting network* that produces a sorted output from any input.

Odd-even sort that we studied earlier is also an oblivious sorting algorithm. The sorting network for odd-even sort for eight elements is shown in Fig. 6.2.

Before we give the details of bionic sort, we describe an important principle, called *zero-one principle*, that is quite useful in analysis of algorithms based on sorting networks. The zero-one principle says that if a sorting network gives correct output on all inputs that consist of only zeroes and ones, then that sorting network will give correct output on all inputs. If we can show zero-one principle, then analyzing sorting networks becomes much simpler; we only need to consider binary inputs. To prove the zero-one principle, we analyze how a comparator works when instead of presenting input $x = (x_1, x_2)$, it is presented with input $(f(x_1), f(x_2))$. Let f be any *monotonic* function, i.e., $x_1 \leq x_2$ implies that $f(x_1) \leq f(x_2)$. It is clear that f distributes over min and max, i.e.,
$f(min(x_1, x_2)) = min(f(x_1), f(x_2))$, and
$f(max(x_1, x_2)) = max(f(x_1), f(x_2))$.

Hence, when $(f(x_1), f(x_2))$ is presented to a comparator, it outputs $f(min(x_1, x_2))$ and $f(max(x_1, x_2))$.

By applying induction on the number of comparators, it can be shown that when the input sequence x is applied and any wire in the network has value x_i, then it has the value $f(x_i)$ when the input sequence $f(x)$ is applied.

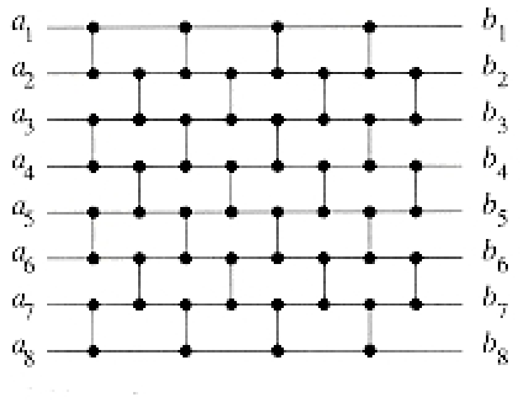


Figure 6.2: Sorting Network for Odd-Even Sort

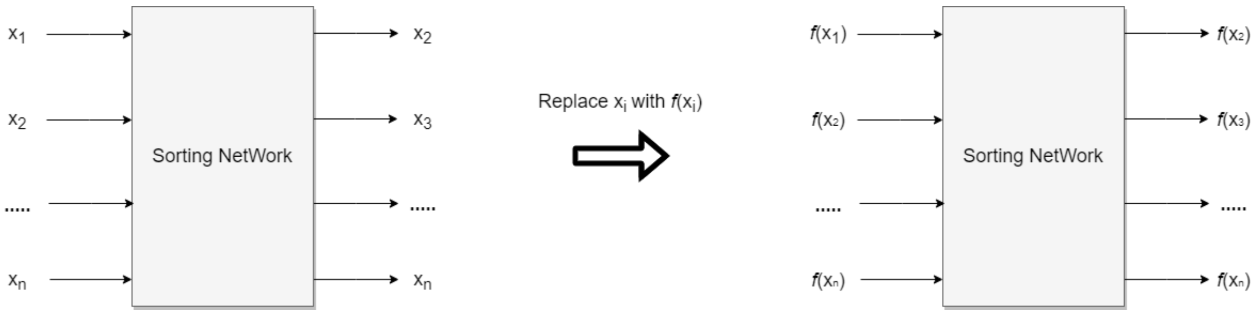


Figure 6.3: Replacing inputs of a sorting network

Armed with the above observation, we show the zero-one principle. Suppose that a network sorts all zero-one sequences correctly but does not sort a sequence of arbitrary numbers. Let that sequence be $(x_1, x_2, \ldots x_n)$ such that the network puts x_i higher than x_j even though x_i is smaller than x_j. We now show a zero-one sequence in which the network gives wrong answers. We use a monotonic function f that maps every element in x to 0 if it is less than or equal to x_i and 1 otherwise. Since x_i is placed higher than x_j, from our earlier observation $f(x_i)$ is placed higher than $f(x_j)$ which is incorrect because 0 is placed higher than 1. Hence, we conclude that there is no sequence of arbitrary elements that is sorted incorrectly by the network.

Bitonic sort is in some sense similar to Mergesort, in that it also merges two sorted sequences. The merge in bitonic sort, called bitonic-merge, has a very simple parallel implementation.

Let us begin with the definition of a bitonic sequence. A bitonic sequence is more conveniently viewed as a circular array. We define a range $A[l \ldots h]$ in a circular array A as follows. If l is less than or equal to h, then this range equals

$$[A[i] | l \le i \le h].$$

If l is greater than h, then this range equals $[A[i] | l \le i \le n-1]$ concatenated with $[A[i] | 0 \le i \le h]$. In other words, we traverse all the indices going from l to h by incrementing the index by 1 modulo n. For example, if $A = [2, 4, 6, 8]$, then $A[2..3] = [6, 8]$, but $A[3..2] = [8, 2, 4, 6]$.

A sequence of numbers A of size n is called *ascending* if it is sorted in ascending order,i.e.,

$$\forall i : 0 \le i < n-1 : A[i] \le A[i+1]$$

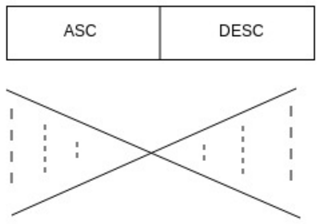


Figure 6.4: Bitonic Merge Lemma

It is *descending* if

$$\forall i : 0 \le i < n-1 : A[i] \ge A[i+1]$$

Definition 6.1 (Bitonic Sequence) *A sequence $A[0], A[1], \ldots A[n-1]$ is* bitonic *if there exists two indices l and h such that $A[l..h]$ is an ascending sequence and $A[h..l]$ is a descending sequence.*

It can be easily verified that $A[l]$ corresponds to a minimum value in array A and $A[h]$ corresponds to a maximum value.

For example, $A = [4, 3, 1, 5, 8, 9, 7, 4]$ is a bitonic sequence because $A[2..5] = [1, 5, 8, 9]$ is ascending and $A[5..2] = [9, 7, 4, 4, 3, 1]$ is descending. Array $B = [5, 6, 7, 6, 5, 4, 3, 2]$ is also bitonic because $B[7..2] = [2, 5, 6, 7]$ is ascending and $B[2..7] = [7, 6, 5, 4, 3, 2]$ is descending. It can be verified that the sequence $[2, 4, 9, 3, 7]$ is not bitonic.

We now state a crucial property of bitonic sequences.

Lemma 6.2 (Bitonic Merge Lemma) *Let A be a bitonic sequence of length $2n$. Consider B and C defined as follows. $B = [min(A[i], A[i+n] | 0 \le i < n]$ $C = [max(A[i], A[i+n] | 0 \le i < n]$ Then B and C are bitonic and all values in B are less than or equal to all values in C.*

Proof: *(Sketch)* We apply the zero-one principle to prove this Lemma. Note that all bitonic sequences of zeroes and ones are either of the form $0^k 1^l 0^m$ or $1^k 0^l 1^m$ for some $k, l, m \ge 0$. By doing case analysis of such sequences the lemma can be shown.

■

Fig. 6.4 shows the process of splitting an array A of size $2n$ such that the first half is ascending and the second half is descending and then taking componentwise minimum B and maximum C. It is easy to see visually that B and C are bitonic and all values in B are less than or equal to all values in C.

We now give the algorithm for bitonic sort. As mentioned earlier, the structure of the algorithm is very similar to that of a Mergesort. In mergesort, two halves of the arrays are recursively sorted and then merged. In Bitonic sort, we use bitonic merge which is more easier to parallelize using a comparator network. In Bitonic sort, instead of sorting both halves in the same direction (ascending or descending), we sort the first half in the ascending order and the second half in the descending order. The resulting array is a bitonic sequence and we apply bitonic merge to it. Bitonic merge splits the array into two parts and does pairwise comparison according to Lemma 6.2. As a consequence of the lemma, all the elements in the lower half are less than the elements in the upper half. Furthermore, each of the halves are bitonic and we can call bitonic merge recursively on each half.

An example of bitonic sorting network for four inputs is shown in Fig. 6.5 and for eight inputs in Fig. 6.6.

BitonicSort algorithm recursively calls BitonicSort(L) and BitonicSort(R) on the left half L and right-half R. BitonicSort(L) and BitonicSort(R) can be done in parallel. BitonicMerge first does $n/2$ comparisons in parallel. It then recursively calls bitonicMerge on both halves. Hence, it takes $O(\log n)$ time. We can now calculate $T(n)$, parallel time, for bitonic sort. We have,

Algorithm BitonicSort: Bitonic Sort

```
procedure bitonicSort(int $low$, int $n$, boolean $isAscending$)
// Assume that $n$ is a power of 2
begin
    if $(n > 1)$
        int $m = n/2$;
        $bitonicSort(lo, m, true)$ in parallel $bitonicSort(lo + m, m, false)$;
        $bitonicMerge(lo, n, isAscending)$;
end

procedure bitonicMerge(int $low$, int $n$, boolean $isAscending$)
begin
    if $(n > 1)$
        int $m = n/2$;
        for (int $i = low; i < low + m; i++$) in parallel do
            int $j = i + m$;
            if $((A[i] > A[j])$ and $isAscending)$ swap$(A[i], A[j])$;
            else if $((A[i] < A[j])$ and $\neg isAscending)$ swap$(A[i], A[j])$;
        $bitonicMerge(lo, m, isAscending)$ in parallel $bitonicMerge(lo + m, m, isAscending)$
end
```

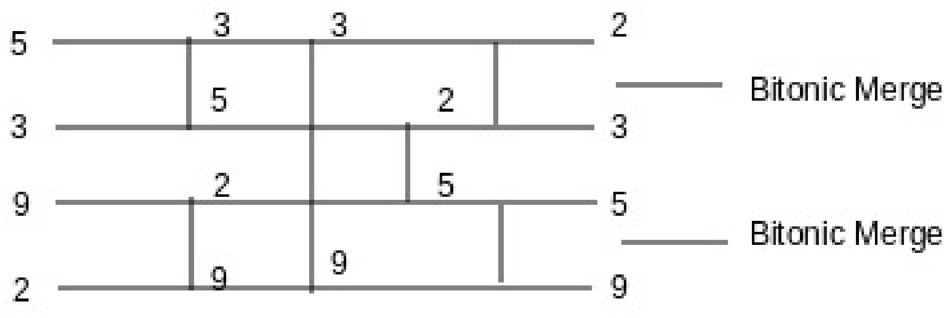


Figure 6.5: Example of a Bitonic Sorting network on four inputs.

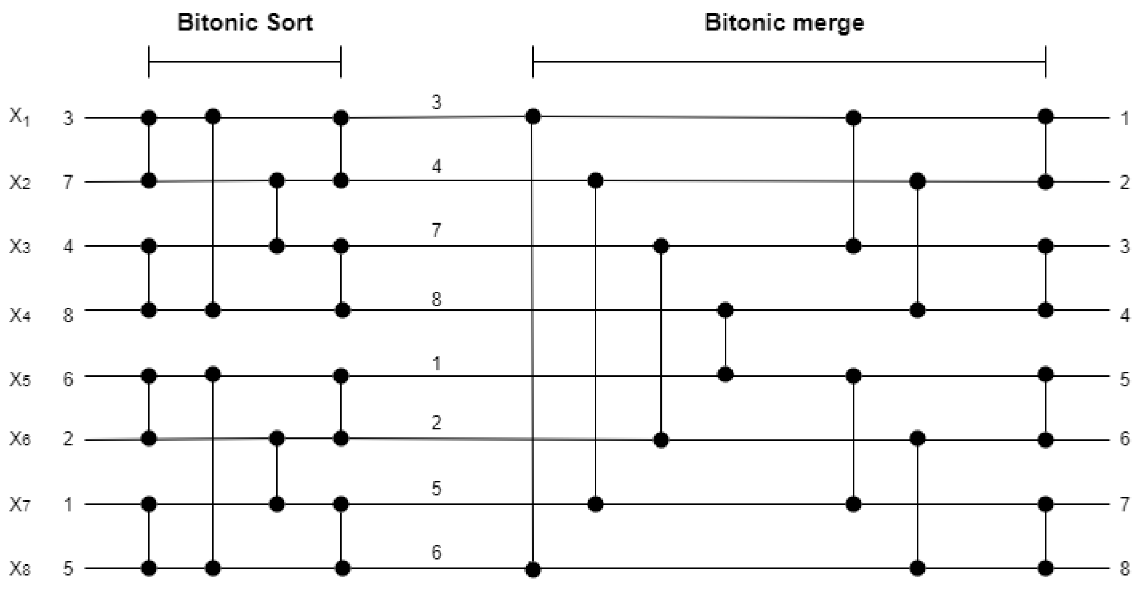


Figure 6.6: Example of a Bitonic Sort on eight inputs.

$T(n) = T(n/2) + O(log(n))$
$T(1) = O(1)$
Therefore, $T(n) = T(n/2) + log(n) = O(log^2 n)$.

The algorithm uses $O(n \log^2 n)$ comparators and therefore its work complexity is also $O(n \log^2 n)$.

6.6 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 follows:

```
**for** $i := 1$ to $k$ do // the key has $k$ digits
        use stable sort for Array $A$ on digit $i$
```

The time complexity of this algorithm is $O(kn)$ assuming that the stable sort is accomplished in $O(n)$ time.

Problem 8 asks you to design a parallel algorithm for Radix Sort.

6.7 Summary

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

Problem	**Algorithm**	**Parallel Time**	**Work**
Sorting	Odd-Even Sort	$O(n)$	$O(n^2)$
Sorting	Insert Sort	$O(n)$	$O(n^2)$
Sorting	Sequential Merge Sort	$O(n \log n)$	$O(n \log n)$
Sorting	Parallel Merge Sort	$O(\log^2 n)$	$O(n \log^2 n)$
Sorting	Sequential Quick Sort	$O(n^2)$	$O(n^2)$
Sorting	Bitonic Sort	$O(\log^2 n)$	$O(n \log^2 n)$

6.8 Problems

1. Implement the binary search algorithm discussed in Section 6.1.
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 a parallel Insertsort algorithm based on parallel-prefix.
4. Give an algorithm to merge k sorted arrays of size n into a single sorted array.
5. 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.
6. Give a randomized version of Quicksort in which the pivot is chosen at random. Give the expected running time of your algorithm.
7. Give a parallel Quicksort algorithm based on parallel-prefix that works as follows. The procedure takes an array and divides into three parts: all elements less than the pivot, all elements equal to the pivot and all elements greater than the pivot. It can then recurse on each of the subparts.
8. Give a parallel algorithm for RadixSort using parallel-prefix.

6.9 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 7

List Ranking and Pointer Jumping Algorithms

7.1 Introduction

In Chapters 4 and 5, we covered parallel algorithms on arrays via reduce and parallel-prefix. The advantage of arrays is that any thread can access any element in the array in constant time. In many applications, we have a linked list of threads such that each thread has some data item. The linked list does not provide us *random access* to its elements. Can we still do operations such as computing the maximum in $O(\log n)$ time where the length of the linked list is n? In this chapter, we introduce the technique of *pointer jumping* that can be applied to linked lists with significant reduction in parallel time complexity.

This chapter is organized as follows. Section 7.2 describe the pointer-jumping algorithm. Section 7.3 gives multiple applications of the pointer jumping technique. Section 7.4 gives the LLP algorithm for pointer jumping. We also consider the case when we have a circular linked list. For the circular linked list, there is no notion of list ranking. For a circular linked list, we are interested in coloring the nodes with at most three colors. Section 7.5 discuss a parallel algorithm that takes $O(\log^* n)$ parallel time.

7.2 Pointer Jumping

Suppose that we have a linked list implemented as follows. Each node i in the list has a field $next[i]$ that returns the next node in the linked list. If the element i is the last node in the linked list, then $next[i]$ returns -1. Our goal is to return an array R that returns the distance of any element from the last node in the linked list. In the Pointer-Jumping, we first copy the *next* pointers in an array Q and then use a method called *pointer jumping* to reduce the length of the longest path in the link structure. The pointer jumping is accomplished by setting $Q[i]$ to $Q[Q[i]]$ for every node i. The algorithm assumes that all nodes execute each iteration of the while loop in a synchronous manner. An execution of the algorithm is shown in Fig. 7.1.

The algorithm satisfies the following invariants. The variable $Q[i]$ always points to a node that is reachable from node i (and to -1 if no node is reachable from i). Initially, $Q[i]$ points to the next node in the linked list except for the tail node and thus satisfies the invariant. The variable $R[i]$ is equal to the length of the path from i to $Q[i]$ in the linked list. Initially, $R[i]$ equals 1 for all nodes except the tail node and therefore satisfies the invariant. In any iteration of the while loop, we consider a node i such that $Q[i]$ and $Q[Q[i]]$ are not null (i.e., if $Q[i]$ is viewed as the parent of node i, then node i has a grandparent). By setting $Q[i]$ to $Q[Q[i]]$, we maintain the invariant on Q. Furthermore, by setting $R[i]$ to $R[i] + R[Q[i]]$, we maintain the invariant on R because the length of the path from i to $Q[Q[i]]$ is the sum of the lengths of the paths from i to $Q[i]$ and from $Q[i]$ to $Q[Q[i]]$.

Algorithm Pointer-Jumping: Finding the distance from the tail of the linked list.

```
1  $P_i$::
2  input: $next[i]$: pointer to the next node in the linked list;
3  var: $Q[i]$: pointer to a node;
4        $R[i]$: int; // distance to the tail of the linked list
5  forall $i$ in parallel do:
6        $Q[i] := next[i]$;
7  forall $i$ in parallel do:
8        if $(Q[i] == -1)$ then $R[i] := 0$ else $R[i] := 1$;
9  forall $i$ in parallel do:
10       while $(Q[i] \neq -1)$ and $(Q[Q[i]] \neq -1)$ do
11           // evaluate the second conjunct only if $Q[i] \neq -1$
12           $R[i] := R[i] + R[Q[i]]$;
13           $Q[i] := Q[Q[i]]$;
```

```
i:       0  1  2  3   4   5
next:    3  4  5  1  -1   0

R:       1  1  1  1   0   1
Q:       3  4  5  1  -1   0
links: 2 --> 5 --> 0  --> 3 --> 1 --> 4

iteration 1:
i:       0  1  2  3   4   5
R:       2  1  2  2   0   2
Q:       1  4  0  4  -1   3
links:     2 --> 0 --> 1 --> 4       and 5 --> 3 --> 4

iteration 2:
i:       0  1  2  3   4   5
R:       3  1  4  2   0   4
Q:       4  4  1  4  -1   4
links:     2 --> 1 --> 4      0 --> 4  5 --> 4  3 --> 4

iteration 3:
i:       0  1  2  3   4   5
R:       3  1  5  2   0   4
Q:       4  4  4  4  -1   4
links:     2 --> 4    1 --> 4   0 --> 4  5 --> 4  3 --> 4
```

Figure 7.1: Execution of Pointer Jumping Algorithm

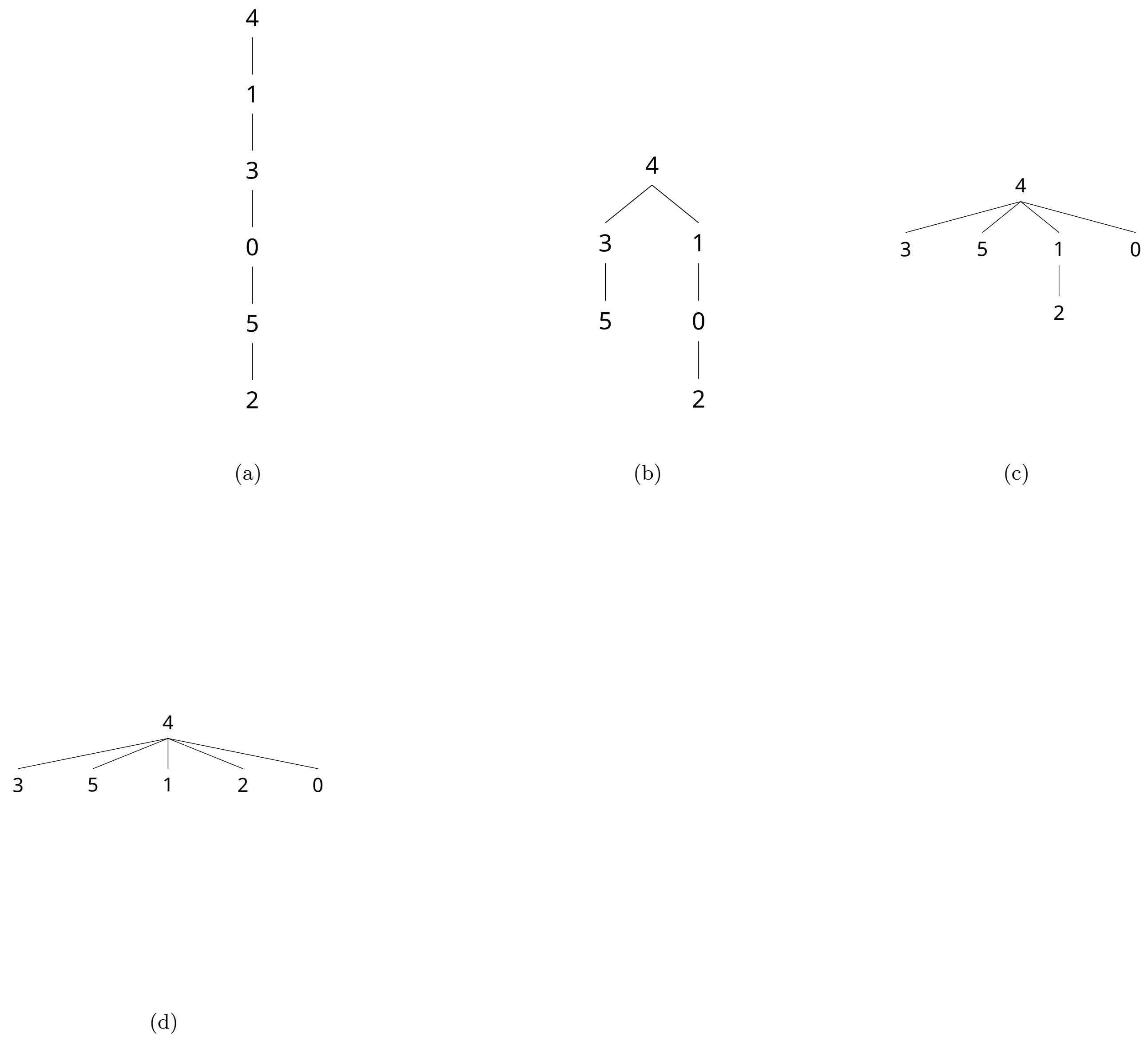


Figure 7.2: Pointer Jumping (a) Initial Tree (b) Tree after first iteration (c) Tree after second iteration (d) Star tree after the third iteration

When there is no node i such that it has a grandparent, we have that every node except the tail node is pointing to the tail node (a reachable node that is pointing to null). From the invariant on R, we get that $R[i]$ equals the length of the path from i to the tail node.

This algorithm takes $O(\log n)$ time where n is the size of the linked list. It does $O(n \log n)$ work and is therefore not work-optimal. It is possible to reduce the work to $O(n)$ using list contraction. We will not discuss those algorithms and refer the interested reader to [JáJ92].

Although the algorithm was given for the linked list, it is easily verified that the idea of *pointer jumping* is applicable to any rooted tree where a node points to its parent in the tree. In case of a linked list, we had a degenerate tree. The algorithm return the distance of any node from the root.

Let us also verify that we need only CREW PRAM for implementation of this algorithm. In any time step a node i writes only to its own $R[i]$ and $Q[i]$. Hence, CREW PRAM is sufficient to implement the pointer jumping algorithm.

7.3 Applications of Pointer Jumping

We now show that the pointer jumping technique can be applied to many problems.

- *Returning the maximum value in a linked list*: Let us consider the problem mentioned in the introduction of this Chapter. Suppose that we have a linked list of numbers and we would like to compute the maximum of these numbers. Each node i has a natural number $num[i]$ stored at i.

Algorithm Max-Pointer-Jumping: Finding the Maximum in a linked list.

```
$P_i$::
input: $next[i]$: pointer to the next node in the linked list;
var: $Q[i]$: pointer to a node;
    $R[i]$: int; // distance to the tail of the linked list
forall $i$ in parallel do:
    $Q[i] := next[i]$;
    $R[i] = num[i]$;
forall $i$ in parallel do:
    while ($Q[i] \neq -1$) and ($Q[Q[i]] \neq -1$) do
        // evaluate the second conjunct only if $Q[i] \neq -1$
        $R[i] := max(R[i], R[Q[i]])$;
        $Q[i] := Q[Q[i]]$;
```

The head node now has the maximum and can write that value on a shared variable.

- *Converting a linked list into an array*: In our original problem, we had computed the distance of any node from the tail of the linked list. Let us suppose that we are given a linked list and would like to convert into an array. If the space for the array is allocated, then every thread can compute the index in which the entry in the linked list needs to be written. Observe that once we have an array, we can use parallel-prefix to solve many problems.

  This technique returns the array in the reversed order. We leave it to the reader to compute the array that has values stored in the same order as the linked list.

7.4 LLP Algorithm for List Ranking

In this section we show how List Ranking algorithm can be viewed as LLP Algorithm. We state the problem as follows. We are given a tree rooted at vertex r as input. Each node i other than the root points to another node. The root node r points to null. Our goal is to determine the distance of every node from the root. We view this as the search problem for an integer valued vector G that satisfies a certain feasibility predicate. There is one component in G for every node other than the root. $G[i].dist$ is initialized to 1 as no node can have distance less than 1 from the root. However, this vector G may not be feasible. Associated with every i, we also keep $G[i].next$ which indicates a node that can be reached in $G[i].dist$ hops. Initially, $G[i].next$ is simply the parent of i in the tree. We define $B(G)$ as $\forall i : G[i].next = r$. Thus, any node j is forbidden in G if $G[j].next \neq r$. To advance G, we set $G[j].dist$ to $G[j].dist + G[G[j].next].dist$ and $G[j].next$ to $G[G[j].next].next$. The resulting algorithm LLP-ListRank returns only when for every j, $G[j].next$ equals r. The execution time of the algorithm is crucially dependent on the order in which the forbidden vertices are advanced. If all vertices that are forbidden are advanced in parallel, then the algorithm takes $O(\log n)$ time in the worst case.

Algorithm LLP-ListRank: A Parallel LLP Algorithm for List Ranking

```
1 input: $parent$: array[1..$n$] of int
2 var: $G$: array[1..$n-1$] of {dist, next};
3 $P_j$: Code for thread $j$
4 init: $G[j].dist := 1; G[j].next := parent[j]$
5 forbidden: $G[j].next \neq r$
6     advance: $G[j].dist := G[j].dist + G[G[j].next].dist$;
7         $G[j].next := G[G[j].next].next$;
```

7.5 Vertex Coloring in a Ring

Suppose that we have a circular linked list of n nodes. We would like to color the nodes in the linked list so that adjacent vertices do not have the same color. It is desirable to use as few colors as possible. If there are odd number of nodes in the circle with 3 or more nodes, then we would need three colors. By sequentially traversing the nodes in the cycle, this task can be accomplished in $O(n)$ time. How can we do this in parallel? We first show that given any valid coloring with greater than 6 colors, the nodes can be assigned another set of colors such that the number of colors in the new set is significantly reduced. Assume that initially all nodes have valid coloring. For example, each node could initially have different color by simply using its unique identifier. The following parallel step accomplishes the reduction in the number of colors.

Algorithm Par-VertexColoring: A Parallel Algorithm for Vertex Coloring in a Ring

```
1 input: $n$ nodes arranged in a cycle such that $next[i]$ gives the next node for node $i$
2     A coloring $c$ of vertices such that for all $i$: $c[i]$ differs form $c[next[i]]$
3 output: Another valid coloring $d$ with fewer colors
4 forall nodes $i$ in parallel do
5     let $k$ be the least significant bit such that $c[i]$ and $c[next[i]]$ differ
6     $d[i] := 2 * k +$ the $k^{th}$ least significant bit in $c[i]$
```

Consider a circular linked list $(12, 9, 13, 1, 3)$ where the node with 3 points back to the node with 12. The execution of the algorithm on this set of nodes is shown below.

node	c	k	d
12	1100	0	0
9	1001	2	4
13	1101	2	5
1	0001	1	2
3	0011	0	1

The algorithm gives a valid coloring. Suppose that $d[i] = d[next[i]]$. This means that the corresponding least significant bit k for i and $next[i]$ must be the same. By our algorithm for defining d, this would imply that k^{th} bit for i and $next[i]$ would also be same. This contradicts the definition of k.

The number of color reduces to $O(\log n)$ after every iteration so long as the initial number of colors is greater than 6. Hence in $O(\log^* n)$ iterations, we can reduce the colors to at most 6, where $\log^*(n)$ is the minimum number of times the log function needs to be applied to n to get a number at most 1. Formally, let $\log^{(i)}(n)$ be the log function iterated i times. Then,

$$\log^*(n) = min\{j \mid log^{(j)}(n) \leq 1\}$$

We can now reduce the number of colors to 3 as follows. We first replace all $3's$ in parallel by a number from $0, 1, 2$. Every node can choose a color different from its predecessor and the successor. We then replace all $4's$ and finally all $5's$. These actions correspond to only three parallel steps. The algorithm takes $O(n \log^* n)$ work because n processors take $O(\log^* n)$ time. Therefore the algorithm is not work-optimal but close to work-optimal since $\log^*(n)$ is a very slowly growing function.

7.6 Summary

The following table lists all the algorithms discussed in this chapter.

Problem	**Algorithm**	**Parallel Time**	**Work**
List Ranking	Pointer Jumping	$O(\log n)$	$O(n \log n)$
List Ranking	LLP	$O(\log n)$	$O(n \log n)$
Vertex Coloring	Par-Vertex Coloring	$O(\log^* n)$	$O(n \log^* n)$

7.7 Problems

1. Show that the number of iterations of the list ranking algorithm is $O(\log n)$ if the length of the linked list is n.
2. Given a linked list such that each thread has reference to a node in the linked list of integers, give a parallel algorithm to determine the total number of even numbers in the linked list.
3. Given a linked list such that each thread has reference to a node in the linked list of integers, give a parallel algorithm to determine the total number of times the integer z appears in the linked list.
4. Give a parallel algorithm to compute the suffix sums for all nodes in a linked list.

7.8 Bibliographic Remarks

Reader will find a detailed description of list ranking algorithms in [JáJ92] and [LB00].

Chapter 8

Basic Graph Algorithms

8.1 Introduction

Graph theory has many practical applications in a variety of fields, including computer science, operations research, social sciences, and biology. 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 8.2 gives a parallel LLP algorithm to traverse a graph. Section 8.3 gives a parallel LLP algorithm to construct a breadth-first-search (BFS) tree from a graph given a source vertex. Section 8.5 gives a parallel LLP algorithm to topologically sort a directed acyclic graph. Section 8.6 gives an LLP algorithm to find connected components in an undirected graph. Section 8.7 gives a faster LLP algorithm based on pointer jumping.

8.2 General 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

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

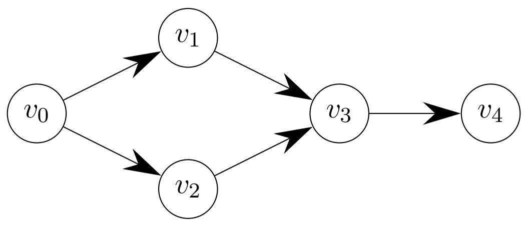


Figure 8.1: A directed graph

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: Finding the reachable set of vertices.

```
Process $P_j$
input: $edges(j)$: list of $1..n$;
init($j$): if $(j = 0)$ $G[j] := 1$ else $G[j] := 0$;
forbidden($j$): $\exists i : j \in edges(i) : (G[i] = 1) \wedge (G[j] = 0)$
    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

```
var $G$: vector of int initially $\forall i : G[i] = 0$;
    $S$: set of indices;
$S.add(0)$; // add 0 as the initial forbidden index
$G[0] := 1$;
while $\neg S.empty()$ do
    $j := S.removeAny(G)$; // remove any index from $S$
    forall $(k \in dep(j))$
        if $(G[k] = 0)$ // $k$ is forbidden
            $G[k] := 1$; // advance on $k$
            add $k$ to $S$;
endwhile;
return $G$; // the optimal solution
```

8.3 Breadth First Search Tree in a Directed Graph

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.

```
Process $P_j$
**input**: $pre(j)$: list of $1..n$;
**init**$(j)$: **if** $(j = 0)$ $G[j] := 0$ **else** $G[j] := \infty$;
**ensure**$(j)$: $G[j] \leq min\{G[i] + 1 \mid i \in pre(j)\}$
```

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.

Algorithm LLP-Traversal-BFS is based on using the predecessor information for each vertex. Alternatively, we can use the adjacency list for each vertex. We now show an implementation of breadth-first-search, when we are also required to 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. 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.

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

```
var $G$: vector of int initially $\forall i \neq 0 : G[i] = maxint; G[0] := 0$;
    $parent$: vector of int initially $\forall i : parent[i] = -1$; // null parent
    $S$: set of indices initially $\{0\}$; // add $v_0$ as the initial forbidden index
while $\neg S.empty()$ do
    $T$ := set of indices initially $\{\}$;
    forall $(j \in S) \wedge (k \in dep(j))$ in parallel do:
        if $(G[k] > G[j] + 1)$
            $G[k] := G[j] + 1$;
            $parent[k] := j$;
            add $k$ to $T$;
    endforall;
    $S := T$;
endwhile;
return $G$;
```

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

```
**var** $G$: vector of int initially $\forall i : G[i] = 0$;
    $parent$: vector of int initially $\forall i : parent[i] = -1$; // null parent
    $S$: stack of indices initially empty;
    $discovered$: vector of int;
    $finished$: vector of int;
    $tick$: integer initially 1;
$S.add(s)$; // add $s$ as initial forbidden index
**while** $\neg S.empty()$ **do**
    $j := S.pop()$; // remove an index from $S$
    if $(G[j] = 0)$ then
        $G[j] := 1$;
        $discovered[j] := tick$;
        $tick := tick + 1$;
        **forall** $(k \in dep(j)) \wedge (G[k] = 0)$:
            $parent[k] := j$;
            $S.push(k)$;
        **endforall**;
        $finished[j] := tick$;
        $tick := tick + 1$;
    **endif;**
**endwhile**;
**return** $G$;
```

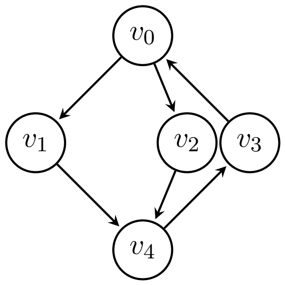


Figure 8.2: An Example of DFS-Traversal

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

Figure 8.3: The values of *discovered* and *finished* for vertices with DFS on the graph in Fig. 8.2

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. 8.2 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. 8.3.

8.5 Topological Sort 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.

```
input: $pre(j)$: list of $1..n$;
init(j): $G[j] := 0$;
    if $pre(j) = \{\}$ then $fixed[j] := true$ else $fixed[j] := false$;
forbidden(j): $\neg fixed[j] \wedge \forall i \in pre(j) : fixed[i]$
    advance(j): $G[j] := \max_{i \in pre(j)} G[i] + 1$; $fixed[j] := true$;
```

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.

8.6 Connected Components in an Undirected Graph: Slow Algorithm

We are given an undirected graph (V,E) on n vertices numbered $1 \ldots n$. Our goal is to label each vertex i with $G[i]$ such that $G[i]$ is the largest numbered vertex in the component to which i belongs. A vector G is defined to be feasible if

$$(\forall i : G[i] \geq i) \wedge (\forall (i,j) \in E : G[i] \geq G[j])$$

We consider the lattice of n dimensional vectors with natural $\leq$ relation on the vectors. We first claim that

Lemma 8.1 *$B_{comp}(G) \equiv (\forall i : G[i] \geq i) \wedge (\forall (i,j) \in E : G[i] \geq G[j])$ is a lattice linear predicate.*

Proof: We show that each of the conjuncts is lattice linear. The first conjunct $(\forall i : G[i] \geq i)$ is lattice linear because it is a conjunction of local predicates. The second conjunct $(\forall (i,j) \in E : G[i] \geq G[j])$ is lattice linear because if there exists $(i,j) \in E$ such that $G[i] < G[j]$, then unless $G[i]$ is advanced to $G[j]$, the predicate B_{comp} can never become true.

■

Lemma 8.2 *The least vector G in the lattice that satisfies B_{comp} gives us the labeling such that $G[i]$ is the largest numbered vertex in the component to which i belongs.*

Proof: Use induction on the distance of i from the largest numbered vertex in its component.

■

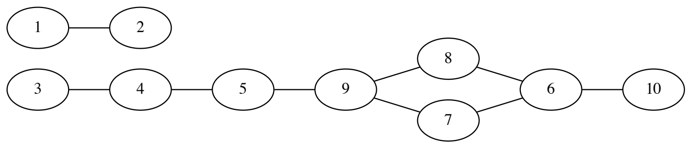


Figure 8.4: An undirected graph

Algorithm SlowComponents: Algorithm *SlowComponent* to find all the connected components of an undirected graph.

```
1   input: $adj(j)$: list of $1..n$;//adjacency list of vertex $j$
2   init: $G[j] := j$;
3   ensure: $G[j] \geq \max\{G[i] \mid i \in adj(j)\}$
```

By applying Lattice-Linear Predicate method, we get the Algorithm SlowComponents. We use n threads in the algorithm, one for each vertex j, responsible to update $G[j]$. Every thread j initializes $G[j]$ to j. If any thread j finds that there exists i such that i and j are adjacent and $G[j]$ is less than $G[i]$, then it updates its component to $G[i]$.

First, it is easy to verify that the algorithm *SlowComponents* maintains the invariant that $G[i] = j$ implies that i and j are in the same connected components. Initially, the invariant holds because $G[j]$ is set to j. Any update to $G[j]$ is to some $G[i]$ such that i is adjacent to j. By induction, $G[i]$ is in the same connected component as i and because i is connected to j, the invariant continues to hold for $G[j]$. Conversely, because LLP algorithm returns the least vector that satisfies B_{comp}, we get that the algorithm terminates with $G[i]$ as the largest numbered vertex in the component to which i belongs on account of Lemma 8.2.

Let us analyze the time complexity of the above algorithm in a synchronous model in which every step consists of a thread j determining whether it is forbidden, and then advancing to the maximum $G[i]$ such that $(i, j) \in E$. The algorithm may require as many as $O(n)$ steps in the worst case when nodes are arranged in a long chain with the largest node at one end of the chain.

8.7 Connected Components: A Faster Algorithm

We now show the application of *pointer jumping* technique to this problem. In this algorithm, we use the notion of prioritized forbidden function. In many situations, an index may be forbidden due to multiple reasons. Determining whether an index is forbidden requires some work and it makes sense to first compute the forbidden functions that are easy to compute. The forbidden function in Algorithm SlowComponents requires a node to scan all its adjacent neighbors. If we view $G[i]$ as the parent of i, then it is clear that $G[i]$ should also be greater than or equal to $G[G[i]]$. Checking violation of this condition is much simpler than scanning all adjacent neighbors. This step can be formalized as **ensure** $G[j] \geq G[G[j]]$.

Instead of using G as our state vector we use a more user-friendly name *parent* as the state vector. Then, the above step can be implemented as follows:

```
while $(\exists j : parent[j] < parent[parent[j]])$
    forall $j \in V$ in parallel do
        $parent[j] := parent[parent[j]]$
```

If the parent pointers form a rooted tree, then the pointer jumping technique reduces the height of the tree by making every node point to its grandparent in one time step. In $O(\log n)$ iterations of the while loop every *parent* pointer would be equal to the *grandparent* pointer.

Combining the predicate corresponding to pointer-jumping with the predicate that updates the parent pointer with the largest parent pointer in the neighborhood of any node, we get the algorithm LLP-FastComponents.

Algorithm LLP-FastComponents: Finding all the connected components of an undirected graph.

```
1    input: $adj(j)$: list of $1..n$;//adjacency list of vertex $j$
2    init: $parent[j] := j$;
3    ensure(1): $\forall i : parent[i] \geq parent[parent[i]]$
4    ensure(2): $\forall i : parent[i] \geq \max\{parent[j] \mid (\forall (i,j) \in E\}$
```

The algorithm has two *ensure* statements. The first ensure statement guarantees that the node i has its parent equal to its grandparent. The second ensure statement guarantees that the node i has its parent pointer at least as big as its neighbors. This statement alone would have been enough to find the largest label in a component. However, adding the first ensure statement speeds up the process.

Our implementation of the algorithm will execute the first ensure statement until it is globally true. In a synchronous model, this step will take $O(\log n)$ parallel time. When the first ensure statement is met, the directed graph created from *parent* pointer can be viewed as a collection of rooted stars. Every rooted star is part of the same components. However, nodes in the same components may belong to different rooted stars. The second ensure statement merges these stars. If there is no edge across rooted stars, then we are done and every rooted star is a component by itself. Otherwise, every rooted star computes the largest parent pointer known to any node in the rooted star and point to it. We can view each rooted star as a *supervertex*. Two vertices are in the same supervertex if their parents are the same. Once we have computed the largest parent known to a supervertex, if the new parent pointer discovered is bigger than the earlier root, then this tree merges to another tree. Otherwise, this rooted tree does not join any other rooted tree. We show that the number of rooted stars reduce by a factor of two in every two such steps. If there is any edge from this rooted star i to another rooted star j, then there are three cases:

1. If $parent[i]$ is less than $parent[j]$, then the star i joins the star j.
2. If $parent[j]$ is less than $parent[i]$ and $parent[i]$ is the largest parent pointer known to star j, then star j joins star i.
3. If $parent[j]$ is less than $parent[i]$ but $parent[i]$ is not the largest parent pointer known to star j, then star j joins some other star k that has bigger parent pointer than star i. In this case, we know that whenever we scan parents of the neighbors next time star i will discover a bigger star and join that star.

From the above analysis, we get that the number of rooted stars go down by a factor of 2 in at most two steps of scanning neighbors of supervertices.

Whenever, we merge rooted stars, we get rooted trees and we can apply the pointer-jumping again. The algorithm terminates when no rooted star can be merged with any other rooted star. At that point, every rooted star corresponds to a component of the graph and the required labeling is given by the parent pointer.

Algorithm FastComponents gives the entire algorithm. It initializes parent node of every node i as itself (line 4). Thus, initially every node is a rooted star by itself. It has a *while* loop that checks for the forbidden condition that there exists an edge between two nodes with different *parent* (line 5) If there is no such edge, then we are done and every rooted star corresponds to a component. Otherwise, every node computes the largest parent that it or one of its neighbor knows (line 8). Once, every vertex has computed this value, we merge the rooted stars by making the root point to the largest parent pointer known by the supervertex. This operation will merge rooted star to the biggest rooted star know to it and create rooted trees. We can then do pointer jumping to create new rooted stars and go back to the while loop.

It consists of $O(\log n)$ iterations such that each iteration takes $O(\log n)$ time giving us the time complexity of $O(\log^2 n)$.

Algorithm FastComponents: Finding Connected Components

```
Input: Undirected Graph: $(V, E)$.
Output: Labeling of vertices by the component id
var
    $parent$: array[1..$n$] of 0..$n$ initially all $\forall i : parent[i] = i$;
while $(\exists (v,w) \in E : parent[v] < parent[w])$
    // Step 1: Compute $vmax$ for every vertex
    forall $v \in V$ in parallel do
        $vmax[v] := \max\{parent[w] \mid$ such that $(v,w) \in E \lor (w = v)\}$
    // Step 2: Make the $parent$ of $v$ as the largest $vmax$ in its neighborhood
    forall $v \in V$ in parallel do
        if $v = parent[v]$ then //$v$ is the root of the star
            $parent[v] = \max\{vmax[u] | parent[v] = parent[u]\}$
    // Step 3: Convert every rooted tree into a rooted star
    while $(\exists j : parent[j] \neq parent[parent[j]])$
        forall $v \in V$ in parallel do
            $parent[v] := parent[parent[v]]$
endwhile
```

We now give an example of the execution of the algorithm on the graph shown in Fig. 8.4. It is easy to see visually that there are two connected components in this graph.

8.8 Summary

The following table lists all the algorithms discussed in this chapter.

Problem	**Algorithm**	**Parallel Time**	**Work**
Breadth-First Search	BFS-Traversal-Parallel	critical path length	$O(m+n)$
Topological Sort	LLP	critical path length	$O(m+n)$
Components	Slow LLP	$O(n)$	$O(m+n)$
Components	Fast LLP	$O(\log^2 n)$	$O(m+n)$

8.9 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.

8.10 Bibliographic Remarks

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

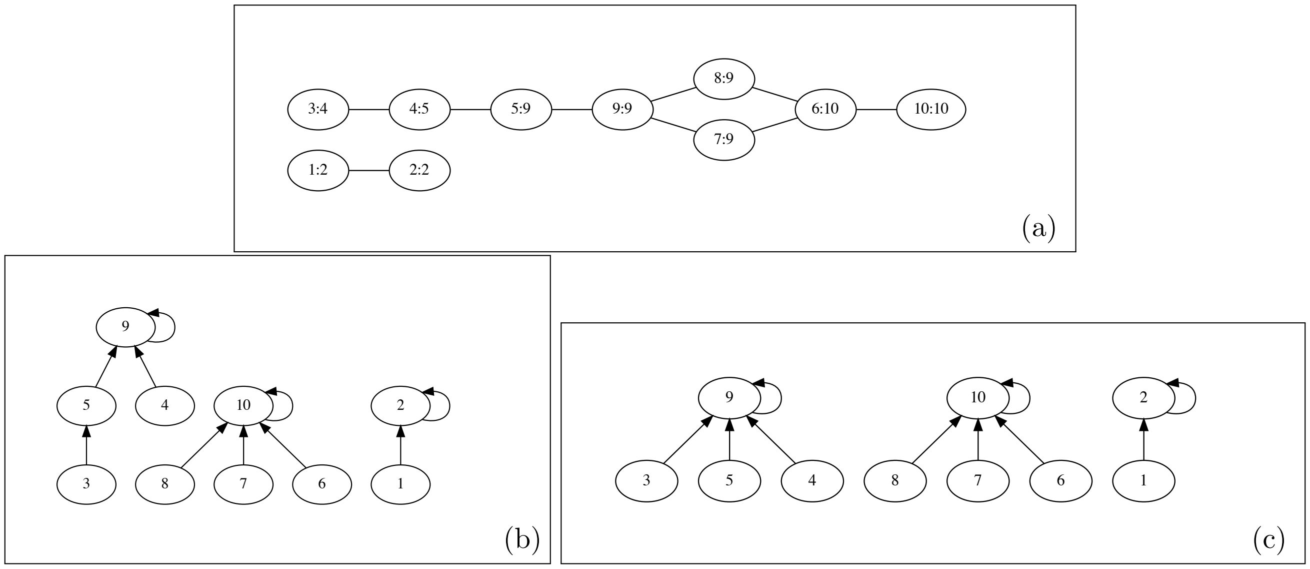


Figure 8.5: (a) *vmax* for every node in Fig. 8.4 (b) Pointing to largest *vmax* to get Rooted-trees (c) Rooted-stars at the end of first iteration.

Chapter 9

The Shortest Path Problem

9.1 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.

Dijkstra's algorithm and the LLP algorithm assume that all edge weights are positive. Section 9.5 derives Johnson's algorithm from a lattice-linear predicate to convert the problem of a graph with negative weights (but no negative cost cycle) to a problem with non-negative weights. Section 9.6 discusses Bellman-Ford algorithm. Bellman-Ford algorithm works on any graph so long as there are no negative cost cycles.

Note that there are multiple parallel algorithms on *undirected* graphs which are not only efficient in terms of time but also in terms of the work. The situation is different for the directed graphs. Many of the problems in the directed graph require reachability. All the time-efficient parallel algorithms for reachability suffer from transitive-closure bottleneck. The transitive closure bottleneck refers to the problem that the *polylog* algorithms require a large number of processors making the parallel algorithm work inefficient (and thus impractical). In this chapter, we cover the problem of computing transitive closure of a matrix. This problem can be solved fast using the technique of *repeated squaring*. The same technique is applicable to solve all pair shortest path algorithm. However, these algorithms are far from being work-efficient. In any case, it is important to study these parallel algorithms to appreciate the transitive closure bottleneck.

In this chapter, we first describe Dijkstra's algorithm in Section 9.2. We then formulate the shortest path problem as searching for an appropriate element in a lattice and derive an LLP algorithm for the shortest path in Section 9.3. Section 9.4 discusses a parallel algorithm for the shortest path problem called the delta-stepping algorithm. Section 9.5 describes techniques that we can use when the directed graph may have edges with negative weights. In particular, it describes Johnson's algorithm that converts a weighted directed graph to another weighted graph that preserves the shortest paths and the resulting graph does not have negative weight edges. Section 9.6 describes Bellman-Ford algorithm that can be used to find the cost of the shortest path when the graph may have negative edge weights. We then consider the problem of finding costs of shortest paths for all pairs of vertices. Section 9.7 starts with an algorithm for matrix multiplication. Then, it extends the algorithm to the problem of computing the transitive closure of a binary matrix. Section 9.8 extends this problem to nonbinary matrices by viewing it as a computation of the cost of shortest path for all pairs of vertices.

9.2 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. 9.1(a) 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.

Algorithm Dijkstra: Finding the shortest costs from v_0 .

```
var $dist$: array$[0 \dots n-1]$ of integer initially $\forall i : dist[i] = \infty$;
    $fixed$: array$[0 \dots n-1]$ of boolean initially $\forall i : fixed[i] = false$;
    $H$: binary heap of $(j, d)$ initially empty;// $j$ is the vertex and $d$ is the cost
$dist[0] := 0$;
$H$.insert((0,dist[0]));
while $\neg H$.empty() do
    $(j, d) := H$.removeMin();
    if $(fixed[j])$ continue;
    $fixed[j] := true$;
    forall $k$: $\neg fixed(k) \wedge (j, k) \in E$
        if $(dist[k] > dist[j] + w[j, k])$ then
            $dist[k] := dist[j] + w[j, k]$;
            $H$.add $(k, dist[k])$;
endwhile;
```

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.

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 9.1 *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 $v_0 \rightarrow v_1$, 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 $v_0 \rightarrow v_2$, 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 $v_2 \rightarrow v_3$, 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 $v_2 \rightarrow v_4$, 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.

9.3 Constrained Single Source Shortest Path Algorithm

In this section, we assume that all edge weights are *strictly positive*. We are required to find the minimum cost of a path from a distinguished *source* vertex v_0 to all other vertices where the cost of a path is defined as the sum of edge weights along that path. For any vertex v, let $pre(v)$ be the set of vertices u such that (u, v) is an edge in the graph. To avoid trivialities, assume that every vertex v (except possibly the source vertex v_0) has nonempty $pre(v)$ and that all nodes in the graph are reachable from the source vertex.

As the first step of the predicate detection algorithm, we define the lattice for the search space. We assign to each vertex v_i, $G[i] \in \mathbf{R}_{\geq 0}$ with the interpretation that $G[i]$ is the cost of reaching vertex v_i. We call G, the *assignment* vector. The invariant maintained by our algorithm is: for all i, the cost of any path from v_0 to v_i is greater than or equal to $G[i]$. The vector G only gives the lower bound on the cost of a path and there may not be any path to vertex v_i with cost $G[i]$. To capture that an assignment is feasible, we define *feasibility* which requires the notion of a *parent*. We say that v_i is a parent of v_j in G (denoted by the predicate $parent(j, i, G)$) iff there is a direct edge from v_i to v_j and $G[j]$ is at least $(G[i] + w[i, j])$, i.e., $(i \in pre(j)) \wedge (G[j] \geq G[i] + w[i, j])$. A node may have multiple parents.

In Fig. 9.1, let G be the vector $(0, 2, 3, 5, 8)$. Then, v_0 is a parent of v_2 because $G[2]$ is greater than $G[0]$ plus $w[0, 2](i.e., 3 \geq 0 + 2)$. Similarly, v_1 is a parent of v_4 because $G[4] \geq G[1] + 2$. A node may have multiple parents. The node v_2 is also a parent of v_4 because $G[4] \geq G[2] + 5$.

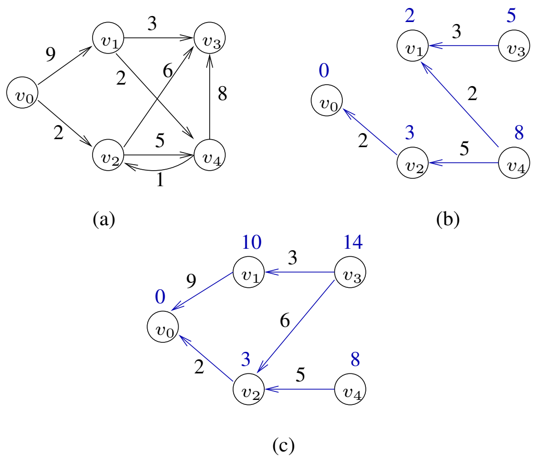


Figure 9.1: (a) A Weighted Directed Graph (b) The parent structure for $G = (0, 2, 3, 5, 8)$ (c) The parent structure for $G = (0, 10, 3, 14, 8)$. Since every non-source node has at least one parent, G is feasible.

Since $w[i, j]$ are strictly positive, there cannot be a cycle in the parent relation. Now, feasibility can be defined as follows.

Definition 9.2 (Feasible for paths) *An assignment G is* feasible for paths *iff every node except the source node has a parent. Formally, $B_{path}(G) \equiv \forall j \neq 0 : (\exists i : parent(j, i, G))$.*

Hence, an assignment G is feasible iff one can go from any non-source node to the source node by following any of the parent edges. We now show that feasibility satisfies lattice-linearity.

Lemma 9.3 *For any assignment vector G that is not feasible, $\exists j : forbidden(G, j, B_{path}(G))$.*

Proof: Suppose G is not feasible. Then, there exists $j \neq 0$ such that v_j does not have a parent, i.e., $\forall i \in pre(j) : G[j] < G[i] + w[i, j]$. We show that $forbidden(G, j, B_{path}(G))$ holds. Pick any H such that $H \geq G$. Since for any $i \in pre(j)$, $H[i] \geq G[i]$, $G[j] < G[i] + w[i, j]$ implies that $G[j] < H[i] + w[i, j]$. Therefore, whenever $H[j] = G[j]$, v_j does not have a parent.

■

Since B_{path} is a lattice-linear predicate, it follows from Lemma 2.3(c), that the set of feasible assignment vectors are closed under meets (the component-wise min operation). Hence, we can use LLP algorithm with $\alpha(G, j, B_{path}) = \min\{G[i] + w[i, j] \mid i \in pre(j)\}$.

For an unweighted graph (i.e., each edge has weight equal to 1), the above parallel algorithm requires time equal to the distance of the farthest node from the root. The LLP algorithm derived from B_{path}, however, may take time that depends on the weights because the advancement along a forbidden process may be small.

We now give an alternative feasible predicate that results in an algorithm that takes bigger steps. We first define a node j to be *fixed* in G if either it is the source node or it has a parent that is a fixed node, i.e., $fixed(j, G) \equiv (j = 0) \vee (\exists i : parent(j, i, G) \wedge fixed(i, G))$.

Observe that node v_0 is always fixed. Any node v_j such that one can reach from v_j to v_0 using parent relation is also fixed. We now define another feasible predicate called B_{rooted}, as $B_{rooted}(G) \equiv \forall j : fixed(j, G)$.

Even though it may first seem that the predicate B_{rooted} is strictly stronger than B_{path}, the following Lemma shows otherwise.

Lemma 9.4 *$B_{path}(G)$ iff $B_{rooted}(G)$.*

Proof: If G satisfies B_{rooted}, then every node other than v_0 has at least one parent by definition of $fixed$, hence $B_{path}(G)$. Conversely, suppose that every node except v_0 has a parent. Since parent edges cannot form a cycle, by following the parent edges, we can go from any node to v_0.

■

It follows that the predicate B_{rooted} is also lattice-linear. Then, the following threshold $\beta(G)$ is well-defined whenever the set of edges from the fixed vertices to non-fixed vertices is nonempty.

$$\beta(G) = \min_{(i,j): i \in pre(j)} \{G[i] + w[i,j] \mid fixed(i,G), \neg fixed(j,G)\}.$$

If the set of such edges is empty then no non-fixed vertex is reachable from the source. We call these set of edges *Heap* (because we would need the minimum of this set for advancement).

We now have the following result in advancement of G.

Lemma 9.5 *Suppose* $\neg B_{rooted}(G)$.
Then, $\neg fixed(j,G) \Rightarrow forbidden(G, j, B_{rooted}, \beta(G))$.

Proof: Consider any assignment vector H such that $H \geq G$ and $H[j] < \beta(G)$. We show that H is not B_{rooted}. In particular, we show that j is not fixed in H. Suppose j is fixed in H. This implies that there is a path W from v_0 to v_j such that all nodes in that path are fixed. Let the path be the sequence of vertices $w_0, w_1, \ldots w_{m-1}$, where $w_0 = v_0$ and $w_{m-1} = v_j$. Let $w_l = v_k$ be the first node in the path that is not fixed in G. Such a node exists because w_{m-1} is not fixed in G. Since w_0 is fixed, we know that $1 \leq l \leq m-1$. The predecessor of w_l in that path, w_{l-1} is well-defined because $l \geq 1$. Let $w_{l-1} = v_i$.

We show that $H[k] \geq \beta(G)$ which contradicts $H[j] < \beta(G)$ because $H[k] \leq H[j]$ as the cost can only increase going from k to j along the path W. We have $H[k] \geq H[i] + w[i,k]$ because i is a parent of k in H. Therefore, $H[k] \geq G[i] + w[i,k]$ because $H[i] \geq G[i]$. Since i is fixed in G and k is not fixed in G, from the definition of $\beta(G)$, we get that $\beta(G) \leq G[i] + w[i,k]$. Hence, $H[k] \geq \beta(G)$.

■

By using the advancement Lemma 9.5, we get the algorithm *ShortestPath* shown in Fig. 9.2. In this algorithm, in every iteration we find all nodes that are forbidden (not fixed) and advance them. All nodes are advanced to $\alpha(G,j)$ that combines $\beta(G)$ with $\min\{G[i] + w[i,j] \mid i \in pre(j)\}$. Note that if a node is fixed, its parent is fixed and therefore any algorithm that advances $G[j]$ only for non-fixed nodes j maintains that once a node becomes fixed it stays fixed.

By removing certain steps, we get Dijkstra's algorithm from the algorithm *ShortestPath*. It is clear that the algorithm stays correct if $\alpha(G,j)$ uses just $\beta(G)$ instead of $\max\{\beta(G), \min\{G[i] + w[i,j] \mid i \in pre(j)\}\}$. Secondly, the algorithm stays correct if we advance G only on the node j such that (i,j) are in Heap edges and the node j minimizes $G[i] + w[i,j]$. Finally, to determine such a node and $\beta(G)$, it is sufficient to maintain a min-heap of all non-fixed nodes j, along with the label that equals $\min_{i \in pre(j), fixed(i,G)} G[i] + w[i,j]$. On making all these changes to *ShortestPath*, we get Dijkstra's algorithm (modified to run with a heap).

It is illustrative to compare the algorithm *ShortestPath* with Dijkstra's algorithm. In Dijkstra's algorithm, the nodes become fixed in the order of the cost of the shortest path to them. In the proposed algorithm, a node may become fixed even when nodes with shorter cost have not been discovered. In Fig. 9.1, node v_1 becomes fixed earlier than nodes v_3 and v_4. This feature is especially useful when we are interested in finding the shortest path to a single destination and that destination becomes fixed sooner than it would have been in Dijkstra's algorithm.

Dijkstra's algorithm maintains a distance vector *dist* such that it is always feasible, i.e., for any vertex v there exists a path from source to v with cost less than or equal to $dist[v]$. We maintain the invariant that the cost of the shortest path from source to v is guaranteed to be at least $G[v]$. Therefore, in Dijkstra's algorithm, $dist[v]$ is initialized to ∞ whereas we initialize G to 0. Dijkstra's algorithm and indeed many algorithms for combinatorial optimization, such as simplex, start with a feasible solution and march towards the optimal solution, our algorithm starts with an extremal point in search space (even if it is infeasible) and marches towards feasibility. Also note that in Dijkstra's algorithm, $dist[v]$ can only decrease during execution. In our algorithm, G can only increase with execution.

The *ShortestPath* algorithm has a single variable G. All other predicates and functions are defined using this variable.

```
Shortest path from node s: LLP algorithm
  input: $pre(j)$: list of $1..n$; $w[i,j]$: positive int for all $i \in pre(j)$
  init: $G[j] := 0$;
  always: $parent[j,i] = (i \in pre(j)) \wedge (G[j] \geq G[i] + w[i,j])$;
      $fixed[j] = (j = s) \vee (\exists i : parent[j,i] \wedge fixed[i])$
      $Heap = \{(G[i] + w[i,k]) | (i \in pre(k)) \wedge fixed(i) \wedge \neg fixed(k)\}$;

  forbidden: $\neg fixed[j]$
  advance: $(G[j] := \max\{\min Heap, \min\{G[i] + w[i,j] \mid i \in pre(j)\}\})$
```

Figure 9.2: **Algorithm** *ShortestPath* to find the minimum cost assignment vector less than or equal to T.

We get the following structural result on the assignment vectors that are feasible.

Lemma 9.6 *Let G and H be two assignment vectors such that they satisfy B_{rooted}, then $min(G,H)$ also satisfies B_{rooted}.*

Proof: Follows directly from Lemma 2.3, because B_{rooted} is a lattice-linear predicate.

■

The set of feasible assignment vectors is not closed under the join operation. In Fig. 9.1, the vectors $(0,10,3,14,8)$ and $(0,9,10,12,11)$ are feasible, but their join $(0,10,10,14,11)$ is not feasible.

We now consider the generalization of the shortest path algorithm with constraints. We assume that all constraints specified are lattice-linear. For example, consider the constraint that the cost of vertex i is at most cost of vertex j. The predicate $B \equiv G[j] \geq G[i]$ is easily seen to be lattice-linear. If any cost vector G violates B, then the component j is forbidden (with $\alpha(G,j)$ equal to $G[i]$). The predicate $(G[i] = G[j])$ is also lattice-linear because it can be written as a conjunction of two lattice-linear predicates $(G[i] \geq G[j])$ and $(G[j] \geq G[i])$. The predicate $B \equiv (G[i] \geq k) \Rightarrow (G[j] \geq m)$ is also lattice-linear. If any cost vector violates B, then we have $(G[i] \geq k) \wedge (G[j] < m)$. In this case, the component j is forbidden with $\alpha(G,j)$ equal to m. Again, from Lemma 2.6, the algorithm *LLP* can be used to solve the constrained shortest path algorithm by combining *forbidden* and α for constraints with B_{rooted}. An application of the constrained shortest path problem is as follows. Suppose that there are n dispatch trucks that start from the source vertex at time 0. Let $w[i,j]$ denote the time it takes for a truck to go from node i to node j. An assignment vector G is feasible if it is possible to design a tree rooted at the source vertex such that the path from the source vertex to vertex i takes $G[i]$ units of time and G satisfies specified constraints. In Fig. 9.1, the vector $(0,9,2,8,7)$ is feasible, but does not satisfy the constraint that $G[1]$ equals $G[2]$. The least vector that satisfies this additional constraint is $(0,9,9,12,11)$. LLP algorithm can be used to find this vector. When the set of additional constraints is empty, we get back the standard shortest path problem.

An example of a predicate that is not lattice-linear is $B \equiv G[i] + G[j] \geq k$. If the predicate is false for G, then we have $G[i] + G[j] < k$. However, neither i nor j may be forbidden. The component i is not forbidden because if $G[i]$ is fixed but $G[j]$ is increased, the predicate B can become true. Similarly, j is also not forbidden.

9.4 Delta-Stepping Algorithm

A popular practical parallel algorithm for SSSP is Δ-stepping algorithm due to Meyer and Sanders [MS03]. Meyer and Sanders also provide an excellent review of prior parallel algorithms in [MS03]. They classify SSSP algorithms

Algorithm LLP-ShortestPath: An Implementation of Algorithm $ShortestPath_b$ to find the minimum cost assignment vector greater than or equal to F .

```
var $G$: array$[0..n-1]$ of real initially $\forall i : G[i] = 0$;
while $\exists i : \neg fixed(i, G)$ do
    $E' := \{ (i,k) \in E \mid fixed(i,G) \wedge \neg fixed(k,G)\}$;
    if $(E' = \phi)$ then return "non-fixed nodes not reachable";
    Let $(i^*, j^*) \in E'$ minimize $G[i] + w[i,j]$;
    $G[j^*] := \beta(G)$;
    for all $j \neq j^*$ such that $forbidden(G,j)$ in parallel do:
        $G[j] := \alpha(G,j)$;
endwhile;
return $G$;

$parent(j,i,G) \equiv (i \in pre(j)) \wedge (G[j] \geq G[i] + w[i,j])$
$fixed(j,G) \equiv (j = 0) \vee (\exists i : parent(j,i,G) \wedge fixed(i,G))$
$\beta(G) = \min\{G[i] + w[i,j] \mid (i,j) \in E'\}$
$forbidden(G,j) \equiv \neg fixed(j,G)$
$\alpha(G,j) = \max\{\beta(G), \min\{G[i] + w[i,j] \mid i \in pre(j)\}\}$
```

as either *label-setting*, or *label-correcting*. Label-setting algorithms, such as Dijkstra's algorithm, relax edges only for fixed vertices. Label-correcting algorithms may relax edges even for non-fixed vertices. Δ-stepping algorithm is a label-correcting algorithm in which eligible non-fixed vertices are kept in an array of buckets such that each bucket represents a distance range of Δ. During each phase, the algorithm removes all vertices of the first non-empty bucket and relaxes all the edges of weight at most Δ. Edges of higher weights are relaxed only when their starting vertices are fixed. The parameter Δ provides a trade-off between the number of iterations and the work complexity. For example, when Δ is ∞, the algorithm reduces to Bellman-Ford algorithm where any vertex that has its D label changed is explored. When Δ equals 1 for integral weights, the algorithm is a variant of Dijkstra's algorithm. There are many practical large-scale implementations of the Δ-stepping algorithm (for instance, by Madduri et al [MBBC07]) in which authors have shown the scalability of the algorithm. Chakravarthy et al [CCM$^+$17] give another scalable implementation of an algorithm that is a hybrid of the Bellman-Ford algorithm and the Δ-stepping algorithm.

We now give the pseudo-code of the Delta-stepping algorithm by Meyers and Sanders [MS03]. The algorithm uses a parameter Δ. We first classify all edges $(i,j) \in E$ to be *light* or *heavy* depending upon whether its weight is less than Δ or not. Formally,

$$E_{light} = \{(i,j) \in E \mid w[i,j] \leq \Delta\}$$

E_{heavy} is simply the complement of E_{light}.

The variable req is the set of all edges that are required to be relaxed. The variable S is used to maintain the set of vertices whose outgoing light edges have been relaxed but not the heavy edges. The variable $B[i]$ denotes the bucket B_i.

The algorithm DeltaStepping explores buckets B_i sequentially in the increasing order. The variable i denotes the current bucket. The outer while loop iterates until there is some bucket. The inner while loop iterates until the current bucket becomes empty. To explore a bucket, the algorithm first explores all the light edges outgoing from the bucket. When these edges are relaxed, some additional vertices may move to the current bucket. Hence, the while loop is executed until all the light edges have been explored. We use S to store all the vertices whose light edges have been explored but the heavy edges remain to be explored. Once the inner while loop terminates, we explore all the heavy edges outgoing from S. Note that exploration of all heavy edges can never result in introduction of any vertex into the current bucket. After the exploration of all heavy edges, we go to the outer while loop to explore the next bucket.

Algorithm DeltaStepping: A Parallel Delta Stepping Algorithm for the Shortest Path Problem

```
var
    $d$: array[0..n-1] of int initially $\forall i : d[i] = \infty$
    $req, S$: set of (int, int)
    $B$: sequence of buckets (sets)
relax(0,0);
int $i := 0$;
while $B \neq \emptyset$
    $S := \{\}$
    while $B[i] \neq \{\}$
        $req := \{(v', d[v] + w[v, v'] \mid v \in B[i] \wedge (v, v') \in E_{light}\}$
        $S := S \cup B[i]$
        $B[i] := \{\}$
        forall $(v', d') \in req$ in parallel do: $relax(v', d')$
    endwhile
    $req := \{(v', d[v] + w[v, v'] \mid v \in S \wedge (v, v') \in E_{heavy}\}$
    forall $(v', d') \in req$ in parallel do: $relax(v', d')$
    $i := i + 1$
endwhile
function relax(int $u$, int $c$)
    if $c < d[u]$
        $d[u] := c$
        move $u$ to the bucket $B[c/\Delta]$
endfunction
```

9.5 Graphs with Negative Weights: Johnson's Algorithm

We now consider directed graphs which have negative weight edges. For such graphs, the shortest path may not be defined if there is a cycle with negative cost. By going around the cycle, one can decrease the cost of the path arbitrarily. Therefore, we assume that even though there are edges with negative costs, there are no negative cost cycles. Our strategy for finding the shortest path in such a graph X would be to convert it into another graph Y on the same set of vertices such that Y has all non-negative edges and it preserves all shortest paths, i.e., a path is shortest in X iff it is also a shortest path in Y. The graph Y has the same set of vertices and edges as X. The weight of any edge (i, j) is updated as follows:

$$w'[i, j] = w[i, j] + p[j] - p[i] \tag{9.1}$$

where p is a *price* vector associated with vertices. A price vector p is a non-negative vector such that when we compute new costs of edges, called *reduced* costs, we get that the new cost of every edges is at least 0. The advantage of updating weights using Equation 9.1 is that it preserves shortest paths.

Lemma 9.7 *Let s and t be any two vertices in the graphs. The weight of any path in the graph Y from s to t equals the weight in the graph X plus $(p[t] - p[s])$.*

Proof: Let the path be v_0, v_1, v_2, v_k where $v_0 = s$ and $v_k = t$. As we compute the cost of the path in Y, we add the price of every intermediate vertex once and subtract it once. Only the price of v_0 is subtracted exactly once and the price of v_k is added exactly once giving us the lemma.

■

Since the cost of all paths between s and t are changed by the same amount, it follows that any shortest path in X is a shortest path in Y and vice-versa. Now our task is reduced to finding a price vector such that $w'[i, j]$ is at least 0 for all edges. We use LLP algorithm to find such a vector. Our feasibility predicate B for pricing vector is

$$\forall (i, j) \in E : w[i, j] + p[j] - p[i] \geq 0$$

Furthermore, we require $p[i] \geq 0$ for all i. We first show that B is lattice linear.

Lemma 9.8 *Let X be any graph such that the edge (i, j) has weight $w[i, j]$ and every vertex i has price $p[i]$. Consider the lattice of all non-negative price vectors. Then, the predicate*

$$B \equiv \forall (i, j) \in E : w[i, j] + p[j] - p[i] \geq 0$$

is lattice linear.

Proof: Since lattice linearity is closed under conjunction, it is sufficient to show that $B_e \equiv w[i, j] + p[j] - p[i] \geq 0$ is lattice linear for arbitrary edge $e = (i, j)$. B_e can be rewritten as $p[j] \geq p[i] - w[i, j]$. The right hand side of this inequality is a monotone function on p and hence from the Key Lemma of lattice-linearity, we get that B_e is lattice linear.

■

By applying LLP algorithm, we get the following method to find the price vector.

Since we have not provided the stopping point in Algorithm LLP-Johnson, the method may not terminate. Indeed, if there is a cycle in the graph with negative cost, the above algorithm will not terminate. We show

Lemma 9.9 *Algorithm LLP-Johnson terminates iff there is no negative cost cycle in the graph X.*

Algorithm LLP-Johnson: Finding the minimum price vector.

```
1 input: $pre(j)$: list of $1..n$; $w[i,j]$: int for all $i \in pre(j)$
2 init: $p[j] := 0$;
3 ensure: $p[j] \geq \max\{p[i] - w[i,j] \mid i \in pre(j)\}$
```

Proof:

We first show that if there is a negative cost cycle, then the algorithm never terminates. By considering s and t to be the same vertex, Lemma 9.7 implies that the cost of any cycle remains unchanged after relabeling. Hence, if there is any negative cost cycle in X, then for all pricing vectors the cost of the cycle is negative in Y. For any cycle to have a negative cost, at least one edge must be negative. Hence, for all pricing vectors there exists i, j such that $w'[i,j] = w[i,j] + p[j] - p[i] < 0$. This implies that there exists j such that $p[j] < p[i] - w[i,j]$. Hence, $forbidden(p, j)$ holds.

Now assume that there is no negative cost cycle in the graph. Since LLP algorithm finds the least feasible element of the lattice that satisfies the feasibility predicate B, it is sufficient to show that there exists at least one feasible solution. Consider an additional vertex z added to the graph such that z has a directed edge to every node in the graph with cost 0. Since there is no incoming edge to z, adding z cannot introduce a negative cost cycle in the graph. Since there are no negative cost cycles, there are finite number of paths with the least cost from z to any vertex x. The cost of the shortest path from z to any vertex is 0 or smaller because there is a direct edge from z to any vertex with weight 0. Let $d[z,x]$ be the distance of any vertex x from z. We define $p[x]$ to be $-d[z,x]$ and claim that $w[i,j] + p[j] - p[i] \geq 0$ for all edges (i,j). If not, we get that $w[i,j] - d[z,j] + d[z,i] < 0$. However, this implies that $d[z,j] > d[z,i] + w[i,j]$ which violates that $d[z,j]$ and $d[z,i]$ correspond to the cost of the shortest paths from z since by first going through i, the cost of reaching j can be reduced.

■

The proof of Lemma 9.9 shows that there always exists a price vector that makes edge weights non-negative. From the Key Lattice-Linearity Lemma, we know that the set of all feasible price vectors are closed under meets. It can also be observed that if the price vector p is feasible, and that if we add a fixed positive constant to all prices, then all prices stay positive and all the reduced weights stay the same. This observation also implies that the least price vector p that satisfies B must have at least one component which is zero. Otherwise, we can subtract the least price from all prices and still get a feasible price vector contradicting that p is the least feasible price vector.

We now bound the number of iterations in the LLP algorithm required to find the price vector.

Lemma 9.10 *The LLP algorithm requires at most n iterations before termination.*

Proof: We claim that after i iterations of LLP algorithm, $p[x]$ is negative of the cost of any path from any vertex y to x with at most i edges. Initially, when i equals 0, $p[x]$ equals 0 which is equal to the negative of the shortest path (from x to itself). Assume that the claim holds for $i-1$ iteration. On i^{th} iteration, we show that $p[x]$ is updated appropriately. Consider any shortest path of length i with y as the predecessor vertex in that path. If this path is shorter than any path of length $i-1$, we have that $-p[y] + w[y,x] \leq -p[x]$. However, this condition is equivalent to $p[x] < p[y] - w[y,x]$ guaranteeing that $p[x]$ is forbidden and increased so that $p[x] = p[y] - w[y,x]$.

Since any shortest path can have at most length n, the lemma follows.

■

Since each LLP iteration can be computed in $O(m)$ time, we get that the graph can be "reweighted" in $O(mn)$ time. Now we can use Dijkstra's algorithm to find the shortest path in the transformed graph for a single source shortest path problem. Since the algorithm for reweighting the graph dominates Dijkstra's shortest path algorithm, we get an algorithm with $O(mn)$ complexity to find shortest paths from a vertex to all vertices in an weighted directed graph possibly with negative costs. Furthermore, all pairs shortest path problem can be solved by using

Dijkstra's algorithm from every vertex after reweighting the graph. This gives us an all pairs shortest path algorithm with the time complexity of $O(mn + n(m + n \log n) = O(mn + n^2 \log n)$. This algorithm has lower time complexity than $O(n^3)$ Floyd-Warshall algorithm whenever the graph is not dense.

9.6 Bellman-Ford's Algorithm

In this section we give an algorithm that computes the shortest path tree from any given vertex without first converting it into graph with non-negative edges. We assume that the graph does not have any negative cost cycle. Given an such graph and a distinguished vertex v_0, we know that the distance of v_0 to itself can never be less than 0. Hence, without loss of generality, we will assume that v_0 does not have any incoming edge (or, simply delete those edges). Now, we consider the distance of other vertices from s. To apply LLP algorithm, we view finding the optimal distance vector as finding the largest $d[x]$ for every vertex $x \neq v_0$ which satisfies the following feasibility predicate:

$$\text{for all edges } (i,j) \in E : d[j] \leq d[i] + w[i,j]$$

If we define B_e for any edge $e = (i,j)$ as $d[j] \leq d[i] + w[i,j]$. The feasibility predicate B can be written as

$$B \equiv \wedge_{e \in E} B_e$$

We will view the search for optimal distance vector as searching for d vector in a lattice defined as follows. The lattice has the bottom element as $(M, M, \ldots, M)$ where M equal $n-1$ times the largest weight edge in the graph. Since we are looking for the largest distance vector satisfying B, it can never be greater than M. In this lattice $d \leq_L d'$ iff $\forall i : d[i] \geq d'[i]$. Thus, finding the least vector in this lattice corresponds to finding the largest vector d that satisfies B.

We now claim that

Theorem 9.11 *B is a lattice linear predicate.*

Proof: It is sufficient to show that B_e is lattice linear because any conjunction of linear predicates is also lattice linear. B_e for $e = (i,j)$ is equivalent to $d[j] \leq d[i] + w[i,j]$. This is equivalent to $d[j] \geq_L d[i] + w[i,j]$. Since the right hand side is a monotonic function of d, we get that B_e is lattice linear.

■

Algorithm LLP-Edge-Relaxation: Finding the minimum distance vector satisfying B

```
1    input: $pre(j)$: list of $1..n$; $w[i,j]$: int for all $i \in pre(j)$
2    init: if $(j = s)$ then $d[j] := 0$ else $d[j] :=$ maxint;
3    ensure: $d[j] \leq \min\{d[i] + w[i,j] \mid i \in pre(j)\}$
```

Algorithm LLP-Edge-Relaxation gives the correct answer; however, it may result in large computation complexity if d are lowered in a non-disciplined manner. We now optimize this algorithm. Algorithm LLP-BellmanFord chooses forbidden indices more carefully. Whenever there is a forbidden vertex, all forbidden vertices are advanced. The *while* loop checks if there is any forbidden vertex. The *forall* loop advances all forbidden vertices.

Let us analyze the sequential time complexity of Algorithm LLP-BellmanFord. There are at most n iteration of the *while* loop. The *forall* loop can be implemented with $O(m)$ time complexity giving us the overall sequential time complexity of $O(mn)$. For the parallel time complexity, it is sufficient to observe that the *forall* loop can be implemented in $O(\log n)$ parallel time. Thus, the parallel time complexity is $O(n \log n)$.

Earlier we had assumed that the graph does not have any negative cost cycle. What if the user input is such that it has a negative cost cycle? The algorithm LLP-BellmanFord will never terminate in that case. Problem 2 asks you to modify the algorithm so that it detects if the graph has a negative cost cycle and then outputs such a message.

Algorithm LLP-BellmanFord: Finding the minimum distance vector satisfying B

```
P_j: Code for thread j
var d: array[0..n-1] of real initially $(d[0] = 0) \wedge (\forall i \neq 0: d[i] = \infty)$
always
    $forbidden(d, j) \equiv \exists i \in pre(j) : d[j] > d[i] + w[i, j]$
while $\exists j : forbidden(d, j)$
    forall $j : forbidden(d, j)$ in parallel do
        $d[j] := \min\{d[i] + w[i, j] \mid i \in pre(j)\}$;
```

9.7 Transitive Closure of a Matrix

We start with the problem of matrix multiplication of two $n \times n$ square matrices A and B, i.e., we want to compute $C = AB$. The entry $C[i, j]$ can be computed as

$$C[i, j] = \sum_k A[i, k] * B[k, j].$$

We first compute an array $D_{i,j}$ such that $D_{i,j}[k] = A[i, k] * B[k, j]$. If we have n^3 processors, then we can assume that n processors can be used for each entry $C[i, j]$. Since we have n processors, this array can be computed in $O(1)$ time. Now, $C[i, j]$ is simply computed using the reduce operation on the array $D_{i,j}$ resulting in $O(\log n)$ time complexity for the algorithm on a CREW PRAM. This algorithm takes $O(n^3)$ work. Here, we have used the standard sequential algorithm for matrix multiplication. By using more advanced algorithms, one can reduce the exponent for the work. For example, by using Coppersmith and Winograd's algorithm, the work can be reduced to $O(n^{2.376})$. However, for simplicity, we will continue to use the standard matrix algorithm in this chapter.

Now consider the case when the matrix is boolean. In this case, we can compute $C[i, j]$ from the array $D_{i,j}$ using the *OR* operation. On a common CRCW PRAM, this can be achieved in $O(1)$ time although on a CREW PRAM we would still need $O(\log n)$ time.

Let $A[i, j]$ be a boolean matrix of size n by n. This can also be viewed as a directed graph on n vertices such that there is an edge from i to j iff $A[i, j]$ equals 1.

We now define the reflexive transitive closure of a matrix as

$$A^* = A^0 + A^1 + A^2 + \dots A^n$$

The entry $A^*[i, j]$ equals 1 iff there is any path from i to j. Our goal is to compute the reflexive transitive closure of A. We can apply the technique of *repeated squaring* to compute the transitive closure. The ideas is to compute A^{2^k} by multiplying $A^{2^{k-1}}$ with itself. Now, it is a simple matter to compute the reflexive transitive closure. We first compute a matrix $C = I + A$. This matrix simply corresponds to adding self-loops in the graph at every vertex. Now, it is easy to show that $A^* = C^n = C^{2^{\log n}}$. Hence, reflexive transitive closure can be obtained in $O(\log^2 n)$ time on a CREW PRAM.

We now show that the problem of reflexive transitive closure can also be solved using the LLP technique. We view the problem as as searching for the least vector G indexed by tuple (i, j) where $i, j \in [0..n-1]$ such that

$$B_{closure} \equiv \forall i, j : G[i, j] \geq \max(A[i, j], \max\{G[i, k] \wedge G[k, j] \mid k \in [0..n-1]\}).$$

Since the right hand side of the inequality is a monotone function of G, it follows that $B_{closure}$ is lattice-linear. We can apply *LLP* algorithm to compute the transitive closure. We have a boolean lattice of all vectors of size $O(n^2)$. Our goal is to find the least vector in this lattice that satisfies $B_{closure}$. The corresponding *LLP* algorithm is shown in Fig. LLP-Transitive-Closure

Let us translate this high level algorithm to a parallel computer with n^3 processors. A processor is indexed by (i, j, k) in the algorithm shown in Fig. 9.3. The processor checks if $G[i, j]$ is less than $G[i, k] \wedge G[k, j]$ and sets $G[i, j]$

Algorithm LLP-Transitive-Closure: A High-Level Algorithm to find the transitive closure of a matrix

```
1 input: $A$: matrix of boolean; // $A[i,j]$ is 1 if there is an edge from $i$ to $j$;
2 var $G$: matrix of boolean initially $\forall i,j : G[i,j] = A[i,j]$;
3 forbidden$(i,j)$: $(\exists k : G[i,j] < G[i,k] \wedge G[k,j])$
4     advance: $G[i,j] := 1$;
```

to 1 whenever the condition is met. Since for the same value of i and j, there may be multiple values of k for which the condition is true, multiple processors may set $G[i,j]$ to 1 concurrently. We will assume Common CRCW PRAM model for this computation. In such a model, it is easy to see that after $\log n$ steps, G matrix will correspond to the transitive closure. This is because the diameter of the graph will reduce by a factor of two after every step. On a CREW PRAM, we can compute the reflexive transitive closure in $O(\log^2 n)$ time.

```
input: $A$: matrix of boolean; // $A[i,j]$ is 1 if there is an edge from $i$ to $j$;
output: transitive closure of $A$;

var $G$: matrix of boolean initially $\forall i,j : G[i,j] = A[i,j]$;
    for $t := 1$ to $\lceil \log n \rceil$ do
        forall $i, j, k$ in parallel do
            if $(G[i,j] < G[i,k] \wedge G[k,j])$ then
                $G[i,j] := 1$;
        endforall;
    endfor;
return $G$; // the transitive closure
```

Figure 9.3: A Synchronous Parallel Algorithm with parallel time complexity $O(\log n)$ to find the transitive closure of a matrix A

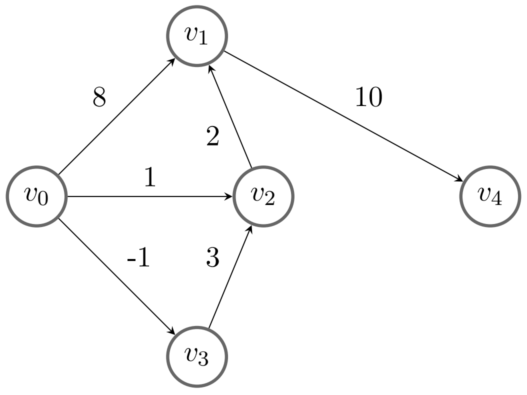
9.8 All Pairs Shortest Path

We now extend the algorithm to nonbinary matrices. 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.

Let us formulate this problem as finding the least vector of size n^2 in a distributive lattice. We initialize G as A. This is the bottom element of our distributive lattice. Our goal is to find the least vector that satisfies the following constraint:

$$B_{Floyd-Warshall} \equiv \forall i,j : i \neq j : G[i,j] \leq \min_k \{G[i,k] + G[k,j]\}.$$

When i equals j, $G[i,j]$ equals 0 and will stay fixed through the execution of the algorithm. The cost of going from i to j must be smaller than the cost of going from i to k and k to j. Fig. LLP-Floyd-Warshall gives a high-level LLP algorithm for all pairs shortest path. Fig. 9.4 gives an implementation that takes $O(\log^2 n)$ time and $O(n^3 \log n)$ work with $O(n^3)$ processors.



Algorithm LLP-Floyd-Warshall: A High-Level Algorithm to find the shortest cost matrix for a directed graph

```
1 input: $A$: matrix of real; // $A[i, j]$ is the weight of the edge from $i$ to $j$;
2 var $G$: matrix of real initially $\forall i, j : G[i, j] = A[i, j]$;
3 forbidden$(i, j)$: $(\exists k : G[i, j] > G[i, k] + G[k, j])$
4     advance: $G[i, j] := \min_k\{G[i, k] + G[k, j]\}$;
```

9.9 Summary

The following table lists all the algorithms discussed in this chapter.

```
input: $A$: matrix of real; // $A[i, j]$ is the weight of the edge from $i$ to $j$;
output: matrix $G$ such that $G[i, j]$ is the cost of the shortest path from $i$ to $j$;

var $G$: matrix of real initially $\forall i, j : G[i, j] = A[i, j]$;
    for $t := 1$ to $\lceil \log n \rceil$ do
        forall $i, j$ in parallel do
            if $(G[i, j] > min_k(G[i, k] + G[k, j]))$ then
                $G[i, j] := min_k(G[i, k] + G[k, j])$;
        endforall;
    endfor;
return $G$;
```

Figure 9.4: A Synchronous Parallel Algorithm with $O(\log^2 n)$ time and $O(n^3 \log n)$ work to find the Shortest Cost Matrix of a Graph

Problem	**Algorithm**	**Parallel Time**	**Work**
Shortest Path	Dijkstra	$O(m)$	$O(m \log n)$
Shortest Path	LLP-Dijkstra	$O(m)$	$O(m \log n)$
Shortest Path	Delta-Stepping	$O(m)$	$O(m \log n)$
Graph Conversion	Johnson	$O(n)$	$O(mn)$
Shortest Path	LLP-Bellman-Ford	$O(n)$	$O(mn)$
All Pairs Shortest Path	Floyd-Warshall	$O(\log^2 n)$	$O(n^3 \log n)$
Transitive closure	LLP-Transitive-Closure	$O(\log^2 n)$	$O(n^3 \log n)$

9.10 Problems

1. Prove Lemma 9.1.
2. Modify LLP-BellmanFord so that it detects if the input graph has a negative cost cycle and then outputs such a message.
3. Give an NC algorithm that that finds all strongly connected components in a directed graph.
4. Give an NC algorithm that checks whether the input directed graph is acyclic.
5. Give an NC algorithm to topologically sort an acyclic graph.
6. Give an NC algorithm to find all nodes reachable from a given vertex.

9.11 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 10

The Minimum Spanning Tree Problem

10.1 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.

The problem of finding minimum spanning tree is a canonical problem for which a greedy approach works. In this chapter, we first present two greedy sequential algorithms: Kruskal's algorithm and Prim's algorithm. Both of these algorithms are sequential in nature because they pick one edge at a time. They are greedy in different ways. Kruskal's algorithm chooses the least cost edge that is feasible as its next edge. Prim's algorithm chooses the next vertex that can be added to the existing tree with the least cost. Next, we discuss parallel algorithms for the minimum spanning tree problem. We first discuss a strategy to parallelize Prim's algorithm. In particular, instead of adding and exploring one vertex to the tree at a time, we show a strategy which allows multiple vertices to be added and explored in parallel. We also discuss Boruvka's algorithm which also chooses multiple edges at a time. Boruvka's algorithm maintains multiple *fragments* and adds edges to all of them in every iteration. 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. 10.1. 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 10.1 *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.*

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 10.1, the edge with weight 7 can be added to the edges with weight 3 and 4 to grow the fragment.

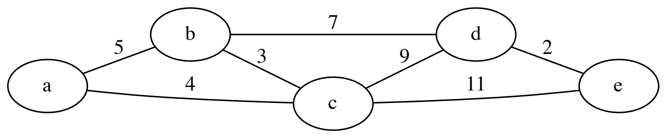


Figure 10.1: An undirected weighted graph

10.2 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.

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 $T \cup Rejected = E$ then return null; // no spanning tree in the graph
7      $e :=$ the least weight edge that is not in $T \cup Rejected$
8      if $e$ forms a cycle in $T$ then
9          $Rejected := Rejected \cup \{e\}$
10     else $T := T \cup \{e\}$
11 endwhile
12 return $T$
```

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. 10.1. 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 weight 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 reader is referred to [CLRS01] for more details on the efficient implementation of the algorithm.

We now give a parallel version for Kruskal's algorithm based on the simple boolean lattice of all edges. Our distributive lattice is the boolean lattice formed from all edges. We assume that edges are given to us in the increasing order. Then, an edge e_j between vertices v_k and v_l is in the minimum spanning tree iff there is no path between vertices v_k and v_l using edges $e_1 \dots e_{j-1}$. Note that when j is 1, e_j corresponds to the lightest edge and is always in the minimum spanning tree. An LLP version is presented as Algorithm LLP-Kruskal. The parallel time complexity of this algorithm is $O(m)$ where m is the number of edges in the graph if we are given edges in the sorted order.

Algorithm LLP-Kruskal: Finding the minimum spanning tree in a graph.

```
var $G$: array[1..$m$] of $\{0,1\}$;
// Edges $G$ is assumed to be in increasing order of weights
init $\forall j : G[j] := 0$;
forbidden(j):
    ($G[j] = 0$) and there exists no path from $v_k$ to $v_l$ with edges $1..j-1$ for the edge $e_j = (v_k, v_l)$
advance(j): $G[j] := 1$;
```

10.3 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 10.1 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. 10.1 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 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 $O(n)$.

10.4 An LLP based algorithm for Prim

We now show an LLP based algorithm to find a minimum spanning tree rooted at a fixed vertex v_0. As before, we assume that all edge weights are unique. Every node other than v_0 has to choose an edge that we call G edge. A node keeps all its edge in the sorted order and begins by choosing its least edge as the proposed edge as its G edge. For the graph in Fig. 10.1, with root as the vertex a, we get the following choices of edges for other vertices.

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 $\infty$; //$d[v]$ 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     $v := arg\ min_i\{d[i] \mid \neg fixed[i]\}$ ;
11     **if** $d[v] = \infty$ then return null;// no spanning tree in the graph
12     **if** $parent[v] \neq -1$ then add $(v, parent[v])$ to $T$
13     $fixed[v] := true$;
14     **forall** $(v, v') \in E$:
15         **if** $w[v, v'] < d[v']$ then
16             $d[v'] := w[v, v']$;
17             $parent[v'] := v$;
18     **end** // forall
19 **end** //while
```

```
b: 3, 5, 7
c: 3, 4, 9, 11
d: 2, 7, 9
e: 2, 11
```

We let $G[i]$ be the edge chosen by the node i. Thus, initially G vector is $\{3, 3, 2, 2\}$. Observe that the set of all possible combinations of edges forms a distributive lattice. Since every node except v_0 has exactly one outgoing edge, by following the chosen edge from every node we either reach v_0, or we end up in a cycle. We define a vertex to be *fixed* if by traversing the path starting from the edge proposed by that vertex leads to v_0. The vertex v_0 is trivially fixed. It is clear that the edge proposed by a non-fixed vertex can only lead to a non-fixed vertex and the edge proposed by a fixed vertex can only lead to a fixed vertex. Thus, any G partitions the set of vertices into *fixed* and *non-fixed* vertices. In our example, the initial vector $\{3, 3, 2, 2\}$ makes vertex a as fixed and all other vertices $\{b, c, d, e\}$ as non-fixed.

We define the set of edges between these two partitions as follows:

$$E'(G) := \{(i, k) \in E \mid fixed(i, G) \wedge \neg fixed(k, G)\}$$

In our example, we get the following edges as $E'(G) = \{(a, b), (a, c)\}$.

Let $x = (i, j)$ be the edge in E' with minimum $w[i, j]$. For our example, it is (a, c) with edge weight of 4. We claim that process j is forbidden in G. Consider any H such that $H[j] < w[i, j]$. Suppose if possible H forms the minimum spanning tree. Any spanning tree must have an edge y between the set $fixed(G)$ and *non-fixed*(G). We claim that H cannot be the minimum spanning tree. This claim holds because the edge set $H - \{y\} + \{x\}$ is also a spanning tree and has a lower weight than H. Therefore, unless process j advances to the edge (i, j), G cannot be the minimum spanning tree.

When we advance $G[j]$ to that edge, v_j becomes fixed. Observe that additional nodes may become fixed if their proposed edge lead to v_j directly or indirectly. The algorithm continues to advance G until either all the vertices become fixed or the edge set $E'(G)$ is empty. In the latter case, the graph is not connected and there is no minimum spanning tree.

Algorithm LLP-Prim shows the forbidden and the advance condition.

Algorithm LLP-Prim: Finding the minimum spanning tree in a graph .

```
1 var $G$: array[1..$n-1$] of real initially $\forall i : G[i]$ = minimum edge adjacent to $i$;
2 always
3     $fixed(j, G) \equiv$ there exists a directed path from $v_j$ to $v_0$ using edges in $G$
4     $E' := \{ (i,k) \in E \mid fixed(i,G) \wedge \neg fixed(k,G)\}$;
5 forbidden(j)$\equiv \exists i : (i,j) \in E'$ such that it has minimum weight $w[i,j]$ of all edges in $E'$
6 advance(j) $G[j] := min\{w[i,j] \mid (i,j) \in E'\}$
```

10.5 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. 10.1. 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 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.

10.6 Boruvka's Algorithm: Parallel Implementation

An advantage of Boruvka's algroithm is that we get one edge for every component. Thus, if there are c components we get at least $c/2$ new tree edges in one iteration of the algorithm. In the sequential implementation, we used the graph BFS traversal to determine the component ids for each node. The BFS traversal is work-efficient but takes as much time as the diameter of the graph. We now show a strategy that takes $O(\log n)$ time to carry out this step. This will give us a parallel algorithm with $O(\log^2 n)$ time complexity.

The heart of the technique is *pointer jumping*. Given any undirected graph (V,E), we first construct a *directed graph* in which we every vertex points to the neighbor connected with the least weight edge. In this graph, each

Algorithm BoruvkaSeq: Finding MST

```
Input: Undirected connected Weighted Graph: $(V, E, w)$.
Output: Minimum Weight Spanning Tree
var
    $T$: { set of edges } initially {};
    $cid$: array[1..$n$] of 0..$n$ initially all 0;
    $mwe$: array[1..$n$] of edge initially all $null$;
    $dist$: array[1..$n$] of 0..$n$ initially all $\infty$;
while ($|T| < n - 1$) do
    $visited$: array[1..$n$] of boolean initially all $false$;
    for $i := 1$ to $n$ do
        if ($\neg visited[i]$)
            // do a BFS in the graph $(V, T)$ from vertex $i$ setting $cid$ of every visited vertex to $i$
            $BFS(i)$;
    for $(i, j) \in E$ such that ($cid[i] \neq cid[j]$) do
            if $w[i, j] < dist[cid[i]]$
                $dist[cid[i]] = w[i, j]$
                $mwe[cid[i]] = (i, j)$
            if $w[i, j] < dist[cid[j]]$
                $dist[cid[j]] = w[i, j]$
                $mwe[cid[j]] = (i, j)$
    forall $i$ do:
        $T := T \cup mwe[cid[i]]$;
endwhile
return $T$
```

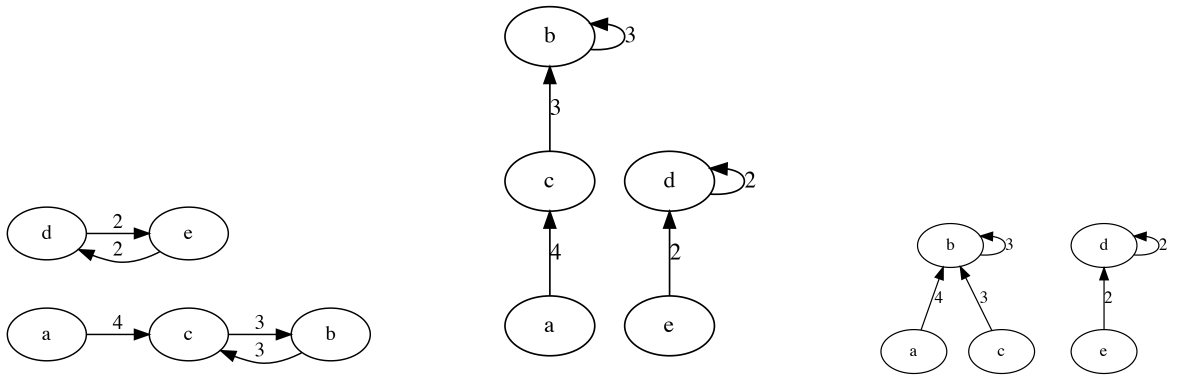


Figure 10.2: (a) Pseudo-tree from the graph in Fig. 10.1 . (b) Rooted-trees (c) Rooted-stars

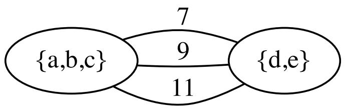


Figure 10.3: Graph after all rooted stars are contracted.

component is a **pseudo-tree**, i.e., a tree with a 2-cycle at the root.

We convert the pseudo-tree into a rooted tree and then by using pointer jumping convert the rooted tree to a rooted star. To convert a pseudo-tree into a rooted tree, we simply need to break the 2-cycle into a self-loop and a directed edge. We make the smaller of the nodes as the root with the self-loop and the bigger node points to the smaller node. Fig. 10.2 shows this conversion. The 2-cycle between b and c is converted to c pointing to b and b as the root. To convert a rooted tree to a rooted star, we simply use pointer jumping. Fig. 10.2(c) shows the rooted stars for the graph.

All the vertices in the rooted star are *contracted* into a single vertex in the graph (V, E). At this point, we have a graph in which each vertex represents a fragment of the original graph. We can now recursively find tree edges in the new graph. The algorithm terminates when there is exactly one vertex in the graph.

It is easy to check that the only step in one level of recursive call of Boruvka's algorithm that takes $O(\log n)$ time is the pointer jumping step. Every other step can be done in $O(1)$ time with m processors. Since each level takes $O(\log n)$ time and there are at most $O(\log n)$ levels, we get the parallel time complexity of $O(\log^2 n)$. Since each level takes $O(m)$ work, we get the work complexity of $O(m \log n)$. We remark here that even though we have given the algorithm in a recursive fashion, it is easy to convert it into an iterative algorithm. We leave that as an exercise (Problem 3).

Algorithm BoruvkaPar: Finding MST

```
Input: Undirected connected Weighted Graph: $(V, E, w)$.
Output: Minimum Weight Spanning Tree
function $Boruvka(V, E)$
var
    $T$: { set of edges } initially {};
    $G$: array$[1..n]$ of $0..n$ initially all 0;
if $|V| = 1$ return {};
// Get Pseudo-tree from the graph
forall $v \in V$ in parallel do
    $G[v] = w$ such that $(v, w)$ is the minimum weight edge of $v$;
    $T := T \cup \{(v, w)\}$;
// Convert it into a rooted tree
forall $v \in V$ in parallel do
    if $G[G[v]] = v$ and $v < G[v]$ then
    $G[v] := v$;
// Convert every rooted tree into a rooted star
while $(\exists j : G[j] \neq G[G[j]])$
    forall $v \in V$ in parallel do
        $G[v] := G[G[v]]$
// Contract all rooted stars into a single vertex
$V' := \{v \in V \mid G[v] = v\}$
$E' := \{(G[v], G[w]) | ((v, w) \in E) \wedge (G[v] \neq G[w])\}$
return $T \cup Boruvka(V', E')$
```

10.7 Problems

1. Show that there is a unique minimum spanning tree in any weighted undirected graph when all edge weights are distinct.
2. Suppose that you are given an undirected graph (V, E, w) such that all edge weights are distinct. You are also given a subset of edges E' such that (V, E') is acyclic. Show that there exists a unique minimum spanning tree T that is constrained to include all edges in E'.
3. Give a parallel iterative implementation of Boruvka's algorithm.

10.8 Bibliographic Remarks

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

Chapter 11

Parallel Algorithms on Trees

11.1 Introduction

In this chapter we present the *Euler Tour Technique* to get $O(\log n)$ time parallel algorithms for solving problems related to trees with n nodes. Consider the simple problem of computing the height of a tree. If the tree is a balanced binary tree, then it may be easy to think of a parallel algorithm that computes the height in $O(\log n)$ time. What if the tree is not balanced? It may seem that computing the height of a tree would require at least as much time as the height itself. However, surprisingly, we can solve the problem with a much faster algorithm. In the extreme case, when the height of the tree is $O(n)$, we would still be able to compute the height in $O(\log n)$ time. The key to this *magic* is to view the tree as a linked list and then apply parallel-prefix sum techniques.

In this chapter, we show many problems on trees for which the Euler Tour technique is applicable. Section 11.2 shows the Euler Tour technique and gives a representation of the graph which allows the technique to be applicable. Section 11.3 shows how one could construct a depth-first-tree using the Euler Tour technique. Section 11.4 shows the application to deriving the postorder numbering of vertices in a tree, Section 11.5 shows the application to deriving the vertex level for every vertex, and Section 11.6 shows the application to deriving the size of each subtree in a tree.

11.2 The Euler Tour Technique

To view a tree as a linked list of edges, we use the idea of an Euler Tour. An Euler tour of a directed graph from any node v is a walk along the edges of the graph starting from vertex v such that every edge is traversed exactly once and ending at vertex v. Since whenever a walk enters a vertex, it also leaves the vertex, it is easy to see that existence of an Euler Tour in a directed graph requires that in-degree of every vertex must be equal to the out-degree. It can be shown that equality of in-degree to the out-degree of every vertex in a strongly connected graph is also a sufficient condition for existence of an Eulerian Tour. Recall that we were concerned with listing edges of a tree and not a directed graph. So we first convert the tree into a directed graph as follows. For every undirected edge (u, v), we put two directed edges (u, v) and (v, u), one in each direction. It is clear that by our construction, any tree results in a strongly connected graph such that it has an Eulerian Tour. Furthermore, we will soon show that this tour can be constructed in $O(1)$ time in parallel!

First, let us determine a way to specify an Eulerian Tour. The idea is that it is sufficient to specify the next edge for every edge e in the directed graph. In other words, after traversing any edge e, the successor edge in the Eulerian tour is given by $next(e)$. We will assume that all vertices are numbered uniquely and therefore for any vertex u, all its neighbors are given in some fixed order. Let u be any vertex with neighbors $v_0, v_1, \ldots v_{m-1}$. Then, for any edge (v_i, u), we define $next(v_i, u) = (u, v_{i+1 \ mod \ m})$ We now have the following claim.

Theorem 11.1 *The next function defines an Eulerian Tour on the directed graph G constructed from the tree T.*

Proof: Left as an exercise.

■

We now give a more concrete implementation of Euler Tour of a tree T. Consider the tree T shown in Figure 11.1. Figure 11.2 shows the adjacency list representation of the T. In this representation, for example, node 1 has two edges: $(1, 4)$, and $(1, 5)$. In addition to the usual adjacency list representation shown in the above figure, we make two additional assumptions. First, the edge list for each node is a circular list. Hence, the edge $(1, 5)$ in the list for node 1 points to the edge $(1, 4)$ instead of pointing to null. The edge $(2, 4)$ points to itself. We will refer to the links in the circular list as black links. Second, we will assume that there are bidirectional links from edge (x, y) to edge (y, x). We refer to these links as blue links. This results in the adjacency list representation as shown in Figure 11.3. Note that every edge has exactly one outgoing black link and one blue link. Given the black and blue links, it is a simple matter to determine $next(e)$ for any edge e. The edge $next(v, u)$ is given by following one blue link and then one black link. Thus, every processor assigned to any edge can determine the next edge in $O(1)$ time.

In this example $next(1, 4) = (4, 2)$. In Figure 11.3, $next$ is the element that is shown by the blue dotted edges followed by black edges. So we can can construct the following Euler Tour directly from Figure 11.3:

$(1, 4) \rightarrow (4, 2) \rightarrow (2, 4) \rightarrow (4, 3) \rightarrow (3, 4) \rightarrow (4, 1) \rightarrow (1, 5) \rightarrow (5, 1) \rightarrow (1, 4)$

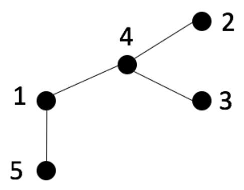


Figure 11.1: A tree T

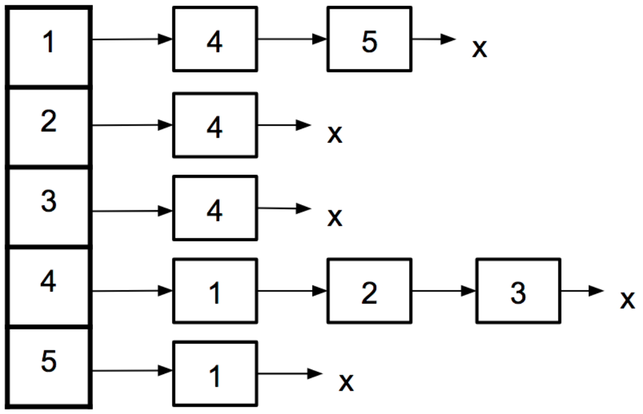


Figure 11.2: The adjacency lists of the tree T

As another example, consider T in Fig. 11.4. In the above tree, we get the following Euler Tour: $(1, 3) \rightarrow (3, 1) \rightarrow (1, 4) \rightarrow (4, 2) \rightarrow (2, 4) \rightarrow (4, 1) \rightarrow (1, 5) \rightarrow (5, 1) \rightarrow (1, 3)$.

Now that we have an Euler Tour constructed in $O(1)$ time, we show its applications.

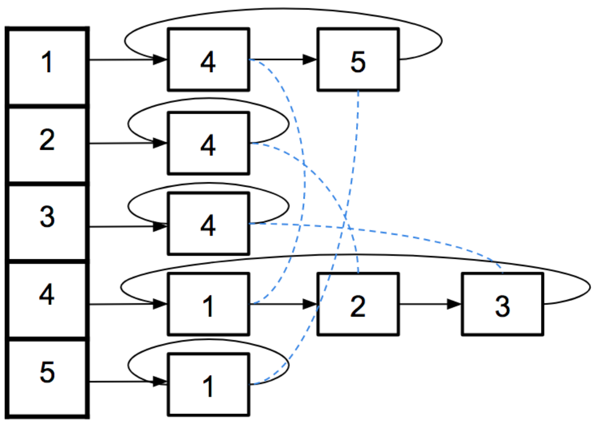


Figure 11.3: The circular adjacency lists of tree T with additional pointers

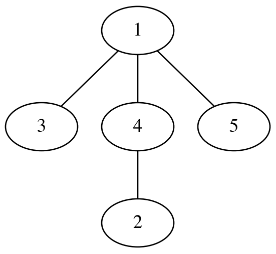


Figure 11.4: Tree with the root as node 1

11.3 Constructing DFS tree rooted at a given node r

We first construct a list called Euler Path out of Euler Tour as follows. Since the Euler Path traversal of a tree rooted at r will start from r and end at r, we make the next edge for the edge (v_{m-1}, u) to be null. Observe that Euler Path is identical to the more familiar depth-first-search (DFS) traversal of the tree. In the last tree, suppose that we are interested in the DFS traversal starting from node 1, then the next $(5, 1)$ is set to null and we get the following linked list.
$(1, 3) \to (3, 1) \to (1, 4) \to (4, 2) \to (2, 4) \to (4, 1) \to (1, 5) \to (5, 1)$.

Our goal is to set the parent pointer for each node in DFS tree rooted at r. In DFS traversal x is parent of y iff the edge (x, y) is traversed before (y, x). How do we determine which edge is traversed first? We assign weight of each edge as 1 and compute parallel prefixSum for all edges in the linked list. Now, for each directed edge (x, y), if prefixSum(x, y) is less than prefixSum(y, x), we assign parent of y as x. In our example, we get

Edges:	(1,3)	(3,1)	(1,4)	(4,2)	(2,4)	(4,1)	(1,5)	(5,1)
Weight:	1	1	1	1	1	1	1	1
prefixSum	1	2	3	4	5	6	7	8

Since prefixSum of $(4, 2)$ is less than the prefixSum of $(2, 4)$, we get that *parent*[2] equals 4. The algorithm can be summarized as follows.

The time complexity of this algorithm is dominated by computation of prefixSum, hence we have $T(n) = O(\log n)$.

```
1 Compute Euler Circuit $C$
2 Break $C$ into list by setting succ($u_{d-1}$, $r$) to 0
3 Assign weight 1 to each edge
4 Compute parallel prefix sum
5 for each edge($x$, $y$) do in parallel:
6     if prefixSum($x$, $y$) < prefixSum($y$, $x$) then parent[$y$] := $x$
```

11.4 Postorder Numbering of a Tree rooted at u

The postorder number of a node is given by the order in which a node is *visited* in the postorder traversal of a tree. The postorder traversal of a tree rooted at node r visits all subtrees of r recursively before visiting the root r.

To find the postorder traversal number of each node, we first construct Euler path as in the previous section. Then, we assign the weight of 1 to edge that corresponds to an edge $(v, parent[v])$ because that edge is traversed when the subtree rooted as v has been traversed. Then, the post-order number of a node v is given by $prefixSum(v, parent[v])$.

The algorithm can be summarized as follows.

```
1 Compute Euler Path for T rooted at $r$
2 for each $v \neq r$ do in parallel:
3     w($v$, p($v$)) := 1
4     w(p($v$), $v$) := 0
5 Compute parallel prefix sum
6 for each $v \neq r$ do in parallel:
7     post[$v$] := prefixSum($v$, p($v$))
8 post[$r$] = $n$
```

In this example, we get the following prefix sums.

Edges:	(1,3)	(3,1)	(1,4)	(4,2)	(2,4)	(4,1)	(1,5)	(5,1)
Weight:	0	1	0	0	1	1	0	1
prefixSum	0	1	1	1	2	3	3	4

the post-order number of 3 is 1 because $prefixSum(3, parent[3]) = prefixSum(3, 1) = 1$.
The post-order number of 2 is 2 because $prefixSum(2, parent[2]) = prefixSum(2, 4) = 2$.
The post-order number of 4 is 3 because $prefixSum(4, parent[4]) = prefixSum(4, 1) = 3$.

11.5 Computing Vertex Level for a Tree rooted at u

If we have the parent information it is also easy to determine the level of any vertex in the tree. The level of the root is always 0. Whenever we traverse an edge $(parent[v], v)$, we are going from the parent of v to v, hence the level number increases by 1. On the other hand, when we traverse the edge $(v, parent[v])$, the level number decreases by 1. Hence, we assign weights to each edge as follows. If the edge is from a node to its child, then its weight is +1; otherwise, its weight is -1. Now, we can compute prefixSum of Euler path. The level of any non-root vertex v is given by the prefixSum of the edge $(parent[v], v)$.

The algorithm can be summarized as follows.

```
Compute Euler Path for T rooted at r
for each $v \neq r$ do in parallel:
    w($v$, p($v$)) := -1
    w(p($v$), $v$) := 1
Compute parallel prefix sum
for each $v \neq r$ do in parallel:
    level[$v$] := prefixSum(p($v$), $v$)
level[$r$] = 0
```

For our example, we get:

Edges:	(1,3)	(3,1)	(1,4)	(4,2)	(2,4)	(4,1)	(1,5)	(5,1)
Weight:	1	-1	1	1	-1	-1	-1	1
prefixSum	1	0	1	2	1	0	1	0

Specific examples: prefixSum(1, 3) = 1 → level[3] = 1

11.6 Compute Size of Subtree for Each Node

Recall that a non-root node v is visited for the first time when the traversal reaches v from its parent and it is visited for the last time when the traversal reaches p(v) from v. The number of nodes traversed between these two visits is given by prefixSum(v, p(v)) minus prefixSum(p(v), v).

```
Compute Euler Path for T rooted at r
for each $v \neq r$ do in parallel:
    w($v$, p($v$)) := 1
    w(p($v$), $v$) := 0
Compute parallel prefix sum
for each $v \neq r$ do in parallel:
    size[$v$] := prefixSum($v$, p($v$)) - prefixSum(p($v$), $v$)
size[$r$] = $n$
```

For our example, we get:

Edges:	(1,3)	(3,1)	(1,4)	(4,2)	(2,4)	(4,1)	(1,5)	(5,1)
Weight:	0	1	0	0	1	1	0	1
prefixSum	0	1	1	1	2	3	3	4

Specific example: prefixSum(3, 1) = 1, prefixSum(1, 3) = 0. Therefore, $size[3] = 1$.

11.7 Problems

1. A preorder traversal is defined by the following recursive method.

```
preorder(node r)
```

```
   visit(r);
   for all children x of r do
      preorder(x);
end
```

Give a parallel algorithm to return the order of node visited during preorder traversal. What is the time and work complexity of your algorithm if the tree has n nodes?

11.8 Bibliographic Remarks

Reader will find parallel algorithms for tree in are [JáJ92].

Chapter 12

Divide and Conquer

12.1 Introduction

A useful strategy for solving a problem in parallel is to divide the problem into sub-problems, solve the sub-problems in parallel and then merge the solutions to the sub-problems to get the solution of the original problem. We give many examples of this approach. Viewing the problem as finding an appropriate element in a distributive lattice, the divide-and-conquer approach corresponds to viewing the problem as finding two appropriate elements in two distributive lattices. This search is carried out independently and in parallel. Once these two elements are determined, we use them to find the required element in the original lattice.

12.2 Splittable Predicates

Let L be a finite distributive lattice and B be any lattice-linear predicate. We call B splittable if there exists two lattices L_1 and L_2 and two lattice-linear predicates B_1 and B_2 such that given the least solution for B_1 and B_2 in L_1 and L_2, one can determine the least solution for B in L. The effort to determine the least solution for B in L is reduced if we have the solutions for B_1 and B_2. If B_1 and B_2 are also splittable, then we can apply this strategy recursively. Once the lattice is trivial, i.e., it is totally ordered, then finding the least solution is simple. Formally,

Definition 12.1 (Splittable Lattice-Linear Predicate) *A nonempty Lattice-Linear Predicate B is splittable for a finite distributive lattice L, if there exists two nonempty lattices L_1 and L_2 and two lattice-linear predicates B_1 and B_2 such that there exists a map $f : L_1 \times L_2 \to L$, where $f(G_1, G_2)$ is the least element satisfying B in L such that G_1 and G_2 are the least elements in L_1 and L_2 satisfying B_1 and B_2.*

As an example of a predicate that is not splittable, consider the stable marriage problem. Given the distributive lattice of all assignments, the predicate B that the assignment is a stable marriage is not splittable.

The predicate B that is true on all elements is trivially splittable into B_1 and B_2 also as true elements. The lattice L can be split into arbitrary $L_1 \times L_2$. We now give many examples of nontrivial splittable predicates.

12.3 Quicksort: Revisited

Let A be any array. Without loss of generality assume that all the entries are unique. Sorting the array is equivalent to finding the permutation such that when the permutation is applied to the array we get a sorted array. A permutation can be represented in many ways. We represent it using the inversion vector. The inversion number of any entry i as the number of entries greater than i that appear to the left of i in the permutation. Consider the lattice of all inversion vectors (show that it is distributive).

Sorting is equivalent to finding the inversion vector which sorts the array. For example, if the array is $[7, 3, 9]$, then the inversion vector is $[0, 1, 0]$ We can order the inversion vectors based on component-wise comparison. Under this order, the set of inversion vectors forms a distributive lattice L. For an array of size 3, there are six elements in the lattice L: $[0, 0, 0]$, $[0, 0, 1]$, $[0, 1, 0]$, $[0, 1, 1]$, $[0, 0, 2]$ and $[0, 1, 2]$. An inversion vector G satisfies B if when G is applied to A, it results in a sorted vector. Formally,

$$B(G) \equiv A \text{ has inversion vector } G$$

We claim that B is splittable. Let *pivot* be an arbitrary element in the array which is not an extremal element. We define L_1 to be the lattice of inversion vectors for entries in an array that are at most *pivot*. Similarly, let L_2 be the lattice of inversion vectors for entries that are strictly bigger than *pivot*. Let G_1 and G_2 be the inversion vectors for entries in L_1 and L_2, respectively. Then, G is given by the inversion vector (G_1, G_2) applied to respective entries in L_1 and L_2. As a concrete example, suppose that $A = [5, 9, 16, 4, 11]$. Let the pivot be 5. There are two entries in A that are less than or equal to *pivot*, and three entries that are bigger than *pivot*. Then, L_1 corresponds to all inversion vectors of length 2 and L_2 corresponds to all inversion vectors of length 3. G_1 equals $[0, 1]$ because the inversion vector $[0, 1]$ sorts the subarray $[5, 4]$ and G_2 equals $[0, 0, 1]$ which sorts the subarray $[9, 16, 11]$. Then, the inversion vector G equals $[0, 1, 0, 0, 1]$ applied to respective entries sorts the array. In our example, we get $[0, 1]$ applied to $[5, 4]$ and $[0, 0, 1]$ applied to $[9, 16, 11]$. Hence, the sorted array is $[4, 5, 9, 11, 16]$. Of course, sorting the array by this method requires the array to be split into two parts: entries less than or equal to pivot and the entries greater than pivot. However, in the worst case our partition may be trivial. For example, when the pivot is the greatest element in the array or the smallest element in the array, one side of the partition is just one less than the original array. We note here that in expectation, the partitioned arrays are not unevenly sized and the algorithm performs well in practice.

12.4 Mergesort: Revisited

In the above example, we first partitioned the array such that all elements on the left side of the split are less than the right side of the split. Once the left side and the right side are sorted, there is nothing else to be done. An alternative version of the split would be to split the lattice trivially, but the merging of the solution requires more effort. In our example of the sorted array $A = [5, 9, 16, 4, 11]$, suppose that we split the lattice into two lattices L_1 and L_2 such that L_1 consists of the first two components and L_2 corresponds to the last three components. The vector $S_1 = [5, 9]$ is the sorted version of the left side $[5, 9]$ and the vector $S_2 = [4, 11, 16]$ is the sorted version of the right side $[16, 4, 11]$. Now, our task is to merge the two sorted arrays. The function f in definition of *splittable lattice-linear predicate* corresponds to merging of these two arrays.

To sort an array of size n, we divide it into two subarrays of size $n/2$ and sort each of the parts in parallel. The main trick is to determine how to merge these two sorted arrays. To understand, an application of the LLP algorithm for mergesort, we consider a related problem of counting the number of inversions of each entry in an array A. Given an array A, our goal is to compute another array G such that for any index i in the array,

$$G[i] = \#\{j < i \mid A[j] > A[i]\} \tag{12.1}$$

To keep things simple, we let all entries of A be unique. To exploit divide-and-conquer, we divide the array into two equal parts (or nearly equal, if the array A is of odd size). Let these two parts be called $Left$ and $Right$. For the left part, computing G remains identical irrespective of the total size of the array, i.e., we still have Equation 12.1. However, for the right part, we have that $G[i] = \#\{(j < i) \wedge (j \in Left) \mid A[j] > A[i]\} + \#\{(j < i) \wedge (j \in Right) \mid A[j] > A[i]\}$. The second part of the above equation is identical to Equation 12.1 for the right part of the array, but we need to find the value of the first part of the equation.

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

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

Algorithm LLP-MergeSort: MergeSort

```
1 LLP-MergeSort(A, low, high)
2 if $low < high$ then
3 $mid = \lfloor (lo + high)/2 \rfloor$ ;
4 $B$ := LLP-MergeSort(A, low, mid) in parallel with $C$ := LLP-MergeSort(A, mid + 1, high)
5 return $MergeTwo(B, C)$;
6 else return $A$;
```

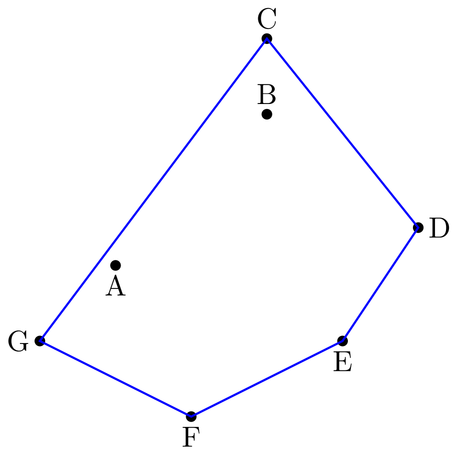


Figure 12.1: An example of Planar Convex Hull

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

Now, we focus on the parallel time complexity. To merge two sorted arrays of size $n/2$ into a sorted array of size n, we need $O(\log n)$ time. Each entry can independently determine the total number of elements less than itself via binary search. Since each subarray is sorted in parallel, we get the following recurrence:

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

Solving this recurrence, we get that the parallel merge sort runs in time $T(n) = O(\log^2 n)$. Since the procedure assumes n processors, we get that the total work done is $O(n \log^2 n)$.

12.5 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.

We first model the problem using lattice-linear predicates. Let L be the boolean lattice on S. Thus, L has size $2^{|S|}$. Our goal is to find S', the smallest subset of S, such that any point in S can be written as a convex combination of points in S'. We define a predicate B to be true on X, a subset of S, if all points in S can be written as a convex combination of points in X. Thus, our goal reduces to finding the least element in L that satisfies B. Since sorting can be achieved in $O(\log n)$ parallel time, 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

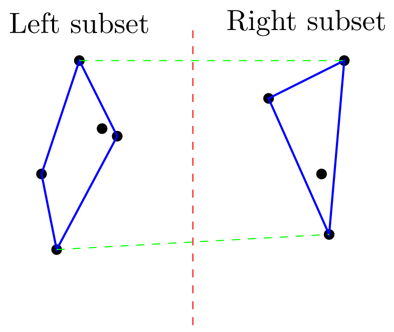


Figure 12.2: Using Divide and Conquer for Planar Convex Hull

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 in parallel. The only task left it 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 and in parallel in $O(\log n)$ time. Hence, we get the following recurrences:

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

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

Solving these recurrences, we get $T(n) = O(\log^2 n)$ and $W(n) = O(n \log n)$.

12.6 Summary

In this chapter, we have shown that many divide-and-conquer problems can be solved using a single *parallel* Lattice-Linear Predicate algorithm. Since each partition can be solved in parallel, we get an algorithm with polylog time complexity if partitioning is roughly even. The divide and conquer strategy is applicable to many other problems including: the nearest neighbor problem in the Euclidean space, computation of the Fast-Fourier Transform and Karatsuba multiplication of two numbers.

The following table lists all the algorithms discussed in this chapter.

Problem	**Algorithm**	**Parallel Time**	**Work**
Sorting	Parallel MergeSort	$O(\log^2 n)$	$O(n \log^2 n)$
Planar Convex Hull	Divide-and-conquer	$O(\log^2 n)$	$O(n \log n)$

12.7 Problems

1. Solve the problem of finding the nearest neighbor in the Euclidean space using the divide and conquer approach.
2. Solve the problem of computing the Fast-Fourier-Transform in parallel using the divide and conquer approach.

Chapter 13

Greedy Algorithms

13.1 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, one at a time until we get back the 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 doesn't 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.

When we are developing parallel algorithms, we are interested in increasing the size of the problem by more than one. This idea can effectively be modeled using LLP algorithms. There can be more than one *forbidden* index and we can advance on all forbidden indices in parallel. For the parallel version of Prim's algorithm, we added multiple edges to the current component so long as these edges are either the smallest going out of the component or the smallest going out the vertex that is being added. For the parallel version of Kruskal's algorithm, we added all edges (v_i, v_j) such that there is no path from vertex v_i to v_j with edges lighter than the edge between v_i and v_j. For the parallel version of Dijkstra's algorithm, we added multiple nodes to the current component so long as these edges are the smallest going out of the component or guaranteed the least distance from the source vertex.

In this chapter, we provide some additional examples of greedy algorithms that can be parallelized.

13.2 Activity Selection Problem

We start with a standard simple greedy algorithm. Suppose that we have m activities with the start times s_i and finish times f_i for jobs $i = 1..m.$, where $s_i \leq f_i$. We assume that activities happen during the open interval $[s_i, f_i)$. Two activities i and j are *compatible* if $f_i \leq s_j$ or $f_j \leq s_i$. Our goal is to select a maximum-size set of mutually compatible activities.

We consider the lattice of completion times for k jobs. $G[i]$ gives the completion of job i. We define $G[i]$ as the least time by which i jobs can be finished. A vector G is feasible if for all j greater than 1 there exists a job that starts after time $G[j-1]$ and finishes at time $G[j]$ and $G[1]$ gives the finishing time of the first job. Since every job takes at least one unit of time, it is clear that for all i, $G[i] < G[i+1]$ for all feasible G.

```
$P_j$: Code for thread $j$
var $G$: linked list of $[1..n]$ of $(1..maxint, 1..maxint, boolean)$;// shared among all threads
init: $\forall j : G[j].s() := s[j]; G[j].f() := f[j]; G[j].fixed := false$
$G[1].fixed := true$;

activity-selection:
    forbidden: $\neg G[this].fixed \wedge G[prev].f() < G[this].s()$
        advance: $G[this].fixed := true$;
    forbidden: $G[prev].fixed \wedge \neg G[this].fixed \wedge G[this].s() < G[prev].f()$
        advance: delete $G[this]$ from $G$;
```

Figure 13.1: LLP Parallel Program for Activity Selection problem

```
$P_j$: Code for thread $j$
var $id : 1..n; s : 1..maxint, f : 1..maxint, fixed : boolean, prev, next : 1..n$;
init: $\forall j : id := j; s := s[j]; f := f[j]; fixed := false; prev := j - 1; next := j + 1$;
if $(j = 1) fixed := true$;

activity-selection:
    forbidden: $\neg fixed \wedge prev.f < s$
        advance: $fixed := true$;
    forbidden: $prev.fixed \wedge \neg fixed \wedge s < prev.f$
        advance: prev.next := next; next.prev := prev; // delete this node from the linked list;
```

Figure 13.2: LLP Parallel Program for Activity Selection problem

The first rule states that if the current job starts after the last job finishes, then the current job is always included in the final set. The second rule states that if the current job starts before the previous job and the previous job is guaranteed to be chosen, then it is always deleted from the consideration. Observe that since we have used a linked list for all the jobs, the previous node for any node may change when a job is deleted from the list. For example, consider the following four jobs: $j_1 = [1, 4), j_2 = [2, 5), j_3 = [4, 5), j_4 = [5, 7)$. 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.

Assuming that the input job list is sorted using finishing times, the LLP algorithm takes $O(1)$ in the best case and $O(n)$ in the worst case. The sequential algorithm always takes $O(n)$ time.

We now add lattice-linear constraints to the problem.

13.3 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 variable length code for symbols rather than fixed length code.

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       $G[i,i] = p[i]$;
4 always: $s(i,j) = \sum_{k=i}^{k=j} p[i]$
5 ensure$(i,j)$:
6       $G[i,j] \geq \min_{i \leq k < j} G[i,k-1] + s(i,j) + G[k+1,j]$
```

The sequential algorithm is as follows. We choose two symbols i and j with the least probability. We give the code of 0 for i and 1 for j. We can replace symbols i and j by another symbol α that occurs with probability $p_i + p_j$. The number of symbols have now decreased by 1. If we knew the code for α, then by appending 0 or 1, we get the code for i and j. We repeat this procedure until there are only two symbols left. These two symbols can be given the code 0 and 1. The sequential algorithm can be implemented using a heap effectively. We can find and delete the two symbols i and j with the least probability using the heap and then insert α in the heap with the sum of probability for i and j. Alternatively, suppose that we are initially given a queue of symbols with increasing probability. By keeping an additional queue of *derived* symbols, we can implement the sequential algorithm in $O(n)$ time if the initial sorted queue is provided to us.

We now give an LLP algorithm for Huffman coding. Instead of computing the code for the original n symbols S, we derive another set of symbols S' such that this set is of length $n-1$ or smaller and given the code for each symbol in s', we can derive the code for symbols in S. If the set of symbols is just one, then we can stop since a single symbol does not require any bits to encode.

Algorithm LLP-HuffmanTree: Finding Huffman Code

```
1 input: $p$:array of real;// frequency of each symbol
2 init: $G[i] = p[i]\ \forall i$;
3 forbidden($i$):
4         $G[i,j] \geq \min_{i \leq k < j} G[i, k-1] + s(i,j) + G[k+1, j]$
```

13.4 Fractional Knapsack Problem

13.5 Bibliographic Remarks

Chapter 14

Dynamic Programming

14.1 Introduction

We show that many problems that can be solved using dynamic programming [Bel52] can also be solved in *parallel* using the LLP algorithm. 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. One can solve the problem using recursion; however, recursion may 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 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 be fit in a knapsack such 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 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.

The second difference between dynamic programming and the LLP algorithm is in terms of parallelism. The dynamic programming solution does not explicitly refers to parallelism in the problem. The LLP algorithm has an explicit notion of parallelism. The solution uses an array G for all problems and the algorithm requires the components of G to be advanced whenever they are found to be *forbidden*. If $G[i]$ is forbidden for multiple values of i, then $G[i]$ can be advanced for all those values in parallel.

The third difference between dynamic programming and the LLP algorithm is in terms of synchronization required during parallel execution of the algorithm. In case of dynamic programming, if the recursive formulas are evaluated in parallel it is assumed that the values used are correct. In contrast, suppose that we check for $G[i]$ and $G[j]$ to be forbidden concurrently such that $G[i]$ ends up using an old value of $G[j]$, the LLP algorithm is still correct. The only requirement we have for parallelism is that when $G[i]$ uses a value of $G[j]$, it should either be the most recent value of $G[j]$ or some prior value. A processor that is responsible for keeping $G[i]$ may get old value from $G[j]$ in a parallel setting when it gets this value from a cache. In a message passing system, it may get the old value of $G[j]$ if the message to update $G[j]$ has not yet arrived at the processor with $G[i]$. Thus, LLP algorithms are naturally parallel with little synchronization overhead.

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 14.2 applies the LLP method to the longest subsequence problem. Section 14.3 give a parallel algorithm for the optimal binary search tree construction problem. Section 14.5 gives an LLP algorithm for the knapsack problem.

14.2 Longest Increasing Subsequences

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 $B \equiv \forall j : G[j] \geq 1 \wedge \forall j : G[j] \geq \max\{G[i] + 1 \mid i \in pre(j)\}$. To understand B, we first consider a stronger predicate $B_* = (G[1] = 1) \wedge \forall j : G[j] = \max\{1, \max\{G[i] + 1 \mid i \in pre(j)\}\}$. The interpretation of $G[j]$ in B_* is that it is the length of the longest path that ends in j. Thus, in the longest increasing subsequence problem we are searching for the vector that satisfies the predicate B_*. Instead of searching for an element in the lattice that satisfies B_*, we search for the least element in the lattice that satisfies B. This allows us to solve for the constrained version of the problem in which we are searching for an element that satisfies an additional lattice-linear constraint.

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 14.1 *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 is sufficient 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. Now, LLP algorithm can be formulated as LLP-Longest-Increasing-Subsequence.

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

```
$P_j$: Code for thread $j$
input: $A$:array of int;
var $G$: array$[1 \ldots n]$ of int;
init: $G[j] = 1$;
    $pre(j) := \{i \in 1..j-1 | A[i] < A[j]\}$;
ensure: $G[j] \geq \max\{G[i] + 1 \mid i \in pre(j)\}$;
```

This algorithm starts with all values as 1 and increases the G vector till it satisfies the constraint $G[j] \geq \max\{G[i] + 1 \mid i \in pre(j)\}$. The above algorithm, although correct, does not preclude $G[j]$ from getting updated multiple times. 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. With this change, our algorithm becomes LLP2-Longest-Increasing-Subsequence.

Algorithm LLP2-Longest-Increasing-Subsequence: Finding the Longest Increasing Subsequence.

```
$P_j$: Code for thread $j$
input: $A$:array of int;
var $G$: array$[1 \ldots n]$ of int;
    $fixed$: array$[1 \ldots n]$ of boolean;
init: $G[j] = 1; fixed[j] := false$;
    $pre(j) := \{i \in 1..j-1 | A[i] < A[j]\}$;
forbidden: $\neg fixed[j] \wedge (\forall i \in pre(j) : fixed[i])$;
    advance: $G[j] := \max\{G[i] + 1 \mid i \in pre(j)\}$;
        $fixed[j] := true$;
```

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 $O(n^2)$.

For the parallel time complexity, assume that we have n^2 processors available. Then, in time $O(\log n)$, one can determine whether the vertex is forbidden and advance it to the correct value if it is forbidden. This is because for every j, we simply need to check that all vertices in $pre(j)$ are fixed and j is not fixed. By using a parallel *reduce* operation, we can check in $O(\log n)$ time whether j is forbidden. If the longest path in the graph H is Δ, then the algorithm takes $O(\Delta \log n)$ time.

Now, let us consider the situation where each thread j writes the value of $fixed[j]$ and $G[j]$ without using any synchronization. If any thread j reads the old value of $fixed[i]$ for some i in $pre(j)$, it will not update $fixed[j]$ at that point. Eventually, it will read the correct value of $fixed[i]$, and perform *advance*. We do assume in this version that if a process reads $fixed[i]$ as true, then it reads the correct value of $G[i]$, because $fixed[i]$ is updated after $G[i]$. Consequently, we get the following result.

Lemma 14.2 *There exists a parallel algorithm for the longest increasing subsequence problem which uses just read-write atomicity and solves the problem in $O(\Delta \log n)$ time.*

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 14.3 *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 $A[j]$.

■

We note here that the problem can also be solved in parallel using repeated squaring of an appropriate matrix. We do not discuss that method here since it is not work-optimal and generally not efficient in practice.

14.3 Optimal Binary Search Tree

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 $i \le k \le j$. 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

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

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

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

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

Lemma 14.4 *The constraint* $B \equiv \forall i, j : G[i, j] \ge \min_{i \le k \le j}(G[i, k - 1] + s(i, j) + G[k + 1, j])$ *is lattice-linear.*

Proof: Suppose that B is false, i.e., $\exists i, j : G[i, j] < \min_{i \le k \le j}(G[i, k - 1] + s(i, j) + G[k + 1, j])$. This means that there exists i, j, k with $i \le k \le j$ 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 $\min_{i \le k \le j} G[i, k - 1] + s(i, j) + G[k + 1, j]$. 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: $G[i, j] = 0 \ \forall i \ne j$;
4     $G[i, i] = p[i]$;
5 always: $s(i, j) = \sum_{k=i}^{j} p[k]$
6 ensure:
7     $G[i, j] \ge \min_{i \le k \le j} G[i, k - 1] + s(i, j) + G[k + 1, j]$
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 $i \le k \le j$ 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 $O(n^3)$. On a CREW PRAM, we can compute all i, j with the fixed difference in parallel. By using $O(\log n)$ span algorithm to compute min, we get the parallel time complexity as $O(n \log n)$. Thus, we have the following result.

Lemma 14.5 *There exists a parallel algorithm for the optimal binary search tree problem which uses just read-write atomicity and solves it in* $O(n \log n)$ *parallel time.*

We now consider the constrained versions of the problem.

Lemma 14.6 *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.

■

14.4 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.

14.5 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 $O(nW)$.

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 14.7 *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[$0 \ldots n$, $0 \ldots W$] of int;
4 init: $G[i,j] = 0 \ if \ (i=0) \vee (j=0)$;
5 ensure:
6 $G[i,j] \geq \max\{G[i-1,j-w_i]+v_i, G[i-1,j]\}$   if  $j \geq w_i$
7            $\geq G[i-1,j]$, 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[$0 \ldots W$] of int;
4 var: $G$:array[$0 \ldots W$] of int;
5 init: $\forall j : G[j] = C[j]$;
6 ensure:
7     $G[j] \geq C[j-w]+v$   if  $j \geq w$
```

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. If we had W cores, then computing $G[i,.]$ from $G[i-1,.]$ can be done in $O(1)$ giving us the span of $O(n)$.

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

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[$0 \ldots W$] of int;
4 init: $\forall j : G[j] = 0$;
5 for $i := 1$ to $n$ do
6       $G := IncrKnapsack2(w[i], v[i], G)$;
```

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.

Lemma 14.8 *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.

■

14.6 Summary

The following table lists all the algorithms discussed in this chapter.

Problem	Algorithm	Parallel Time	Work
Increasing subseq	LLP	$O(\Delta \log n)$	$O(n^2 \Delta \log n)$
Optimal binary tree	LLP	$O(n \log n)$	$O(n^3)$
Knapsack problem	LLP	$O(n)$	$O(nW)$

14.7 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.)*

14.8 Bibliographic Remarks

Sequential algorithms for dynamic programming can be found in [CLRS01]. LLP based parallel algorithms for dynamic programming are taken from [Gar22].

Chapter 15

The Housing Allocation Problem

15.1 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 15.2 gives Gale's Top Trading Cycle Algorithm. Section 15.3 applies LLP method to the unconstrained Housing market problem and derives a high-level parallel algorithm. Section 15.4 give a parallel Las Vegas algorithm for the Housing market problem.

15.2 Gale's Top Trading Cycle Algorithm

Consider the housing market instance shown in Fig. 15.1. 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. 15.1.

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. 15.2 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.

Agents' Preferences	Initial Allocation	Matching returned by the TTC algorithm
$a_1 : h_2, h_3, h_1, h_4$	$a_1 : h_1$	$a_1 : h_2$
$a_2 : h_1, h_4, h_2, h_3$	$a_2 : h_2$	$a_2 : h_1$
$a_3 : h_1, h_2, h_4, h_3$	$a_3 : h_3$	$a_3 : h_4$
$a_4 : h_2, h_1, h_3, h_4$	$a_4 : h_4$	$a_4 : h_3$

Figure 15.1: 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.

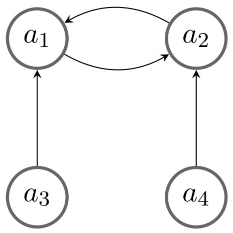


Figure 15.2: The top choice graph at the first stage.

15.3 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 15.1 (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 15.2 *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 15.3 *$submatching(G, J_1)$ 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 15.4 (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 15.5 *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

```
var
    $G$: array[1..$n$] of int initially 1;// every agent starts with the top choice
always
    $S(G)$ = largest $J$ such that $submatching(G, J)$
    $forbidden(G, j, B) \equiv (j \notin S(G)) \wedge (wish(G, j) \in S(G))$
        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 15.6 *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.

15.4 An Efficient Parallel Algorithm for the Housing Market Problem

We now present an efficient parallel algorithm for the housing market problem. We note here that [ZG19] gives a distributed algorithm with $O(n^2)$ messages for the housing market 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.

By renumbering houses, if necessary, we assume that initially agent a_i has the house h_i. We assume that the preference list is provided as two data structures: $prefList$ and $prefPointer$. The variable $prefList$ is an array of doubly linked list such that $prefList[i]$ points to the list of preferences of agent i. As the algorithm executes, we advance on $prefList$ and the head of the $prefList[i]$ corresponds to the variable $wish$ for agent a_i in Fig. LLP-Housing-Market-Algorithm

To facilitate quick deletion of houses from this list, we also have a data structure $prefPointer$. The variable $prefPointer$ is a two dimensional array such that $prefPointer[i][j]$ points to the node corresponding to house h_j in the doubly-linked list of agent a_i. If at any stage in the algorithm, we find out that the house h_j has been permanently allocated to some other agent than a_i, then we need to remove the house h_j from the preference list of a_i. Since $prefPointer[i][j]$ points to that node in the doubly linked list $prefList[i]$, we can delete the house in $O(1)$ time. Due to these deletions, we maintain the invariant that the head of $prefList[i]$ always corresponds to the top choice of the agent a_i. Note that if the input is given as the two dimensional array $pref$, where $pref[i][j]$ is the top j^{th} choice for the agent a_i, then it can be converted into $prefList$ and $prefPointer$ in $O(n)$ time with $O(n)$ processors.

We keep the array $fixed$ such that $fixed[i]$ indicates that the agent i has been assigned its final house. If an agent i is fixed, then it can never be forbidden in Fig. LLP-Housing-Market-Algorithm. Once all agents are fixed, we get that no agent is forbidden and the algorithm terminates.

At every iteration, we keep the array $inCycle[i]$ that indicates agents that are in Top Trading Cycle at that iteration. In Fig. LLP-Housing-Market-Algorithm, these agents correspond to $S(G)$ in the global state G. Algorithm LLP-TTC uses a *while* loop to fix some number of agents in every iteration. At least one agent is fixed in every iteration, and therefore there are at most n iterations of the while loop.

Each iteration has four steps. In the first step, we initialize *inCycle* to be false by default. In the second step (function *markRoots*) we use symmetry breaking via randomization and pointer jumping to mark one node called *root* in every cycle as belonging to a cycle. The reader is referred to [JáJ92] for symmetry breaking and pointer jumping. During the process of pointer jumping, we also construct a tree rooted at a vertex such that it consists of all the nodes in the cycle. In the third step (function *informTree*), we inform all the agents that are in some rooted tree that they are in a cycle. In the fourth step, we fix all the agents that are in cycles and remove their houses from $prefList$. This step corresponds to advancing G in Fig. LLP-Housing-Market-Algorithm.

Algorithm LLP-TTC: Parallel LLP Top Trading Cycle Algorithm

```
// By renumbering houses, ensure that initially agent $a_i$ is assigned house $h_i$
var
    $prefList$:array[1..$n$] of list initially $\forall i : prefList[i]$ has preferences for $a_i$;
    $prefPointer$:array[1..$n$, 1..$n$] of pointer to the node in $prefList$;
    $fixed$: array[1..$n$] of booean initially $\forall i : fixed[i] = false$;
    $inCycle$: array[1..$n$] of booean initially $\forall i : inCycle[i] = false$;
    $children$: array[1..$n$] of set of nodes that $a_i$ traversed initially $\{\}$;

while ($\exists i : \neg fixed[i]$)
    // Step 1: initialize $inCycle$
    forall $i : \neg fixed[i]$ in parallel do: $inCycle[i] := false$;
    // Step 2: Mark one node in every cycle as the root
    markRoots();
    // Step 3: inform all the agents in any rooted tree that they are in a cycle
    informTree();
    // Step 4: Now delete all the agents that are in cycle
    forall $i : \neg fixed[i] \wedge inCycle[i]$, $j : \neg[fixed[j] \wedge \neg inCycle[j]$ in parallel do
                delete the node $prefPointer[j][i]$ from the linked list $prefList[j]$;
    forall $i : \neg fixed[i] \wedge inCycle[i]$ in parallel do
        $fixed[i] := true$;
endwhile
return $prefList$; //$prefList[i]$ points to the house assigned to the agent $a_i$
```

The function *markRoots* uses variable *active* to denote agents that are active. Initially, all agents are active. The variable $succ[i]$ is used to point to the next active agent. Initially, $succ[i]$ points to the agent who has the top choice house of agent i. The variable $done[i]$ indicates whether a cycle has been discovered in the subgraph that agent i is pointing to. Once, a cycle has been discovered then any active agent knows that it cannot be part of any cycle and it becomes inactive.

The function *markRoots* uses a *while* loop at line 5 to run while there is any active node. Every active agent flips a coin at line 7. If its own coin is a head and its successor gets a tail, then this agent becomes inactive at line 9. It is clear that two consecutive agents can never become inactive in the same round because we require an agent to get "head" and its successor to get "tail" to become inactive. It is also clear that the number of active agents reduce by a constant fraction in every round of coin toss in expectation. Thus, the outer while loop at line 5 is executed expected $O(\log n)$ times.

If an agent is active, it traverses its *succ* pointer till it reaches the next *active* node. This is done using the *while* loop at line 11. This traversal has length of one or zero because there cannot be two consecutive inactive agents due to the rule of becoming active.

If agent i reaches itself as the next active node at line 15, it marks *inCycle* to be true. It also sets *done*$[i]$ to be true so that any active node j that points to i knows that a cycle has been found and that the node j can stop looking for the cycle. If the successor of the node is different, then we check if the successor is done. If the successor is done, then this node is not part of the cycle and can therefore make itself inactive and also mark itself as done. Since all agents execute the statements in *forall* in parallel, we get that the function *markRoots()* has parallel expected time complexity of $O(\log n)$. Also, for every cycle in the graph, there is exactly one node that sets its *inCycle* to be true.

The function *informTree* uses variable *rootSet* to initially include all the roots found in the function *markRoots*. Once all the nodes in any rooted tree have been informed, the root is deleted from the *rootSet*. To inform agents in the tree, we follow the usual method of broadcasting a value from the root to its children. To detect that all agents in the tree have been notified, we let any subtree that has finished informing its subtree to leave the tree by deleting itself from the children set of its parent. If the agent is a root, then it deletes itself from the *rootSet*. Once all roots have deleted themselves, the function terminates. Since the height of any tree is expected to be $O(\log n)$ and the number of children of any node is also $O(\log n)$, we get that the algorithm takes $O(\log^2 n)$ time.

Algorithm markRoots: Function markRoots for the Parallel LLP Top Trading Cycle Algorithm

```
1  function markRoots()
2      $succ$: array[1..$n$] of 1..$n$ initially $\forall i : succ[i] = prefList[i].head()$; //successor of $a_i$ which is active
3      $active$: array[1..$n$] of booean initially $\forall i : active[i] = true$;
4      $done$: array[1..$n$] of booean initially $\forall i : done[i] = false$;
5      while ($\exists i : active[i]$)
6          forall $i : \neg fixed[i] \wedge active[i]$ in parallel do
7              $coin[i]$ := "head" or "tail" // based on the flip of a coin
8              if ($coin[i]$ = "$head$") $\wedge$ ($coin[succ[i]]$ = "$tail$") then
9                  $active[i] := false$;
10             else // node $i$ is active
11                 while $\neg active[succ[i]]$ do
12                     $children[i] := children[i] \cup succ[i]$
13                     $succ[i] := succ[succ[i]]$
14                 endwhile
15                 if ($succ[i] = i$) // found a cycle
16                     $done[i] := true$
17                     $inCycle[i] := true$
18                     $active[i] := false$
19                 else if $done[succ[i]]$ then
20                     $active[i] := false$
21                     $done[i] := true$
22         endforall
23     endwhile
```

Algorithm informTree: Function informTree for the Parallel LLP Top Trading Cycle Algorithm

```
function informTree()
    $informed$: array[1..$n$] of booean initially $\forall i : informed[i] = false$;
    $parent$: array[1..$n$] of 1..$n$ initially $\forall i : parent[i] = i$;
    $rootSet$: set of 1..$n$ initially $\{i \mid inCycle[i]\}$
    while ($rootSet \neq \{\}$) do
        forall $i : \neg fixed[i]$ in parallel do
            if ($inCycle[i] \wedge \neg informed[i]$) then
                $informed[i] := true$
                for ($j \in children[i]$) do
                    $inCycle[j] := true$
                    $parent[j] := i$
                endfor
            if ($inCycle[i] \wedge informed[i] \wedge (children[i] = \{\})$) then
                if ($parent[i] = i$) then $rootSet.remove(i)$;
                else $children[parent[i]] := children[parent[i]] - \{i\}$
        endforall
    endwhile
```

We first show the correctness of the parallel algorithm LLP-TTC.

Theorem 15.7 *The algorithm LLP-TTC returns the core of the housing market problem.*

Proof: It is sufficient to show that the algorithm LLP-TTC finds all top trading cycles in each iteration. Consider any top trading cycle of size 1 at node i. The function markRoot can never mark node i in the cycle as inactive due to the requirement of the coin turning at node i as head and its successor, itself, as tail. Furthermore, since $succ[i]$ equals i, node i is marked as *inCycle*. Now, consider any top trading cycle of size $k > 1$. Since we require the successor of the node to have a different toss to turn inactive, all nodes cannot turn inactive. The active nodes keep the inactive nodes following it as its children. After every coin toss, the length of the cycle for active nodes is expected to shrink by a constant factor. Hence, in expected $O(\log n)$ coin tosses, the cycle reduces to size 1 and the former case applies.

Now consider any node i that is not in any top trading cycle. Since our graph is functional (every vertex has out-degree exactly one), node i leads to a cycle by following the *succ* edge. By previous discussion in $O(\log n)$ expected time, one of the nodes in that cycle, say j will set $inCycle[j]$ and $done[j]$ to be true. Since any path of active nodes reduces by a constant factor, in $O(\log n)$ expected time node i will point to a node that is *done* and will also mark itself as *done*.

The function $informTree$ simply sets the variable *inCycle* of all nodes in the cycle to be true. Finally, step 4 removes all houses and agents that are in cycle and thus implement the top trading cycle mechanism.

■

We now analyze the time and work complexity of LLP-TTC.

Theorem 15.8 *LLP-TTC takes expected $O(n \log^2 n)$ time and expected $O(n^2 \log n)$ work.*

Proof: Since every functional graph has at least one cycle, there exists at least one new node that finds itself in a cycle in every iteration of the while loop. Hence, there are at most n iterations of the while loop. In each iteration, Step 1 takes $O(1)$ time and $O(n)$ work. Step 2 takes expected $O(\log n)$ time and expected $O(n \log n)$ work. Step 3 takes $O(\log^2 n)$ time and $O(n)$ work. Let α_k be the number of agents that are fixed in the k^{th} iteration of the while loop. Step 4 takes $O(\alpha_k)$ time and $O(n\alpha_k)$ work. Adding up over all iterations, we get the desired time and work complexity.

■

15.5 Problems

1. Give an NC algorithm to determine if the given matching is the core of the housing market.

15.6 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 16

The Assignment Problem

16.1 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.

16.2 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.

$$\begin{aligned}
\text{maximize} \quad & \sum_{(b,i)\in[n]\times[n]} v_{b,i}x_{b,i} \\
\text{subject to} \quad & \sum_b x_{b,i} = 1 \quad \forall i \in [n], \\
& \sum_i x_{b,i} = 1 \quad \forall b \in [n], \\
& x_{b,i} \geq 0 \quad \forall i \in [n], b \in [n]
\end{aligned} \tag{16.1}$$

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:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i\in[n]} p_i + q_i \\
\text{subject to} \quad & p_i + q_b \geq v_{b,i} \quad \forall i \in [n], b \in [n], \\
& p_i \geq 0 \quad \forall i \in [n], \\
& q_b \geq 0 \quad \forall b \in [n]
\end{aligned} \tag{16.2}$$

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.

16.3 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. 16.1). There are n processes corresponding to n items in the computation such that each process has C events. In Fig. 16.1, 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

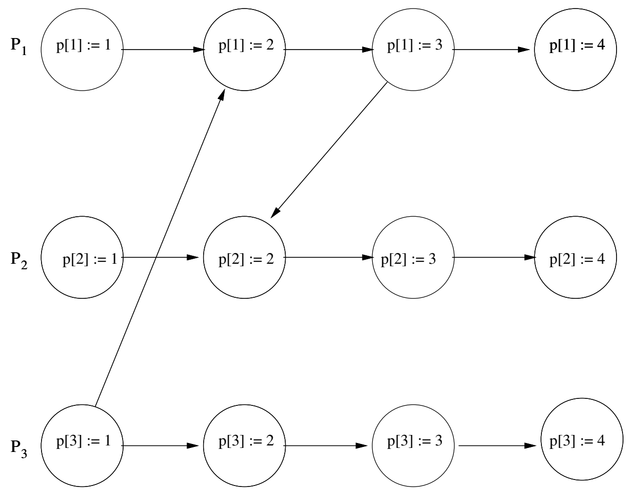


Figure 16.1: The computation graph for a market with three items and three bidders. The valuation of and price for any item is a number between 0 and 4.

Lemma 16.1 *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. 16.2, 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.

16.4 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 $clearing(G) \wedge B(G)$. From Lemma 2.6, it is sufficient to give an algorithm for $clearing(G)$.

```
$P_j$: 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 16.2: **Algorithm ConstrainedMarketClearingPrice** to find the minimum cost assignment vector

Algorithm LLP-Assignment: Finding the minimum clearing price vector

```
var
    $G$: real initially $\forall i : G[i] = 0$;
while(true)
    $E := \{(i,b) \mid \forall j : (v_{b,i} - G[i]) \geq (v_{b,j} - G[j])\}$;
    if there exists a perfect matching $M$ in $E$
        then return $M$
    else
        $J$ := minimal OverDemanded set in the bipartite graph with edges $E(G)$;
        forall $j \in J$ in parallel do
            $\delta_j = \min\{(v_{b,j} - G[j]) - \max_{i \neq j}(v_{b,i} - G[i]) \mid (j,b) \in E(G)\}$
            $G[j] := G[j] + \delta_j$;
end
```

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. 16.2. In conjunction with Lemma 2.6, 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. 16.3). 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.

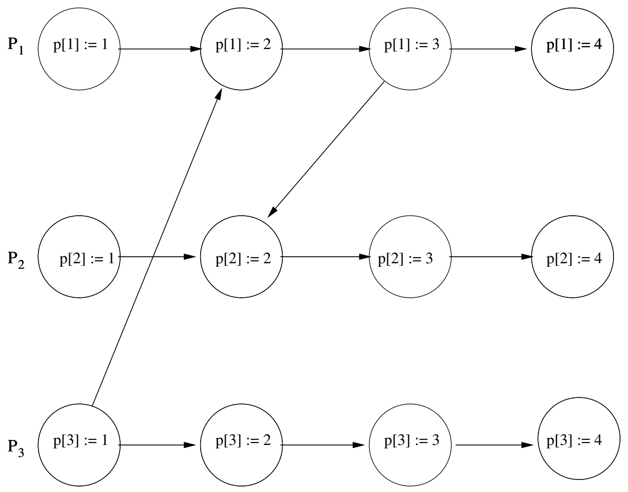


Figure 16.3: The computation graph for a market with three items and three bidders. The valuation of and price for any item is a number between 0 and 4. The computation graph models the constraint $B \equiv (p[2] \geq 2 \Rightarrow p[1] \geq 3) \wedge (p[1] \geq 2 \Rightarrow p[3] \geq 1)$

For example, suppose that the valuation for three items in Fig. 16.3 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.

16.5 Problems

1. Show that the set of market clearing price vectors are also closed under the join operation.

16.6 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 17

Bipartite Matching

17.1 Introduction

This chapter is organized as follows. Section 17.2 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.

We note here that there exists RNC algorithms to check if there exists a perfect matching in a bipartite graph. These algorithms are not work-efficient and would not be discussed in the chapter.

17.2 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 17.1 (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

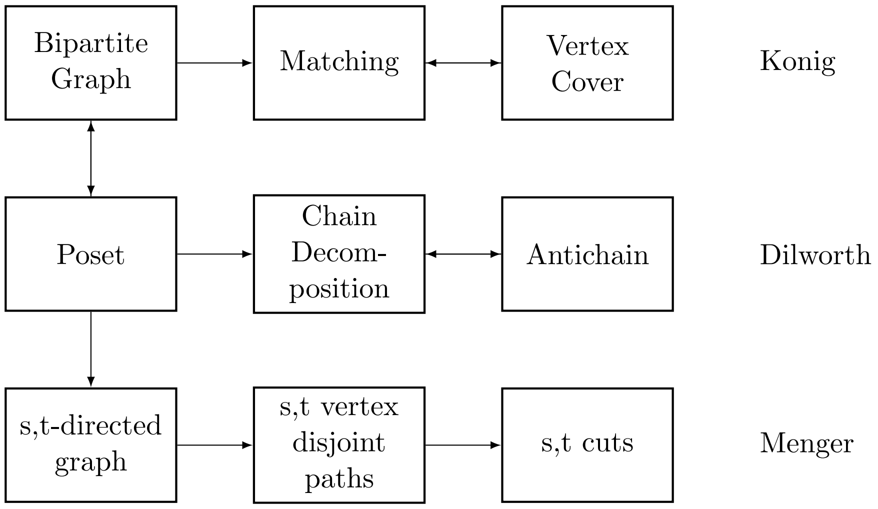


Figure 17.1: Various Structures and Transformations between them

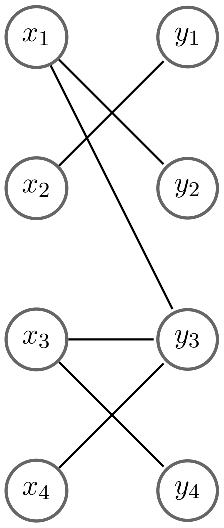


Figure 17.2: A Bipartite Graph

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 $M$;
```

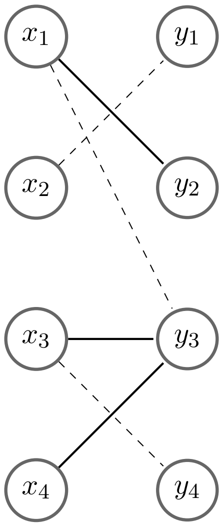


Figure 17.3: A Matching M shown with dashed edges in the Bipartite Graph

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. 17.3, 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

```
input: $(L, R, E)$: bipartite graph
var: $G$: array[1..$n$] of int ;
init: $\forall j : G[j] := 0$;
$S$: array[1..$n$] of 0..$n$;
init: $\forall j : S[j] := 0$;
for $j := 1$ to $n$ do
    if there exists an alternating path from $v_j$ to any vertex $k$ in $R$ with $(S[k] = 0)$
        Assign $S[]$ in the alternating path;
        $G[j] := 1$;
endfor
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.

17.3 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. Modify Algorithm BipartiteMatching to output the vertex cover in addition to bipartite matching.

Chapter 18

Horn and 2-SAT Satisfiability

18.1 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 $\{x_1, x_2\}$.

18.2 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.

18.3 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 18.1 *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 18.1 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

```
var $G$: vector of boolean initially $\forall i : G[i] = false$;
$P_j$: code for thread $j$
forbidden: $\exists h$ : all antecedents of $B_h$ are true, they imply $x_j$, and $x_j$ is false in $G$
    advance: $G[j] := true$;
forbidden: $\exists h$ : a negative clause $B_h$ is false, and $x_j$ is a part of $B_h$
    return null; // "no satisfying assignment exists"
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 18.1 from its lattice-linearity.

Corollary 18.2 *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.

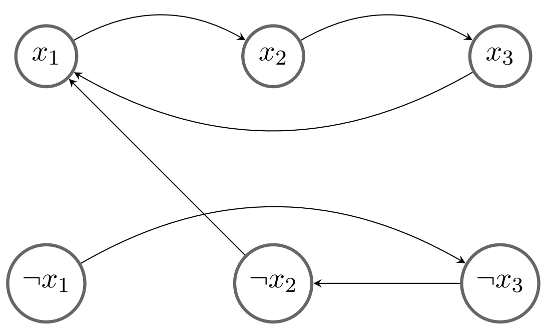


Figure 18.1: Implication Graph of the predicate $B \equiv (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$

18.4 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 $(1-y_{i_1})+(1-y_{i_2})+\dots(1-y_{i_{k-1}})+y_{i_k} \geq 1$ which is equivalent to $y_{i_k} \geq y_{i_1}+y_{i_2}+\dots+y_{i_{k-1}}-(k-1)$. The right hand side is a monotone function of y; hence the predicate is lattice-linear. A pure negative clause $(\neg x_{i_1} \vee \neg x_{i_2} \dots \vee \neg x_{i_k})$ gets translated to $(1 - y_{i_1}) + (1 - y_{i_2}) \dots + (1 - y_{i_k}) \geq 1$. This predicate is equivalent to $k - 1 \geq y_{i_1} + y_{i_2} + \dots + y_{i_k}$ which once it becomes false it stays false. Hence, predicates for negative clauses are also lattice-linear.

18.5 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. 18.1.

We leave the following as an exercise.

Exercise 18.1 *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

```
vector function getFeasible($B$: predicate)
var $G$: array[$1 \dots n$] of boolean initially $\forall i : G[i] = false$;
    $fixed$: array[$1 \dots n$] of boolean initially $\forall i : fixed[i] = false$;
    $I(B)$ := implication graph of $B$;
    if there exists a path from $\neg x_j$ to $x_j$ and from $x_j$ to $\neg x_j$ in $I(B)$ then
        return null; //"no satisfying solution"
    while ($\exists j : \neg fixed[j]$)
        if there exists a path from $\neg x_j$ to $x_j$ then $G[j] := true$;
        $fixed[j] := true$;
        $unitPropagation(B, j)$;
    endwhile;
    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.

18.6 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 18.2 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 19

Stable Marriage Problem with Ties

19.1 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 3.

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.

19.2 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 19.1 *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 19.2 *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     $\forall z \in Y(j) : \exists i \neq j : \exists k \leq G[i] : (z \in mpref[i][k]) \wedge (wrank[z][i] \leq wrank[z][j]))$
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 19.3 *The set of superstable marriages, $L_{superstable}$, is a sublattice of the lattice L.*

Proof: From Lemma 19.2, 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.

19.3 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

$((mrank[m][w] \leq mrank[m][M(m)]) \wedge (wrank[w][m] < wrank[w][M(w)]))$
$\vee((mrank[m][w] < mrank[m][M(m)]) \wedge (wrank[w][m] \leq wrank[w][M(w)]))$.

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.

Theorem 19.4 *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.

Algorithm LLP-ManOptimalStronglyStableMarriage: A Parallel Algorithm for Man-Optimal Strongly Stable Matching

```
P_j: Code for thread j
input: $mpref[i,k]$: set of int for all $i,k$; $wrank[k][i]$: int for all $k,i$;
init: $G[j] := 1$;
always: $Y(j) = mpref[j][G[j]]$;
forbidden($j$):
    $j$ is a member of the critical subset of men in the graph $Y(G)$
    advance: $G[j] := G[j] + 1$;
```

19.4 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.

19.5 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 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 $G[i]$.

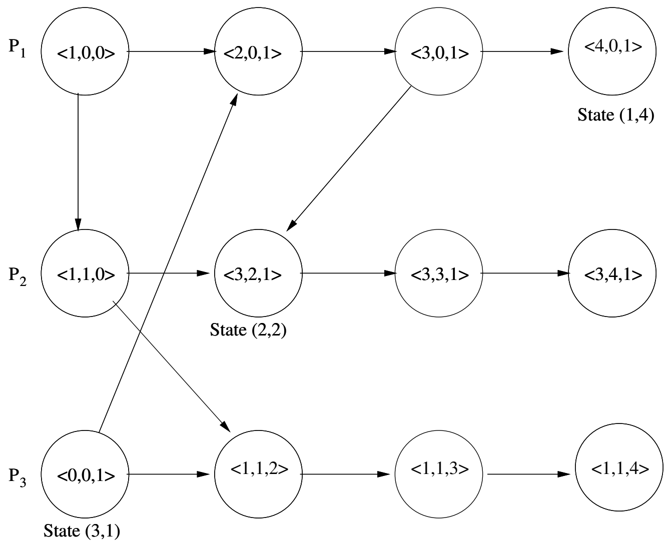


Figure 20.1: State-Based Model of a Distributed Computation

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
    $forbidden(G, j, B) \equiv \exists i : G[j] \rightarrow G[i]$
while $\exists j : forbidden(G, j, B)$ 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: $states : array[1 \dots n][1 \dots m]$ of vectorClock // Sequence of local states at each process
Output: Consistent Global State as array $cut[1 \dots n]$

Step 1: Create $F$: set of states rejected in the first round
 var $F : array[1 \dots n]$ of $0 \dots 1$ initially $0$;
 for all $(i \in 1 \dots n, j \in 1 \dots n)$ in parallel do
         if $((i, 1) \to (j, 1))$ then
            $F[i] := 1$;

Step 2: Create $R$: State Rejection Graph // Represented as an Adjacency Matrix
 var $R : [(1 \dots n, 1 \dots m), (1 \dots n, 1 \dots m)]$ of $0 \dots 1$;
 for all $(i \in 1 \dots n, j \in 1 \dots m)$ in parallel do
      $R[(i, j), (i, j)] := 1$;
 for all $(i \in 1 \dots n, j \in 1 \dots m - 1, i' \in 1 \dots n, j' \in 1 \dots m)$
         such that $i \neq i'$ in parallel do
         if $((i', j') \to (i, j + 1))$ then
            $R[(i, j), (i', j')] := 1$;
         else
            $R[(i, j), (i', j')] := 0$;

Step 3: Create $RT$: transitive closure of $R$
 var $RT : array[(1 \dots n, 1 \dots m), (1 \dots n, 1 \dots m)]$ of $0 \dots 1$;
 $RT := TransitiveClosure(R)$;

Step 4: Create $valid$: replace invalid states by 0
 var $valid : array[[1 \dots n][1 \dots m]$ of $0 \dots 1$;
 for all $(i \in 1 \dots n, j \in 1 \dots m)$ in parallel do
      $valid[i][j] := 1$;
 for all $(i \in 1 \dots n, i' \in 1 \dots n, j' \in 1 \dots m)$ in parallel do
      if $(F[i] = 1) \wedge (RT[(i, 1), (i', j')] = 1)$ then
         $valid[i'][j'] := 0$;


Step 5: Create $cut$: First Consistent Global State
 var $cut : array[1 \dots n]$ of $0 \dots m$ initially $0$;
 for all $(i \in 1 \dots n, j \in 1 \dots m)$ in parallel do
         if $(valid[i][j] \neq 0)$ then
            if $(j = 1) \vee ((j > 1) \wedge (valid[i][j - 1] = 0)$ then
               $cut[i] := j$;
 for all $(i \in 1 \dots n)$ 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.

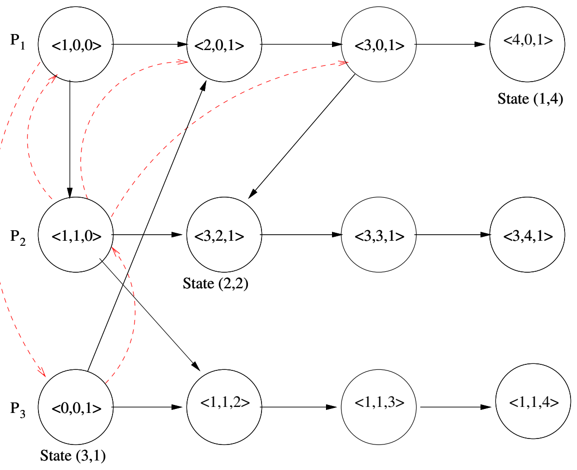


Figure 20.4: State Rejection Graph of a computation shown in dashed arrows

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 Level k

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 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].

Chapter 21

Complexity Of Predicate Detection

21.1 Introduction

In this chapter, 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.

21.2 Predicate Detection Problem

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 21.1 *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 21.1. 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.

■

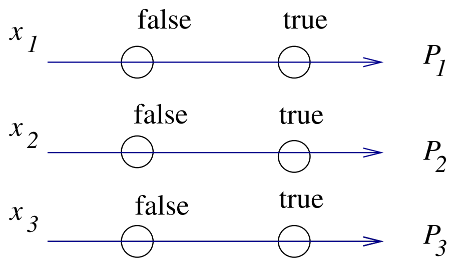


Figure 21.1: Transformation from SAT to global predicate detection

The above result shows that detection of a general global predicate is intractable even for simple distributed computations.

21.3 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.

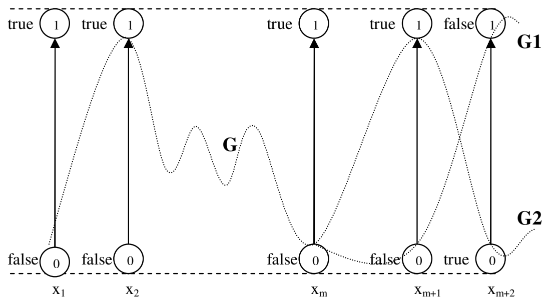


Figure 21.2: Transformation for Theorems 21.2 and 21.3.

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 21.2 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 21.2 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 21.2.

$$G1 = (G, 0, 1)$$

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 21.3 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 21.2 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.

21.4 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 21.4 (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 21.1, 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.

21.5 Problems

21.6 Bibliographic Remarks

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 22

Enumeration Algorithms

22.1 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.

22.2 Birkhoff's Theorem

We first define *join-irreducible* elements as follows.

Definition 22.1 (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 22.1(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 $(E, \to)$ by $C(E)$ ($\to$ 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 22.2 (Birkhoff's Representation Theorem) *Let L be a finite distributive lattice. Then the map $f : L \to C(J(L))$ 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 $g : P \to J(C(P))$ defined by

$$g(a) = \{x \in P \mid x \leq a\}$$

is an isomorphism of P onto $J(C(P))$.

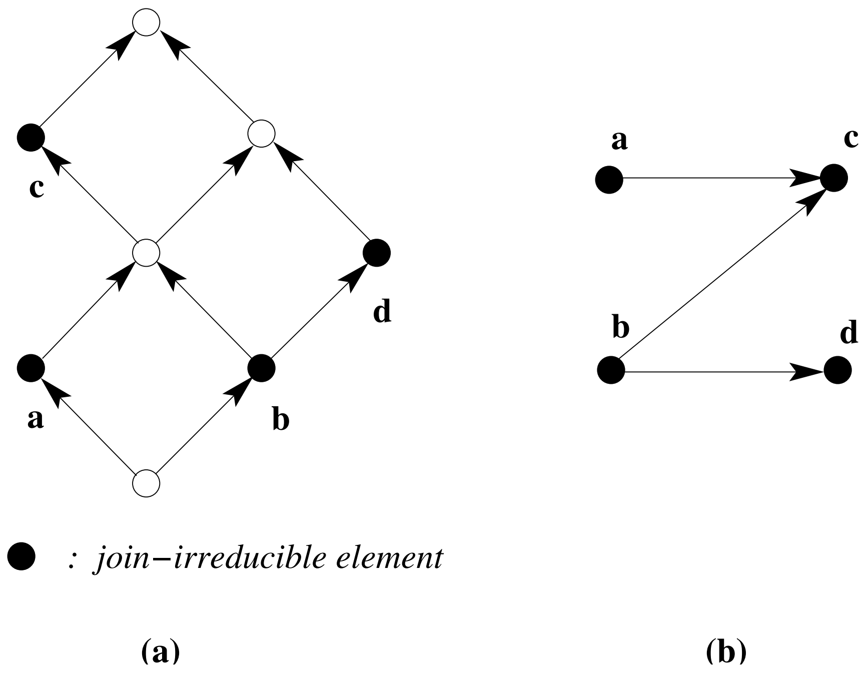


Figure 22.1: (a) An example of a distributive lattice (b) its partial order representation.

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 22.1(b) gives the poset corresponding to the lattice in Figure 22.1(a).

From the above discussion it is clear that given any finite distributed computation, the structure *finite distributive lattice* completely characterizes its execution graph.

22.3 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 $\bot$ and $\top$ such that $\bot$ is the smallest vertex and $\top$ is the largest vertex (i.e., there is a path from $\bot$ to every vertex and a path from every vertex to $\top$). It is easy to see that any nontrivial ideal will contain $\bot$ 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. 22.2(a) is shown in Fig. 22.2(c).

The slice of a directed graph P with respect to a predicate B (denoted by *slice*(P, B)) is a graph derived from

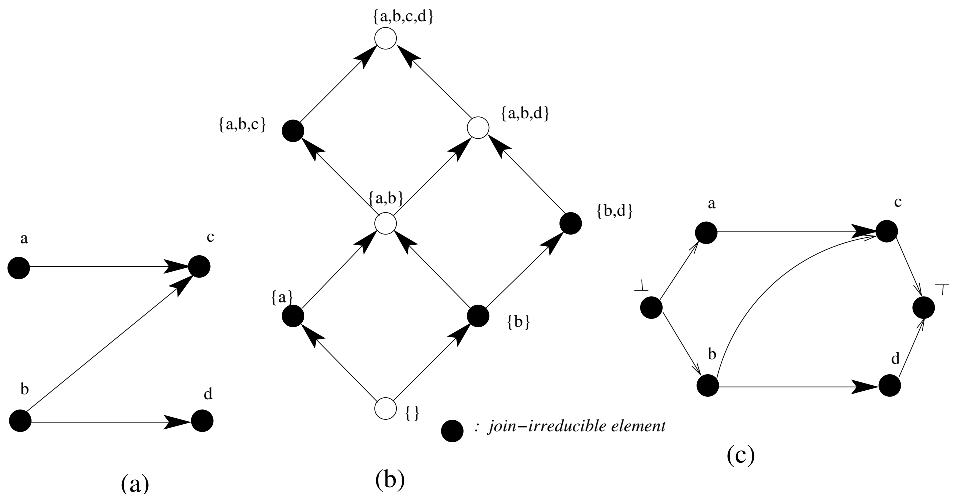


Figure 22.2: (a) P: A partial order (b) the lattice of ideals. (c) the directed graph P'

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 22.3 (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'.

22.4 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 22.4 *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 22.4, 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. 22.3. 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 $f \in J(B, e)$ do;
(7)              add edge $(f, e)$ to $R$;
(8)      return $R$;
(9)  end;
```

Figure 22.3: 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.

22.5 Problems

22.1. Show that $x \in L$ is join-irreducible iff
(i) $x \neq 0$
(ii) $\forall a, b : (a < x) \wedge (b < x) \Rightarrow ((a \sqcup b) < x)$

22.2. 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 $M(L)$.

22.6 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].

Chapter 23

Appendix: List of Lattice Linear Programs in Java

```
import java.util.Arrays;
public class JobScheduling extends LLP {
   final int[] time;
   final int[][] prerequisites;
   int[] G;
   public JobScheduling(int[] time, int[][] prerequisites) {
      super(time.length);
      this.time = time;
      this.prerequisites = prerequisites;
      G = Arrays.copyOf(time, time.length);
   }
   public boolean forbidden(int j) {
      for (int i : prerequisites[j])
         if (G[j] < G[i] + time[j]) return true;
      return false;
   }
   public void advance(int j) {
      int newStart = -1;
      for (int i : prerequisites[j])
         newStart = Math.max(newStart, G[i] + time[j]);
      G[j] = newStart;
   }
   public int[] getSolution() { return G; }
}
```

Figure 23.1: Job Scheduling Program

```java
import java.util.Arrays;
public class ParallelReduce extends LLP {
   final int n;
   final int[] A;
   int[] G;
   public ParallelReduce(int[] A) {
      super(A.length);
      this.n = A.length;
      this.A = A;
      G = new int[n];
      Arrays.fill(G, Integer.MIN_VALUE);
      G[0] = 0;
   }
   public boolean forbidden(int j) {
      return ((1 <= j && j < n/2 && G[j] < G[2*j] + G[2*j+1]) ||
            (n/2 <= j && j < n && G[j] < A[2*j-n] + A[2*j-n+1]));
   }
   public void advance(int j) {
      if (1 <= j && j < n/2) G[j] = G[2*j] + G[2*j+1];
      if (n/2 <= j && j < n) G[j] = A[2*j-n] + A[2*j-n+1];
   }
   public int[] getSolution() {
      int[] reduce = new int[n - 1];
      for (int i = 1; i < n; i++)
         reduce[i-1] = G[i];
      return reduce;
   }
}
```

Figure 23.2: Parallel Reduce Program

```
import java.util.Arrays;
public class ParallelPrefix extends LLP {
   final int n;
   final int[] A;
   final int[] S;
   int[] G;
   public ParallelPrefix(int[] A, int[] S) {
      super(A.length * 2);
      this.n = A.length;
      this.A = A;
      this.S = S;
      G = new int[2 * n];
      Arrays.fill(G, Integer.MIN_VALUE);
      G[0] = -1;
   }
   public boolean forbidden(int j) {
      return ((j == 1 && G[j] < 0) ||
            (j % 2 == 0 && G[j] < G[j/2]) ||
            (j % 2 == 1 && j > 1 && j < n && G[j] < S[j-2] + G[j/2]) ||
            (j % 2 == 1 && j > 1 && j > n && G[j] < A[j-n-1] + G[j/2]));
   }
   public void advance(int j) {
      if (j == 1) G[j] = 0;
      else if (j % 2 == 0) G[j] = G[j/2];
      else if (j % 2 == 1 && j < n) G[j] = S[j-2] + G[j/2];
      else if (j % 2 == 1 && j > n) G[j] = A[j-n-1] + G[j/2];
   }
   public int[] getSolution() {
      int[] prefix = new int[n];
      for (int i = n; i < 2*n; i++)
         prefix[i-n] = G[i];
      return prefix;
   }
}
```

Figure 23.3: Parallel Prefix Program

```java
public class StableMarriage extends LLP {
   final int m, w;
   final int[][] mprefs;
   final int[][] wprefs;
   int[] G;
   public StableMarriage(int[][] mprefs, int[][] wprefs) {
      super(mprefs.length);
      this.m = mprefs.length;
      this.w = wprefs.length;
      this.mprefs = mprefs;
      this.wprefs = wprefs;
      this.G = new int[m];
   }
   public boolean forbidden(int j) {
      int z = mprefs[j][G[j]];
      int[] wpref = wprefs[z];
      for (int i = 0; i < m; i++) {
         if (i==j) continue;//only for different men
         if (z != mprefs[i][G[i]]) continue;
         if (wpref[i] < wpref[j]) return true;
      }
      return false;
   }
   public void advance(int j) {
      G[j]++;
   }
   public int[] getSolution() {//Return the partner for each man as a list
      int[] assignment = new int[m];
      for (int i = 0; i < m; i++)
         assignment[i] = mprefs[i][G[i]];
      return assignment;
   }
}
```

Figure 23.4: Stable Marriage Program

```
public class ListRank extends LLP {
   public class Node {
      int dist;
      int next;
      public Node(int dist, int next) {
         this.dist = dist;
         this.next = next;
      }
   }

   Node[] G;
   int r;
   int n;
   public ListRank(int[] parent) {
      super(parent.length);
      this.n = parent.length;
      this.G = new Node[n];
      for (int i = 0; i < n; i++)
         if (parent[i] == -1) {
            this.r = i;
            G[i] = new Node(0, i);
         } else G[i] = new Node(1, parent[i]);
   }
   public boolean forbidden(int j) {
      return G[j].next != this.r;
   }
   public void advance(int j) {
      while (true) {
         int dist = G[G[j].next].dist;
         int next = G[G[j].next].next;
         // Ensure the values haven't changed in between reads.
         if (G[G[j].next].dist == dist) {
            G[j].dist += dist;
            G[j].next = next;
            break;
         }
      }
   }
   public int[] getSolution() {//Return the distance of every node to the root
      int[] distances = new int[n];
      for (int i = 0; i < n; i++)
         distances[i] = G[i].dist;
      return distances;
   }
}
```

Figure 23.5: List Rank Program

```
public class Traversal extends LLP {
   int n; //number of vertices
   final int[][] edge;
   int[] visited;
   int[] G;
   public Traversal(int n, int[][] edge, int source) {
      super(n);
      this.edge = edge;
      this.G = new int[n];
      visited = new int[n];
      visited[source] = 1;
   }
   public boolean forbidden(int j) {
      if ((visited[j] == 1) && (G[j] == 0)) return true;
      else return false;
   }
   public void advance(int j) {
      G[j] = 1;
      for (int k: edge[j]) visited[k] = 1;
   }
   public int[] getSolution() { return G; }
}
```

Figure 23.6: Graph Traversal Program

```
public class TraversalBFS extends LLP {
   final int[][] pre;
   int[] G;
   public TraversalBFS(int n, int[][] pre, int source) {
      super(n);
      this.pre = pre;
      G = new int[n];
      for (int i=0; i<n; i++)
         G[i] = n+1; //equivalent to infinity
      G[source] = 0;
   }
   public boolean ensure(int j) {
      boolean changed = false;
      for (int i: pre[j])
         if (G[i] < n)
            if (G[j] > G[i]+1) {
               changed = true;
               G[j] = G[i]+1;
            }
      return changed;
   }
   public int[] getSolution() { return G; }
}
```

Figure 23.7: Graph BFS Traversal Program

```
public class Layering extends LLP {
   final int[][] prerequisites;
   int[] G;
   boolean[] fixed;
   public Layering(boolean[] fixed, int[][] prerequisites) {
      super(fixed.length);
      this.fixed = fixed;
      this.prerequisites = prerequisites;
      this.G = new int[fixed.length];
      fixed[0] = true;
      G[0] = 0;
   }
   public boolean forbidden(int j) {
      if (fixed[j]) return false;
      for (int i : prerequisites[j])
         if (!fixed[i]) return false;
      return true;
   }
   public void advance(int j) {
      int newStart = 0;
      for (int i : prerequisites[j])
         newStart = Math.max(newStart, G[i]+1);
      G[j] = newStart;
      fixed[j] = true;
   }
   public int[] getSolution() { return G; }
}
```

Figure 23.8: Graph Layering Program

```java
public class SlowComponent extends LLP {
   final int[][] adj;
   int[] G;
   int num;
   boolean debug = true;
   public SlowComponent(int num, int[][] adj) {
      super(num);
      this.adj = adj;
      this.G = new int[num];
      for (int i=0; i<num; i++) G[i] = i;
   }
   public boolean ensure(int j) {
      boolean changed = false;
      for (int i: adj[j])
         if (G[j] < G[i]) { G[j] = G[i]; changed = true; }
      return changed;
   }
   public int[] getSolution() { return G; }
}
```

Figure 23.9: Slow Component Program

```java
public class FastComponent extends LLP {
   final int[][] adj;
   int[] G;
   int num;
   boolean debug = true;
   public FastComponent(int num, int[][] adj) {
      super(num);
      this.adj = adj;
      this.G = new int[num];
      for (int i=0; i<num; i++) G[i] = i;
   }
   public boolean ensure(int j) {
      boolean changed = false;
      if (G[j] != G[G[j]]) { G[j] = G[G[j]]; changed = true; }
      else
         for (int i: adj[j])
            if (G[j] < G[i]) { G[j] = G[i]; changed = true; }
      return changed;
   }
   public int[] getSolution() { return G; }
}
```

Figure 23.10: Fast Component Program

```
import java.util.*;
class Node implements Comparable<Node>{
   int index;
   int distance;
   public Node(int index, int distance) {
      this.index = index;
      this.distance = distance;
   }
   public int compareTo(Node other) { return Integer.compare(distance,
       other.distance); }
}
public class Dijkstra extends LLP {
   final List<List<Node>> adjacency;
   final int[][] w;
   int[] G;
   int num;
   boolean debug = false;
   boolean[] fixed;
   PriorityQueue<Node> H;
   public Dijkstra(int num, List<List<Node>> adjacency, int[][] w) {
      super(num);
      this.adjacency = adjacency;
      this.w = w;
      G = new int[num];
      fixed = new boolean[num];
      Arrays.fill(G, Integer.MAX_VALUE);
      Arrays.fill(fixed, false);
      int start = 0;
      G[start] = 0;
      H = new PriorityQueue<Node>();
      H.add(new Node(start, 0));
   }
   public boolean forbidden(int j) {
      if (H.isEmpty()) return false;
      Node temp = H.peek();
      if (temp.index != j) return false;
      else return true;
   }
   public void advance(int j) {
      Node t = H.poll();
      if (fixed[t.index]) return;
      fixed[t.index] = true;
      for (Node k : adjacency.get(t.index))
         if (G[t.index] + k.distance < G[k.index]) {
            G[k.index] = G[t.index] + k.distance;
            H.add(new Node(k.index, G[k.index]));
         }
   }
   public int[] getSolution() { return G; }
}
```

Figure 23.11: Dijkstra's Shortest Path Program

```java
public class BellmanFord extends LLP {
   final int[][] pre;
   final int[][] w;
   int[] G;
   int num;
   public BellmanFord(int num, int[][] pre, int[][] w) {
      super(num); this.pre = pre; this.w = w;
      G = new int[num];
      for (int i=0; i<num; i++)
         G[i] = 1000; // Integer.MAX_VALUE;
      G[0] = 0;
   }
   public boolean ensure(int j) {
      boolean changed = false;
      for (int i: pre[j])
         if (G[j] > G[i] + w[i][j]) {
            G[j] = G[i] + w[i][j];
            changed = true;
         }
      return changed;
   }
   public int[] getSolution() { return G; }
}
```

Figure 23.12: Bellman-Ford Shortest Path Program

```java
public class Johnson extends LLP {
   final int[][] pre;
   final int[][] w;
   int[] G;
   int num;
   public Johnson(int num, int[][] pre, int[][] w) {
      super(num); this.pre = pre; this.w = w;
      G = new int[num];
      for (int i=0; i<num; i++) G[i] = 0;
   }
   public boolean ensure(int j) {
      boolean changed = false;
      for (int i: pre[j])
         if (G[j] < G[i] - w[i][j]) {
            G[j] = G[i] - w[i][j];
            changed = true;
         }
      return changed;
   }
   public int[] getSolution() { return G; }
}
```

Figure 23.13: Johnson Shortest Path Program