Parallel Algorithms for the k Shortest Paths and Related Problems

Eric Ruppert

A thesis submitted in conformity with the requirements
for the degree of Master of Science to the
Graduate Department of Computer Science
University of Toronto

Parallel Algorithms for the k Shortest Paths and Related Problems

Eric Ruppert

A Master of Science thesis submitted to the

Graduate Department of Computer Science,

University of Toronto

1996

Abstract

A parallel algorithm is developed to find the k shortest paths between pairs of vertices in an edge-weighted directed graph. The concurrent-read exclusive-write PRAM is used as the model of computation. The algorithm computes an implicit representation of the k shortest paths to a given destination vertex from each vertex of a graph with n vertices and m edges, using $O(m + nk\log^2 k)$ work and $O(\log^3 k \log^* k + \log n(\log\log k + \log^* n))$ time, assuming that a shortest path tree rooted at the destination is precomputed.

Parallel algorithms are also described for a weighted version of the problem of selecting an element of given rank from an unsorted array and for the selection of the kth smallest element in a matrix with sorted columns.

The k shortest paths algorithm is applied to obtain a parallel implementation of a dynamic programming algorithm for the list Viterbi decoding problem, where one must find the most probable state sequences of a Markov process, given noisy observations of the state transitions. Algorithms are developed for the problem of computing quickest paths for the transmission of data through a graph, where the time to send data across an edge depends on the capacity of the edge and on its latency.

Acknowledgements

My supervisor, Faith Fich, has placed her extensive knowledge of computer science at my disposal during our discussions and has given me invaluable feedback at every stage in the preparation of this thesis. It has been extremely pleasant to work with her, and I am grateful for her guidance.

I offer sincere thanks to Derek Corneil, who carefully read a draft of this thesis and suggested improvements and corrections.

Financial support was provided by the Natural Sciences and Engineering Research Council of Canada.

I am deeply thankful for my family's continuing support and encouragement.

Contents

1 Introduction **1**

1.1 Graph Terminology . . . 3

1.2 The PRAM Model . . . 4

1.3 Basic PRAM Algorithms . . . 5

1.3.1 Prefix Sums and Processor Allocation . . . 5

1.3.2 Merging . . . 6

1.3.3 Sorting . . . 7

1.3.4 Selection . . . 7

1.3.5 Tree Contraction . . . 8

2 Some Parallel Selection Algorithms **10**

2.1 The Weighted Selection Problem . . . 10

2.1.1 Performance Analysis . . . 11

2.2 Selection in a Matrix With Sorted Columns . . . 12

2.3 A Recursive Algorithm for Selection in a Matrix With Sorted Columns . . . 13

2.3.1 Correctness of the Algorithm . . . 14

2.3.2 Performance Analysis . . . 16

2.3.3 Remarks . . . 19

2.4 A Parallel Elimination Algorithm for Selection in a Matrix With Sorted Columns . . . 19

2.4.1 Representation of the Matrix . . . 19

2.4.2 Reducing the Number of Active Elements to $O(k)$ . . . 20

2.4.3 Reducing the Number of Active Elements to $O(n)$ . . . 23

2.5 Combining the Two Selection Algorithms . . . 25

3 The k Shortest Paths Problem **27**

3.1 Background . . . 27

3.2 A Sequential Algorithm to Compute the k Shortest Paths . . . 28

3.3 A Parallel Algorithm to Compute the k Shortest Paths . . . 33

3.3.1 The Data Structure Used by the Algorithm 34
3.3.2 Computing A_v^0 35
3.3.3 Computing A_v^i for $i > 0$ 35
3.3.4 Extracting Information From the Implicit Representation 38

4 Applications **41**

4.1 Quickest Paths 41
4.1.1 Sequential Algorithms 42
4.1.2 Finding the k Quickest Paths in Parallel 45
4.2 Adapting Dynamic Programming Algorithms to Find the k Best Solutions in Parallel 49
4.2.1 Dynamic Programming and Shortest Paths 49
4.2.2 The Viterbi Decoding Problem 50
4.2.3 The Viterbi Algorithm 51
4.2.4 A Parallel Algorithm for the List Viterbi Problem 53

5 Open Problems **54**

Bibliography **55**

Index **59**

Chapter 1

Introduction

The problem of finding shortest paths in an edge-weighted graph is an important and well-studied problem in computer science. The more general problem of computing the k shortest paths between nodes of a graph also has a long history, and many applications to a diverse range of problems. A parallel algorithm will be developed for the k shortest paths problem using the PRAM model of parallel compuation.

Many algorithms which compute the k best solutions to an optimization problem first compute an optimal solution using a standard algorithm. A number of candidates for the second best solution are then generated by modifying the optimal solution, and the algorithm outputs the best candidate as the second best solution to the problem. In general, the kth best solution is chosen from a set of candidates, each one a modification of one of the best $k-1$ solutions. It seems that this approach cannot be used to obtain parallel algorithms with running times that are polylogarithmic in k, since the best $k-1$ solutions must be known before the algorithm can compute the kth best solution. A different technique is used here to produce a parallel algorithm for the k shortest paths problem with a running time that is polylogarithmic in k, and in the size of the problem instance, without performing too much more work than the best sequential algorithm for the problem.

It is useful to compute the k best solutions to an optimization problem when some of the constraints on admissible solutions are difficult to specify. Some of the constraints may be somewhat subjective and therefore difficult to formalize. There may be some constraints which make an otherwise tractable problem into a very complex problem. In both cases, one could first enumerate a number of the best solutions to the simpler problem obtained by omitting the difficult constraints, and then choose from among them a solution that satisfies the additional constraints. For example, in the area of natural language understanding, it is possible to specify formal rules for the syntax of a legal sentence, and to use these rules to parse a given utterance efficiently and obtain a formal representation of the input. (This representation may not match the input perfectly, because of noisy input, idiosyncratic speech, ambiguity or other real-world conditions). The problem would be far more complicated if constraints on the meaning of the sentence were also introduced at the parsing stage to help choose the best representation. Instead, one can enumerate the parses that best represent the utterance syntactically until one is found that can be given a reasonable semantic interpretation. An algorithm for the k shortest paths problem has been applied in this way to solve the list Viterbi decoding problem and improve the performance of a speech recognition system in some simple settings [41]. The list Viterbi decoding problem, and a parallel algorithm that solves it will be described in Section 4.2.

There is another important motivation for computing the k best solutions to a problem. When solving an optimization problem, it is often helpful to know whether the solution (or parts of the solution) are likely to change radically if the problem instance changes slightly. This is particularly important when there is reason to believe that the input parameters will not be known precisely. In some cases, there may be a very large number of independent problem parameters, so that it is not feasible to solve the optimization problem for a representative sample of small perturbations of the given problem instance to determine whether they have similar optimal solutions. Instead, a collection of the best solutions to the optimization problem may be examined. If one of the best solutions is very different from the optimal solution but has a cost that is only slightly sub-optimal, it is likely that minor modifications to the problem could cause the sub-optimal solution to become optimal. In fact, knowledge of such a sub-optimal solution may be useful in determining the input parameters to which the solution is sensitive.

In the remainder of this chapter, some background information is presented. Graph theory terminology is reviewed in Section 1.1 and Section 1.2 contains the definition of the PRAM model of parallel computation. Brief descriptions of a number of known PRAM algorithms that will be used as building blocks of the algorithms in later chapters are collected in Section 1.3.

New parallel algorithms for some selection problems which will be used in the development of the k shortest paths algorithm are presented in Chapter 2. Section 2.1 contains an $O(\log n \log\log n)$-time algorithm that performs $O(n)$ work on an EREW PRAM to solve the weighted selection problem. This problem is a generalization of the standard problem of selecting the kth smallest element from an unsorted array of n integers. The problem of selecting the kth smallest element in an $m \times n$ matrix with sorted columns is discussed in the remainder of Chapter 2. It will be shown that the problem can be solved using $O(\log k \log n \log^* n)$ time and $O(n \log(k/n))$ work if $k > n$, or $O(\log n \log^* n + \log k(\log m \log^* k + \log\log k))$ time and $O(n)$ work if $k \leq n$.

A parallel algorithm for the k shortest paths problem appears in Chapter 3. The algorithm computes an implicit representation of the k shortest paths from each vertex to a given destination vertex in an edge-weighted graph. The algorithm uses $O(\log n(\log^* n + \log\log k) + \log^3 k \log^* k)$ time and $O(m + nk \log^2 k)$ work for an n-node graph with m edges, in addition to the resources required to find the tree of shortest paths rooted at the destination node. The paths themselves can be retrieved from the implicit representation in $O(\log n + \log k)$ time using $O(n \log n + L)$ work, where L is the length of the output.

Some applications of the k shortest paths algorithm are considered in Chapter 4. Section 4.1 contains a discussion of quickest path problems, which model the transmission of data through a network by assigning a capacity and a latency to each edge of a graph. The quickest path between two nodes minimizes the transmission time for a given amount of data. A number of variants of the problem are discussed, and the k shortest paths algorithm of Chapter 3 is used to design a parallel algorithm for the k quickest paths problem. The application of the k shortest paths algorithm to optimization problems that are generally solved sequentially using dynamic programming is illustrated in Section 4.2. As a concrete example, the list Viterbi decoding problem is presented. This problem is to estimate the state sequences of a Markov process that best explain a sequence of noisy observations of its state transitions.

Chapter 5 contains a description of some open problems related to the topics of this thesis.

1.1 Graph Terminology

Graphs can be used as a mathematical model for a wide variety of problems in which there are a finite number of objects (which are represented by vertices) and a binary relation on the objects (represented by edges). For example, the computers in a network can be represented by the vertices of a graph, and the connections between computers can be represented by the edges. This section contains the definitions of terms related to graphs.

A *directed graph* is an ordered pair $G = (V, E)$, where V is a finite set of *vertices* (or *nodes*) and E is a set of ordered pairs of vertices called *edges*. In a directed *multigraph*, there may be any number of edges between each pair of vertices. The vertex u is called the *tail* of edge (u, v) and the vertex v is called the *head* of the edge. The *indegree* of a vertex v is the number of edges of which v is the head, and the *outdegree* is the number of edges of which v is the tail. An *edge-weighted* graph is a graph in which each edge is assigned a real number. Often, the weights are restricted to be non-negative.

A *path* in a graph G is a finite sequence of vertices $v_1 \to v_2 \to \ldots \to v_l$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < l$. A path is *simple* if no vertex appears in it more than once. If $v_1 = v_l$, then the path is called a *cycle*. A graph is *acyclic* if it does not have any cycles.

A *subgraph* of the directed graph $G = (V, E)$ is a directed graph $G' = (V', E')$ for which $V' \subset V$ and $E' \subset E$.

An *in-tree* is a directed acyclic graph in which each node has outdegree one, except for one distinguished node, called the *root* of the tree, which has outdegree zero. If (u, v) is an edge in the tree, u is the *child* of its *parent*, v. Two children of the same parent are called *siblings*. Nodes that have no children are called *leaves*. If there is a path from u to v, then v is called an *ancestor* of u, and u is v's *descendant*. Under this definition, each node is an ancestor and a descendant of itself. Trees in which each node has at most two children are called *binary* trees. The *depth* of a node in a tree is the number of edges along the unique simple path from the node to the root, and the *height* of a tree is the depth of the deepest node. An *out-tree* is the same as an in-tree, except that edges are directed from nodes to their children. Unless stated otherwise, trees will be in-trees.

1.2 The PRAM Model

The parallel random-access machine (PRAM) is a widely used abstract model for parallel computation (see [26] and [36] for a description of the PRAM model and a wide variety of PRAM algorithms). It is a parallel version of the sequential random-access machine (RAM) model. The PRAM consists of a set of processors that have private memories and also have access to a shared memory. The memories are made up of registers, each of which can hold a finite number of bits. As in the RAM model, the size of each register is assumed to be logarithmic with respect to the size of the problem instance. For example, if the input to the algorithm is an edge-weighted graph, a single register will be large enough to store one vertex label or one edge weight.

The computation proceeds in synchronous steps; each processor can perform one instruction during each step. The instruction set allows a processor to read or overwrite one memory register (using direct or indirect addressing), perform an arithmetic operation or bit manipulation, or conditionally branch.

In the exclusive-read exclusive-write (EREW) PRAM model, at most one processor can access each shared register during each step. In the concurrent-read exclusive-write (CREW) PRAM, two processors are not allowed to write into the same memory register simultaneously, but each register may be read by any number of processors in a single step. In the concurrent-read concurrent-write (CRCW) PRAM, the contents of each register may be written or read by more than one processor during a single step. When several processors write to the same shared memory location, the new contents of the register can be determined in several ways. In the Common CRCW model, all processors which attempt to overwrite the contents of the same register during a single step must write the same value. In the Arbitrary CRCW model, different processors may attempt to write different values to the same register, and an algorithm must work correctly regardless of which of the values actually gets written. In the Priority CRCW model, each processor is assigned a unique number, and write-conflicts are resolved in favour of the processor with the lowest number. Listed in decreasing order by power, the PRAM models are

$$\text{Priority CRCW} > \text{Arbitrary CRCW} > \text{Common CRCW} > \text{CREW} > \text{EREW};$$

an algorithm that runs on one model will run unchanged on any of the more powerful models. See Fich's survey [14] for a detailed account of the relationships between the various PRAM models.

There are several measures that can be used to assess the performance of an algorithm. The (worst-case) running time of an algorithm, $T(n)$ is defined to be the maximum number of parallel steps performed by the algorithm on an input of size n. The work performed by an algorithm, $W(n)$, is defined to be the product of $T(n)$ and the number of processors used to solve a problem instance of size n. When designing PRAM algorithms, one generally seeks algorithms that have polylogarithmic running times (*i.e.*, $T(n) = O(\log^{c_1} n)$ for some constant c_1) and perform a polynomial amount of work (*i.e.*, $W(n) = O(n^{c_2})$ for some constant c_2). Performance bounds may also be stated in terms of more than one parameter.

Any parallel algorithm that performs $W(n)$ work can be converted into a sequential algorithm whose running time is $W(n)$, since a single processor can simulate each parallel step performed by p processors in p sequential steps. Therefore, it is reasonable to compare the work performed by a parallel algorithm to the time used by a sequential algorithm for the same problem to determine which algorithm uses more computational effort.

1.3 Basic PRAM Algorithms

A number of PRAM algorithms which will be used as building blocks for algorithms in later chapters are briefly described here.

1.3.1 Prefix Sums and Processor Allocation

Given an array X of n integers, the prefix sum problem is to compute the sums $\sum_{i=1}^{k} X[i]$ for $1 \leq k \leq n$. The prefix sum problem can be solved easily using $O(\log n)$ time on an n-processor EREW PRAM (Ladner and Fischer, [31]). Without loss of generality, suppose the size of the array is a power of two. The elements can be assigned to the leaves of a complete binary tree of height $\log n^{\dagger}$. First, the sum of the leaves of each complete subtree is computed, beginning at the bottom of the tree and working towards the root. This takes $\log n$ steps. Then the sum of the leaves to the left of the subtree rooted at each node is computed, starting at the root of the tree and working towards the leaves. This also takes $\log n$ steps. This algorithm also works if the addition operation is replaced by any associative binary operation.

†All logarithms have base 2.

The number of processors can be reduced to $n/\log n$ without increasing the running time of the algorithm. The input array is broken into blocks of $\log n$ elements. One processor is assigned to each block to compute prefix sums within the block using the obvious sequential algorithm. The sum for each block is entered into an array Y of length $n/\log n$, and the algorithm of the previous paragraph is used to compute the prefix sums of Y. Each prefix of X consists of a number of complete blocks and a prefix of one of the blocks, and its sum can therefore be computed by a single addition of a prefix sum of Y and a prefix sum from its block. The amount of work performed is $O(n)$, and this matches the sequential complexity of the problem. In general, if $p \leq n$ processors are available, this approach can be used to obtain a prefix sum algorithm that runs in $O(\log p + n/p)$ time by using p blocks of size n/p.

Cole and Vishkin [11] describe a slightly faster prefix sum algorithm for $O(\log n)$-bit integers which runs in $O(\log n/\log\log n)$ time and performs $O(n)$ work on an Arbitrary CRCW PRAM.

The prefix sum algorithm is often used to allocate processors to tasks. Suppose there are n tasks, and that task i requires p_i processors for $1 \leq i \leq n$. If $p = \sum_{i=1}^{n} p_i$ processors numbered 1 to p are available, the sums $S(k) = \sum_{i=1}^{k} p_i$ can be computed and the processors numbered $S(k-1)+1$ to $S(k)$ can be assigned to task k.

1.3.2 Merging

Borodin and Hopcroft [4] describe a CREW PRAM implementation of Valiant's algorithm [42] to merge two sorted arrays X and Y of length n and m that runs in $O(\log\log n)$ time using $n+m$ processors. X and Y are divided into blocks of size $\sqrt{n}$ and $\sqrt{m}$, respectively, and then the position in the output array for the first element of each block of X is determined. This is done by comparing the first element of each block of X with the first element of each block of Y to determine which block of Y it fits into, and then comparing it with each element within that block. The problem is then solved by recursively solving $\sqrt{n}$ disjoint subproblems: one for each block of X.

The technique used to reduce the number of processors required by the prefix sum algorithm can also be applied to the merging algorithm. Suppose that only $(n+m)/\log\log n$ processors are available. Let Z be the array of size $n+m$ in which the merged array is to be stored. Each of the two input lists are divided into blocks of size $\log\log n$ and the arrays $\tilde{X}$ and $\tilde{Y}$ are formed by taking the first element of each block of X and Y (see Figure 1.1). $\tilde{X}$ and $\tilde{Y}$ can be merged using Borodin and Hopcroft's algorithm. This determines the index in Z of each element of $\tilde{X}$ within $\log\log n$, since it is known how many complete blocks of Y must precede each element of $\tilde{X}$. To find the exact index of each element of $\tilde{X}$, a sequential binary search of the appropriate block of Y can be performed. The elements of $\tilde{Y}$ can be entered into Z in the same way. When all of the elements of $\tilde{X}$ and $\tilde{Y}$ are placed into Z, the $(n+m)/\log\log n$ gaps in Z will each be of length at most $2\log\log n - 2$. One processor can be allocated to fill in each gap using a sequential merge algorithm on the pair of appropriate blocks of X and Y. The running time of the CREW PRAM algorithm is still $O(\log\log n)$, but only $O(m+n)$ work is performed.

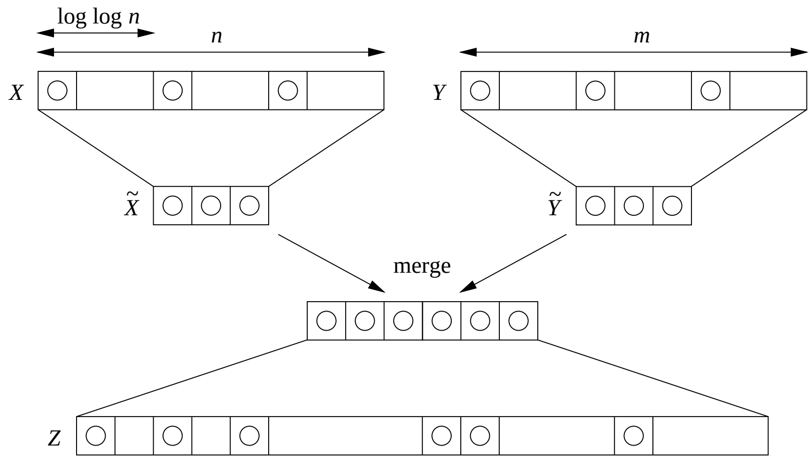


Figure 1.1: Reducing the number of processors to merge two arrays

1.3.3 Sorting

The sequential complexity of sorting an array of n elements drawn from a totally ordered set using comparisons is $\Theta(n \log n)$. The fastest known parallel sorting algorithm that performs $O(n \log n)$ work is Cole's parallel merge sort [10], which runs in $O(\log n)$ time on an n-processor EREW PRAM. The algorithm stores the elements of the unsorted array at the leaves of a complete binary tree. For each internal node, the sorted array of the node's leaf descendants is computed. In order to achieve the running time $O(\log n)$, the algorithm begins to compute the arrays for internal nodes before their children's arrays have been completely determined. The algorithm proceeds in stages, successively refining approximations to each node's array, using information received from the approximate arrays computed at the node's children in the previous stage.

1.3.4 Selection

The time required to select the kth smallest element of an unsorted array of n elements using a sequential algorithm is $\Theta(n)$. This was originally shown by Blum *et al.* [3]. Schönhage, Paterson and Pippenger [39] gave a more complicated algorithm which reduced the number of comparisons by a constant factor. The algorithm of Blum *et al.* was adapted to run in $O(\log n \log\log n)$ time and $O(n)$ work on an EREW PRAM by Vishkin [43]. Vishkin's algorithm is described in Section 2.1, below, where it is modified to solve the more general problem of weighted selection. Cole [9] developed a faster parallel algorithm, which performs $O(n)$ work in $O(\log n \log^* n)$ time on an EREW

PRAM, using an approach that originally appeared in Floyd and Rivest's randomized sequential selection algorithm [15]. Cole's algorithm runs in $\log^* n$ stages, reducing the number of candidates by computing two approximations to the kth element: one that is larger than the required element and one that is smaller than it. All candidates outside this range are discarded before the following stage. After $\log^* n$ stages, the remaining candidates can be sorted to find the required element.

1.3.5 Tree Contraction

Parallel tree contraction is a useful technique for designing parallel algorithms that work quickly even on poorly-balanced trees. Miller and Reif [32] gave the original version of tree contraction. Abrahamson *et al.* [1] and Kosaraju and Delcher [30] independently gave the improved version described below, which performs less work.

Here, it is assumed that each internal node has exactly two children. If an internal node in the input tree has only one child, an extra dummy child may be added. If an internal node has $c > 2$ children, it can be replaced by a binary tree with $c - 1$ dummy internal nodes whose leaves are the children of the original internal node. These transformations only increase the size of the tree by a constant factor.

Tree contraction consists of a sequence of basic operations, called shunts, which reduce the tree to a single node. The shunt(v) operation deletes a leaf v and its parent and reattaches the sibling of v in the former location of v's parent (Figure 1.2). Each leaf has a sibling, since the shunt operation preserves the property that every internal node has exactly two children. Shunts are performed in parallel on separate parts of the tree and can be organized so that any tree of n nodes is reduced to a single node, which was a leaf of the original tree, in $O(\log n)$ time and $O(n)$ work on an EREW PRAM.

Tree contraction can be used to compute the minimum ancestor of each node in a node-weighted tree. This application of tree contraction will be used in Chapter 3, below.

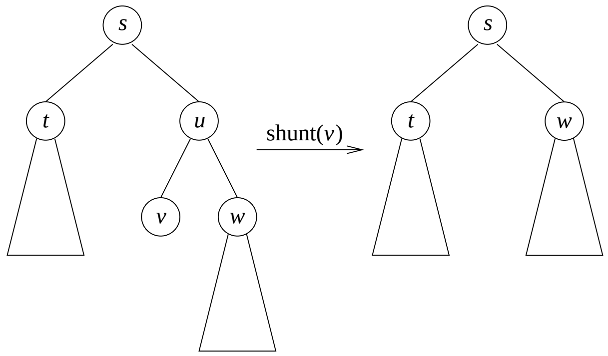


Figure 1.2: A shunt operation

Computing Minimum Ancestors

Suppose each node in a tree is assigned an integer value. Reid-Miller, Miller and Modugno [35] give an algorithm to compute, for each node v, the minimum value assigned to any of v's ancestors. Suppose that the minimum of v's ancestors is to be stored in a variable m_v. The computation of m_v proceeds in two stages. First, tree contraction is applied to reduce the tree to a single node. During this stage of the algorithm, m_v is used to store the minimum value of any ancestor of v in the original tree that was below its current parent (Figure 1.3). Then, the tree contraction process is run in reverse, expanding the single node into the full tree by undoing the shunt operations. During this second stage, the final values of the variables m_v are computed as the shunt operations are undone (Figure 1.4). This technique allows the minimum ancestors problem to be solved in $O(\log n)$ time using $O(n)$ work on an EREW PRAM.

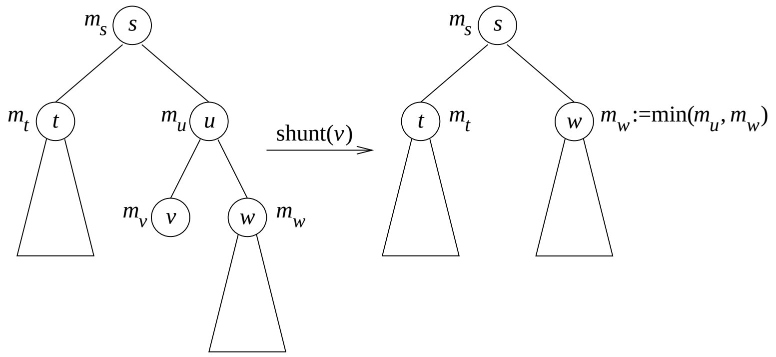


Figure 1.3: The shunt operation used to compute minimum ancestors

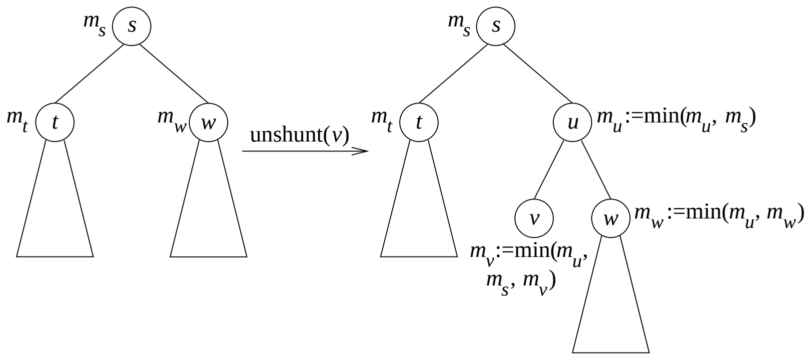


Figure 1.4: Undoing a shunt operation when computing minimum ancestors

Chapter 2

Some Parallel Selection Algorithms

The problem of selecting the kth smallest integer from an unsorted array is a fundamental one in computer science. Here, two variations on this problem are considered and parallel algorithms are developed for them. For the weighted selection problem, integers in an unsorted array are assigned weights. An element must be selected from the array, not according to the number of smaller elements, but rather their total weight. In the second variation, the input is given as a matrix of integers, each of whose columns is sorted, and this partial ordering information will be exploited by a parallel algorithm to select the kth smallest element. The weighted selection algorithm will be used as a subroutine of the algorithm for selection in a matrix with sorted columns, which in turn will be used in the next chapter to solve the k shortest paths problem.

2.1 The Weighted Selection Problem

Let $X = [x_1, ..., x_n]$ be an unsorted array of n distinct integers. A non-negative integer weight, w_i, is associated with each element x_i. Given a number K, the problem of selecting the Kth weighted element of X is to compute $x^* = \min\{x_i \mid \sum_{x_j \leq x_i} w_j \geq K\}$. The problem of selecting the Kth smallest element of X is a special case of this problem, when each weight is one.

Johnson and Mizoguchi [27] give a sequential algorithm for this problem that runs in $O(n)$ time. The algorithm executes a binary search for x^*. The median of X is found, and the sum s of the weights of the array elements less than the median is computed. If this sum is less than K, the elements less than median are discarded and K is reduced by s; otherwise the elements larger than the median are discarded, and K remains unchanged. This reduces the problem to one that is half as large, and the smaller problem is solved by a recursive call to the algorithm.

The straightforward parallel version of this algorithm would run in $\Omega(\frac{\log^2 n}{\log\log n})$ time, since the summation of the weights of elements less than the median requires $\Omega(\frac{\log n}{\log\log n})$ parallel steps, if the number of processors is polynomial in n [2], and over the course of the algorithm this contributes

$\Omega(\sum_{i=0}^{\log n} \frac{\log(n/2^i)}{\log\log(n/2^i)}) = \Omega(\frac{\log^2 n}{\log\log n})$ steps to the running time of the algorithm.

A better algorithm can be obtained by modifying Vishkin's selection algorithm [43] to solve this more general problem. Vishkin's original algorithm selects the kth smallest element of an unsorted array of n integers in $O(\log n \log\log n)$ time on an EREW PRAM with $p = n/(\log n \log\log n)$ processors. The version of the algorithm that solves the weighted selection problem uses the same amount of time and the same number of processors as the original algorithm.

The algorithm partitions the matrix into p subarrays, finds the median, a, of the medians of these groups, and then eliminates either those elements that are larger than a or those that are smaller. The algorithm is then called on the smaller array of remaining elements. The base case of the recursion occurs when the size of the problem is reduced to p, at which time the solution can be found by sorting the array. A detailed description of the algorithm is shown in Figure 2.1.

```
WeightedSelect(X, K)
    Broadcast n and K to all processors
    If X has at most p elements
        Sort the elements of X using Cole's parallel merge sort [10]
        Compute the prefix sums of the sorted array
        Output the leftmost element whose corresponding prefix sum is at least K
    Else
        Divide X into p groups, each with either ⌈n/p⌉ or ⌊n/p⌋ elements
        Use one processor per group to find the median of each group with a linear-
            time sequential algorithm
        Sort the p medians, using Cole's parallel merge sort
        Broadcast a, the median of the medians, to all processors
        s := Σ_{x_i ≤ a} w_i
        If s ≥ K
            Collect the elements of X that are less than or equal to a into array Y
            Return WeightedSelect(Y, K)
        Else
            Collect the elements of X that are greater than a into array Y
            Return WeightedSelect(Y, K − s)
```

Figure 2.1: Weighted Selection Algorithm

2.1.1 Performance Analysis

The broadcasts, which are necessary because memory registers can be read by only one processor at a time, can be done in $O(\log p)$ time, if the processors are organized (conceptually) into a balanced binary tree, and each processor passes the information to its children in the tree, starting from the root. The time to compute the median of each of the subarrays is $O(n/p)$. The medians can be sorted to find a in $O(\log p)$ time. To compute the sum, s, of the weights of elements smaller than or equal to a, the prefix sum algorithm of Section 1.3.1 is used, with the weights of elements greater than a replaced by 0. Using the p processors available, the sum can be computed in $O(\log p + n/p)$ time. The elements of the array Y can be extracted from X in the same amount of time by assigning 1 to each element that is to be retained and 0 to the others, and performing a prefix sum operation on these bits to obtain the location in array Y for each extracted element.

Because at least one quarter of the elements of the array are discarded at each level of recursion, the number of elements will be reduced to p after $O(\log(n/p)) = O(\log\log n)$ recursive calls. The running time of the final call to WeightedSelect is $O(\log p)$. The total running time for the algorithm is therefore $O(\sum_{i=0}^{O(\log\log n)}(\log((\frac{3}{4})^i n) + (\frac{3}{4})^i n/p) + \log p) = O(\log n \log\log n + n/p) = O(\log n \log\log n)$. This completes the proof of the following theorem.

Theorem 2.1 *The weighted selection problem on an array of size n can be solved in $O(\log n \log\log n)$ time using $O(n)$ work on an EREW PRAM.*

2.2 Selection in a Matrix With Sorted Columns

Let M be an $m \times n$ matrix whose elements are drawn from a totally ordered set. The entries in each column of M are sorted into non-decreasing order. The remainder of the chapter contains parallel algorithms for the problem of selecting the kth smallest element of M for a given k, where $1 \leq k \leq mn$. Frederickson and Johnson describe a sequential solution that runs in $O(n)$ time if $k \leq n$, and $O(n \log(k/n))$ time otherwise, and they give a matching lower bound for comparison-based algorithms [19]. Their approach will be described in Section 2.4, below, where it will be used in the development of a parallel algorithm for the problem.

Some similar selection problems, where the input is constrained in some way, have been studied. The Cartesian sum of two vectors $X = (x_1, \ldots, x_n) \in \mathbb{R}^n$ and $Y = (y_1, \ldots, y_n) \in \mathbb{R}^n$ is the $n \times n$ matrix M whose elements are defined by $M_{ij} = x_i + y_j$. Johnson and Mizoguchi describe an $O(n \log n)$ algorithm for selection in the Cartesian sum of a pair of unsorted vectors. If the vectors X and Y are sorted, then the Cartesian sum matrix is an example of a sorted matrix; the elements of each column and of each row are sorted into non-decreasing order. Frederickson and Johnson [20] and Mirzaian and Arjomandi [33] both give $O(n)$ algorithms for selection in an $n \times n$ sorted matrix. A technique similar to the one used by Mirzaian and Arjomandi will be used in Section 2.3 to obtain a parallel algorithm for selection in a matrix with sorted columns. Sarnath and He [38] use an approach similar to the algorithm described in [20] to develop an EREW PRAM algorithm for selection in a sorted matrix that runs in $O(\log n \log\log n \log^* n)$ time and performs $O(n \log\log n)$ work.

In the following two sections, two EREW PRAM algorithms will be described for selection in an $m \times n$ matrix with ordered columns. The first recursively selects two approximations to the required element, one no larger than the kth element and one no smaller, and then selects from

	$k \leq n$		$k > n$	
	Time	Work	Time	Work
Algorithm of Section 2.3	$O(\log n \log^* n + \log m \log k \log^* k)$	$O(n + k \log m)$	$O(\log m \log n \log^* n)$	$O(n \log m)$
Algorithm of Section 2.4	$O(\log n \log^* n + \log k \log \log k + \log m \log k \log^* k)$	$O(n)$	$O(\log \frac{k}{n} \log n \log \log n + \log n \log \frac{mn}{k} \log^* n)$	$O(n \log \frac{k}{n})$
Algorithm of Section 2.5	-	-	$O(\log k \log n \log^* n)$	$O(n \log \frac{k}{n})$

Table 2.1: Performance of the parallel algorithms for selection in an $m \times n$ matrix with sorted columns

among the elements between the two estimates. The second algorithm uses several techniques to discard sections of the matrix which could not possibly include the kth smallest element, and then chooses from among the remaining elements. In Section 2.5, the two algorithms are combined to produce an algorithm that is better than either one individually, for some values of the parameters. The performance bounds are summarized in Table 2.1.

2.3 A Recursive Algorithm for Selection in a Matrix With Sorted Columns

The EREW PRAM algorithm of this section, Select(M, k), selects the k^{th} smallest element of an $m \times n$ matrix M with sorted columns in $O(\log m \log n \log^* n)$ time, performing $O(n \log m)$ work. The algorithm uses a technique similar to Mirzaian and Arjomandi's sequential algorithm for selection in a matrix whose rows and columns are both sorted [33]. The algorithm uses a recursive routine, Biselect(M, k_1, k_2), which selects the k_1th-smallest element and the k_2th-largest element of a matrix M with sorted columns. The Biselect routine calls itself on a matrix that is half as large as M. The elements of M that are between the two elements, a and b, returned by the recursive call are gathered into an array, L. L will contain $O(n)$ elements, and the two required elements will be among them, so a standard selection algorithm can then be used on L to find them.

Let rank$^-(M, x)$ be the number of elements of M that are less than x, and let rank$^+(M, x)$ be the number of elements of M that are greater than x. Let $\overline{M}$ denote the matrix that is obtained from M by discarding the even-numbered rows.

For simplicity, assume that the number of rows in M is $2^i + 1$ for some i so that $M, \overline{M}, \overline{\overline{M}}, \ldots$ each have an odd number of rows. If necessary, extra rows filled with infinite values may be added to the bottom of the matrix to ensure that this assumption holds. These extra rows will at most double the height of the matrix, and need not be explicitly added to the input matrix; instead, any access to an element below the last row of the input matrix will return ∞. Without loss of generality, the elements of M may be assumed to be distinct: if two elements in the input matrix are identical, the leftmost one (or the topmost, if the two elements appear in the same column) can be considered the smaller of the two.

The algorithm uses two global array variables, called ra and rb, each containing n integers in the range $0..m$. These arrays represent the upper and lower boundary of the region of M that contains the elements of L, which contains the elements between the values a and b, inclusive. All elements of the jth column of M except the first $\mathrm{ra}[j] = \mathrm{rank}^-(j\text{th column of } M, a)$ and the last $\mathrm{rb}[j] = \mathrm{rank}^+(j\text{th column of } M, b)$ appear in the L. A description of the algorithm is given in Figure 2.2. Details regarding the implementation of the algorithm on an EREW PRAM are given below, in the analysis of the performance of the algorithm.

```
Select(M, k)
    (a, b) := Biselect(M, k, mn − k + 1)
    Return a

Biselect(M, k₁, k₂)
    If M has only one row then
        Select the k₁th smallest element, a, and k₂th largest element, b, of M using
            a standard parallel selection algorithm
        For each j between 1 and n, compute
            ra[j] = rank⁻(jth column of M, a)
            rb[j] = rank⁺(jth column of M, b)
        Return (a, b)
    Else
        k̄₁ := ⌈k₁/2⌉
        k̄₂ := ⌈k₂/2⌉
        (a, b) := Biselect(M̄, k̄₁, k̄₂)
        For each j between 1 and n, compute
            ra[j] := rank⁻(jth column of M, a) and rb[j] := rank⁺(jth column of M, b)
                using the previous values of ra and rb
        Collect into an array L the elements M_ij of M for which a ≤ M_ij ≤ b.
        Compute rank⁻(M, a) = Σ_{j=1}^{n} ra[j]
        Compute rank⁺(M, b) = Σ_{j=1}^{n} rb[j]
        Select and return the (k₁ − rank⁻(M, a))th smallest and (k₂ − rank⁺(M, b))th
            largest elements of L using a standard selection algorithm
```

Figure 2.2: The Select algorithm

2.3.1 Correctness of the Algorithm

The following lemma will be used to show that the array L of elements between a and b (which are computed by the recursive call to Biselect) contains the two elements that are to be returned by the

Biselect algorithm.

Lemma 2.2 $\text{rank}^-(M,x) \leq 2\text{rank}^-(\overline{M},x)$
$\text{rank}^+(M,x) \leq 2\text{rank}^+(\overline{M},x)$

Proof: $\overline{M}$ is formed by discarding the even-numbered rows of M. Associate with each element in an even-numbered row of M the element immediately above it. The element associated with each discarded element of M that is less than x appears in $\overline{M}$ and must also be less than x. Thus, the number of discarded elements that are less than x is smaller than the number of elements less than x that remain in $\overline{M}$. The first inequality follows. The second inequality can be proved using a symmetric argument, since the number of rows in M is assumed to be odd. □

This lemma is illustrated in Figure 2.3 for a 5×6 matrix. The elements that are discarded to form $\overline{M}$ are marked with an $\times$. Suppose that elements above the solid line are the $\text{rank}^-(M,x)$ elements that are smaller than x. The number of elements above the dotted line is $2\text{rank}^-(\overline{M},x)$.

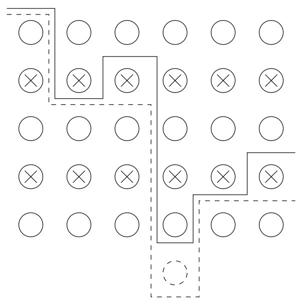


Figure 2.3: Illustration of Lemma 2.2

The following lemma will be used to show that L is non-empty. It will later be used in the analysis of performance to bound the size of L, and to bound the resources required to compute the values of ra and rb.

Lemma 2.3 *Whenever* Biselect *is called during the execution of the* Select *algorithm,* $|M| - n \leq k_1 + k_2 \leq |M| + 1$, *where* $|M|$ *denotes the number of elements in the matrix* M.

Proof: The first time Biselect is called, $k_1 + k_2 = |M| + 1 \geq |M| - n$.

Now suppose that the lemma holds for some call to Biselect. Then,

$$\begin{aligned}
\overline{k_1} + \overline{k_2} &\geq k_1/2 + k_2/2 \\
&\geq (|M| - n)/2 && \text{(by the induction hypothesis)} \\
&= (mn - n)/2 \\
&= (m+1)n/2 - n \\
&= |\overline{M}| - n && \text{(since } m \text{ is odd)}
\end{aligned}$$

$$
\begin{aligned}
\overline{k_1} + \overline{k_2} &\leq \frac{k_1+1}{2} + \frac{k_2+1}{2} \\
&\leq \frac{nm+1}{2} + 1 && \text{(by the induction hypothesis)} \\
&\leq (m+1)n/2 + 1 \\
&= |\overline{M}| + 1 && \text{(since } m \text{ is odd)}
\end{aligned}
$$

The lemma will therefore hold for all calls to Biselect. □

Lemma 2.4 *Whenever* Biselect(M, k_1, k_2) *is called during the execution of the* Select *algorithm, it returns the k_1th smallest and k_2th largest elements of M.*

Proof: The Biselect algorithm is clearly correct if M has only one row. Now consider the case where M has more than one row. Assume that the recursive call correctly returns the $\overline{k_1}$th smallest and $\overline{k_2}$th largest elements of $\overline{M}$. Then,

$$
\begin{aligned}
\text{rank}^-(M, a) &\leq 2\text{rank}^-(\overline{M}, a) && \text{(by Lemma 2.2)} \\
&= 2(\overline{k_1} - 1) && \text{(by the induction hypothesis)} \\
&\leq k_1 - 1
\end{aligned}
$$

Similarly, $\text{rank}^+(M, b) \leq k_2 - 1$. L contains all elements of M between a and b, inclusive, and therefore includes all elements between the k_1th smallest and k_2th largest elements, inclusive. L is non-empty, since $k_1 + k_2 \leq |M| + 1$, by Lemma 2.3. Thus, the k_1th smallest element of M is the $(k_1 - \text{rank}^-(M, a))$th element of L, and the k_2th largest element of M is the $(k_2 - \text{rank}^+(M, b))$th largest element of L, and these elements are returned by the Biselect routine. It follows by induction that Biselect always returns the k_1th smallest and k_2th largest elements of M. □

The correctness of the Select algorithm follows immediately from the correctness of the Biselect algorithm.

2.3.2 Performance Analysis

To execute the Select algorithm, $\lceil \log m \rceil$ calls to the Biselect routine are performed. It will be shown that each call to Biselect uses $O(\log n \log^* n)$ time and performs $O(n)$ work, excluding the recursive call.

Each time Biselect is called, the values of ra$[j]$ and rb$[j]$ must be updated. The resources required to perform this update are now considered. If M has only one row, the values can be computed in constant time, performing $O(n)$ work, by comparing each element to a and to b. Otherwise, the values saved in ra and rb by the recursive call can be used to compute the updated values. In the following discussion, bars over variable names will be used to denote the values that the variables are given by the recursive call, to differentiate them from the value that they are assigned by the primary call to Biselect.

Lemma 2.5 *During each call of* Biselect *in which M has more than one row,*
(i) $a \geq \overline{a}$, $b \leq \overline{b}$,
(ii) $2\,\overline{\mathrm{ra}[j]} - 1 \leq \mathrm{ra}[j] \leq m + 1 - 2\,\overline{\mathrm{rb}[j]}$, *for* $1 \leq j \leq n$,
(iii) $2\,\overline{\mathrm{rb}[j]} - 1 \leq \mathrm{rb}[j] \leq m + 1 - 2\,\overline{\mathrm{ra}[j]}$, *for* $1 \leq j \leq n$, *and*
(iv) the number of elements collected into the array L is at most $3n + 2$.

Proof: *(i)* If $\overline{M}$ has only one row, then $a = \overline{a}$ and $b = \overline{b}$. Otherwise,

$$\begin{aligned}
\mathrm{rank}^-(\overline{M}, \overline{a}) &\leq 2\,\mathrm{rank}^-(\overline{\overline{M}}, \overline{a}) \qquad \text{(by Lemma 2.2)} \\
&= 2(\overline{\overline{k_1}} - 1) \\
&= 2(\left\lfloor \frac{\overline{k_1} + 1}{2} \right\rfloor - 1) \\
&\leq \overline{k_1} - 1 \\
&= \mathrm{rank}^-(\overline{M}, a)
\end{aligned}$$

It follows that a is at least as big as $\overline{a}$. The proof that $b \leq \overline{b}$ is symmetric.

(ii) For any j,

$$\begin{aligned}
\mathrm{ra}[j] &= \mathrm{rank}^-(j\text{th column of } M, a) \\
&\geq \mathrm{rank}^-(j\text{th column of } M, \overline{a}) \qquad (\text{since } a \geq \overline{a}) \\
&\geq 2\,\mathrm{rank}^-(j\text{th column of } \overline{M}, \overline{a}) - 1 \\
&\qquad \text{(if } r \text{ elements of column } j \text{ of } \overline{M} \text{ are less than } \overline{a}\text{, then at least } 2r-1 \\
&\qquad \text{elements of column } j \text{ of } M \text{ are less than } \overline{a}) \\
&= 2\,\overline{\mathrm{ra}[j]} - 1
\end{aligned}$$

Similarly, $\mathrm{rb}[j] \geq 2\,\overline{\mathrm{rb}[j]} - 1$. By Lemma 2.3, $\overline{k_1} + \overline{k_2} \leq |\overline{M}| + 1$, so the $\overline{k_1}$th smallest element of $\overline{M}$, a, is smaller than or equal to the $\overline{k_2}$th largest element of $\overline{M}$, b. The elements of M are assumed to be distinct, so $\mathrm{ra}[j] + \mathrm{rb}[j] \leq m$. It follows that $\mathrm{ra}[j] \leq m - \mathrm{rb}[j] \leq m - 2\overline{\mathrm{rb}[j]} + 1$.

(iii) Symmetric to *(ii)*.

(iv) The number of elements in the array L is

$$\begin{aligned}
&\mathrm{rank}^-(M, b) - \mathrm{rank}^-(M, a) + 1 \\
= \; &\mathrm{rank}^-(M, b) - (mn - \mathrm{rank}^+(M, a) - 1) + 1 \\
\leq \; &2(\mathrm{rank}^-(\overline{M}, b) + \mathrm{rank}^+(\overline{M}, a)) - mn + 2 \qquad \text{(by Lemma 2.2)}
\end{aligned}$$

$$
\begin{aligned}
&= 2(\frac{m+1}{2}n - \overline{k_2} + \frac{m+1}{2}n - \overline{k_1}) - mn + 2 \qquad (m \text{ is odd}) \\
&= 2(m+1)n - 2(\overline{k_1} + \overline{k_2}) - mn + 2 \\
&\leq 2(m+1)n - 2(\frac{m+1}{2}n - n) - mn + 2 \qquad \text{(by Lemma 2.3)} \\
&= 3n + 2 \quad \square
\end{aligned}
$$

This lemma constrains the set of candidates for the new values of ra[j] and rb[j], so that they can be computed quickly. Consider the computation of ra[j]. Column j of M is assigned $p_j = m + 3 - 2\,\overline{\text{ra}[j]} - 2\,\overline{\text{rb}[j]}$ processors. This allows one processor to be assigned to each of the possible new values of ra[j]. Section 1.3.1 describes how a prefix sum operation can be used to allocate processors to the columns in $O(\log n)$ time and $O(n)$ work.

Each possible value, i, for ra[j] can then be checked simultaneously using constant time and $O(n)$ work, by comparing M_{ij} and $M_{i-1,j}$ to a. The processor that finds the correct new value for ra[j] writes it into the variable. The amount of work performed is $O(n)$, since

$$
\begin{aligned}
\sum_{j=1}^{n} p_j &= \sum_{j=1}^{n} (m + 3 - 2\,\overline{\text{ra}[j]} - 2\,\overline{\text{rb}[j]}) \\
&= 2n + 2\sum_{j=1}^{n} (\frac{m+1}{2} - \overline{\text{ra}[j]} - \overline{\text{rb}[j]}) \\
&= 2n + 2(\text{number of elements of } \overline{M} \text{ in the interval } [\overline{a}, \overline{b}]) \\
&\leq 2n + 2(3n+2) = O(n) \qquad \text{by Lemma 2.5} \textit{(iv)}
\end{aligned}
$$

The new values of the array rb can also be computed using $O(1)$ time and $O(n)$ work in exactly the same way.

To gather the $O(n)$ elements of L into an array, $m - \text{ra}[j] - \text{rb}[j]$ processors can be allocated to retrieve the elements in column j of M. This can be done using a prefix sum, as described in Section 1.3.1. Each processor is responsible for retrieving one element of L. By Lemma 2.5*(iv)*, $O(n)$ processors are sufficient. The retrieval can therefore be done using constant time and $O(n)$ work.

Finally, the two required elements can be selected from L in $O(\log n \log^* n)$ time by $n/(\log n \log^* n)$ processors using Cole's selection algorithm (see Section 1.3.4).

Summing up, each call to Biselect uses $O(\log n \log^* n)$ time and $O(n)$ work, excluding the recursive call, when M has more than one row. If M has only one row, Biselect can also be performed in $O(\log n \log^* n)$ time using $O(n)$ work using Cole's selection algorithm. There are $\lceil \log m \rceil$ calls to Biselect during the execution of Select. This completes the proof of the following theorem.

Theorem 2.6 *The kth smallest element of an $m \times n$ matrix with sorted columns can be selected by an EREW PRAM in $O(\log m \log n \log^* n)$ time and $O(n \log m)$ work.*

2.3.3 Remarks

When Biselect makes its recursive call, it need not explicitly pass the matrix $\overline{M}$ as an argument; it is sufficient that the depth of the recursive call is known, so that only the appropriate rows of the input matrix M will be considered by the Biselect routine.

If k is less than n, one could begin by selecting the kth element of the first row, using Cole's selection algorithm, and execute the rest of the algorithm only on those columns whose top element is less than the kth element of the top row. This reduces the total running time to $O(\log n \log^* n + \log m \log k \log^* k)$ and the amount of work performed to $O(n + k \log m)$.

If k is less than m, the running time can be reduced to $O(\log k \log n \log^* n)$ by ignoring all rows below the kth.

After the Select algorithm terminates, the information that is stored in the array rb can be used to gather into an array all of the elements that are less than the kth element of M. The set of such elements is simply $\{M_{ij} \mid i \leq m - \mathrm{rb}[j]\}$. Gathering these elements using a prefix sum operation will not increase the asymptotic running time of the algorithm, but would require $O(k)$ work.

2.4 A Parallel Elimination Algorithm for Selection in a Matrix With Sorted Columns

This section contains an EREW PRAM implementation of Frederickson and Johnson's sequential algorithm to select the kth smallest element of an $m \times n$ matrix M of integers with sorted columns [19]. The sequential algorithm runs in $O(n)$ time if $k \leq n$, and $O(n \log \frac{k}{n})$ time otherwise. In Section 2.5, the parallel algorithm will be combined with the algorithm of Section 2.3 to improve the running time.

The algorithm eliminates large sections of the matrix that could not possibly include the kth element. The algorithm has four phases. After the first two phases, the number of remaining elements is $O(k)$, and after the third phase, $O(n)$ elements remain. In the fourth phase, a standard selection algorithm is used to find the required element. At any time, the elements that have not yet been eliminated will be called *active* elements.

2.4.1 Representation of the Matrix

Elements are always eliminated from the top or bottom ends of columns, so the active elements in a column always form a contiguous block. To keep track of which elements are active, it is therefore sufficient to store the indices of the first and last active elements in each column. The arrays top$[1..n]$ and bottom$[1..n]$ are used for this purpose. Initially, top$[j] = 1$ and bottom$[j] = m$ for all j.

Some parts of the algorithm require columns of M to be interchanged. To allow this operation to be performed quickly, no elements of M are actually moved; instead, the columns of M are reindexed. A permutation of the column indices is maintained in an array $c[1..n]$, so that current values of the jth column of the matrix are stored in column $c[j]$ of M.

The elements of the input matrix are never changed, and at any time, the ith active element of the jth column can be accessed in constant time by looking up the $(i - \text{bottom}[c[j]] + 1)$th element of column $c[j]$ in the original matrix.

2.4.2 Reducing the Number of Active Elements to $O(k)$

As described in Section 2.3, there is no loss of generality in considering matrices whose elements are distinct. If $m > k$, only the first k rows of the matrix must be considered, so from now on it is assumed that $m \leq k$.

If k is larger than n, the size of the matrix can also be reduced. The kth-smallest element, e, of the top row can be found using a standard algorithm for selection in an unsorted array (see Section 1.3.4), and only the columns whose first elements are no larger than e must be considered for the rest of the computation. These columns can be reindexed so that they fit into an $m \times k$ matrix by assigning a 1 to each of them and a 0 to the other columns and then performing a prefix sum operation on these bits to find the new index of each column. This preprocessing step uses $O(\log n \log^* n)$ time and performs $O(n)$ work. For simplicity, it will be assumed that n is a power of 2. If this is not the case, one can add columns of infinite values to the matrix, increasing the size of the matrix by at most a factor of 2.

The first phase of the algorithm repeatedly finds an element that is greater than or equal to at least k elements in the matrix. Any element of M that is larger than the chosen element must be larger than the kth smallest element of M, and can therefore be eliminated. The elements are chosen so that the eliminations can be performed efficiently. The algorithm chooses e_j to be the jth smallest element of row $\lceil k/j \rceil$, for $j = n/2, n/4, n/8, \ldots 1$. The columns of the matrix are rearranged so that e_j appears in column j, and all smaller elements in the same row appear to the left of e_j. Then, all elements that are above and to the left of e_j are smaller than e_j, and there are $j \lceil k/j \rceil \geq k$ such elements. All elements below and to the right of e_j are larger than e_j and can therefore be discarded. The details of this phase of the algorithm are shown in Figure 2.4.

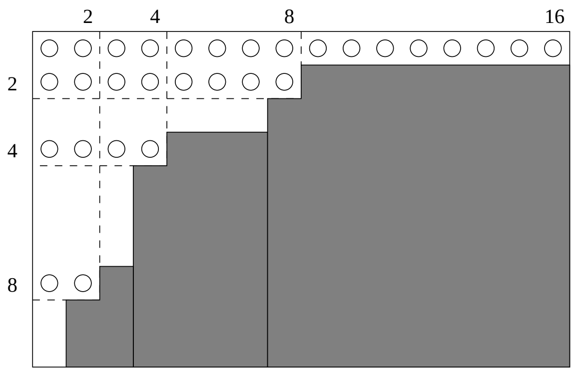
See Figure 2.5 for an illustration of phase 1 of the algorithm on a 10×16 matrix with $k = 15$. The shaded regions of the array are eliminated by the iterations of the loop in phase 1. The dashed rectangles show the elements known to be no larger than the kth element during the various iterations (each rectangle contains at least k elements). The circled positions in the matrix will play a special role in phase 2, described below.

Let Q be the set of active elements that appear in row 1 or in any of the rows that were considered in Phase 1 (the circled elements in Figure 2.5). The second phase partitions the active elements of

Phase 1 $j := n/2$
Loop
  $i := \lceil k/j \rceil$
  Exit when $i > m$
  Select the jth smallest active element, e_j, of row i
  Rearrange those columns with active elements in the ith row so that e_j appears in column j and the smaller active elements in row i are to the left of e_j
  Eliminate the elements that are below and to the right of e_j by setting bottom$[c[j]] := i$ and bottom$[c[l]] := i - 1$ for $l \in \{j+1, \ldots, 2j-1\}$
  $j := j/2$

Phase 2 Let Q be the set of active elements in the rows i that were considered in phase 1 together with the active elements of row 1
Associate with each element q of Q a weight equal to the number of active elements in q's column which are at least q and less than the next largest member of Q in the column
Compute the $(6k)$th-weighted element, q^*, of Q
For all j, bottom$[j] := \min\{i \mid M_{ij} \in Q \text{ and } M_{ij} > q^*\} - 1$ if this set is non-empty

Figure 2.4: Reducing the number of active elements to $O(k)$



iteration	j	i
1	$n/2 = 8$	$\lceil k/8 \rceil = 2$
2	$n/4 = 4$	$\lceil k/4 \rceil = 4$
3	$n/8 = 2$	$\lceil k/2 \rceil = 8$
4	$n/16 = 1$	$\lceil k/1 \rceil = 15 > m$ $\Rightarrow$ terminate

Figure 2.5: Phase 1 for a 10×16 matrix with $k = 15$

the matrix into $O(n)$ sets. Each set in the partition is a contiguous section of a column that contains exactly one element of Q as its smallest element, and this element acts as the representative of the set. Each element of Q is assigned a weight equal to the size of the set that it represents, and the $(6k)$th-weighted element, q^*, of Q is chosen using the algorithm of Section 2.1. All of the sets whose representative is greater than q^* are eliminated.

Lemma 2.7 (Frederickson and Johnson [19]) *When phase 2 of the algorithm is completed, $O(k)$ elements remain active, and the k smallest elements of M are among them.*

Proof: In phase 2, the weight assigned to each element of Q is at most m, so the total weight of the elements less than or equal to q^* can exceed $6k$ by at most m. The number of elements that remain active after phase 2 is equal to the sum of the weights of the elements of Q that are retained and is at most $6k + m \leq 7k = O(k)$ (since it was assumed that $m \leq k$).

It was argued above that no iteration of the loop in phase 1 discards any of the k smallest elements of M. Now consider phase 2. The number of elements above row $\lceil 2k/n \rceil$ (exclusive) that remain active and are larger than q^* is at most $2k$, since there are n columns. In any column, the number of elements in or below row $\lceil 2k/n \rceil - 1$ that remain active and are larger than q^* is less than half the total number of elements that remain active in that column. For, if $\lceil k/j \rceil$ is the row index of the last element of Q in a column that remains active after phase 2, there are at most $\lceil 2k/j \rceil - 1$ active elements in the column. The number of active elements in the column that are larger than q^* is then at most $\lceil 2k/j \rceil - 1 - \lceil k/j \rceil < 2k/j - k/j \leq \lceil k/j \rceil$ and at least $\lceil k/j \rceil$ active elements in the column are no bigger than q^*. Thus, if N elements remain active in the entire matrix, fewer than $N/2 + 2k$ of them will be larger than q^*. The number of elements less than q^* that remain active is at least $N - (N/2 + 2k) = N/2 - 2k \geq 6k/2 - 2k = k$ ($N \geq 6k$, since there are at least $6k$ active elements that are smaller than $q*$, by the definition of q^*). All elements less than q^* that were active after phase 1 remain active after phase 2. Therefore, phase 2 does not discard any of the k smallest elements. $\square$

The lth iteration of the loop in phase 1 can be performed in $O(\log(n/2^l)\log^*(n/2^l))$ time using $O(n/2^l)$ work. Cole's algorithm (see Section 1.3.4) is used to select the kth smallest of $n/2^l$ elements. Prefix sum operations are used to rearrange the columns; each column which has an element smaller than e_j in row i is assigned a value 1, the others are assigned a value of 0 and a prefix sum is performed to compute the locations of the columns with value 1. The columns with elements larger than e in row i are moved into their new locations in a similar way.

There are at most $\log(mn/k)$ iterations of the loop in phase 1, since i is equal to $\lceil k2^l/n \rceil$ during the lth iteration, and i exceeds m whenever l exceeds $\log(mn/k)$. Phase 1 can be executed in $O(\sum_{l=1}^{\log(mn/k)} \log(n/2^l)\log^*(n/2^l)) = O(\log(mn/k)\log n \log^* n)$ time, using $O(\sum_{l=1}^{\log(mn/k)} n/2^l) = O(n)$ work. If $k \leq n$, the n's in these bounds can be reduced to k, since the preprocessing step reduced the number of columns to k. Phase 1 therefore uses $O(\log m \log k \log^* k)$ time and $O(k)$ work when $k \leq n$.

The weighted partition performed in phase 2 can be accomplished in $O(\log n \log\log n)$ time and $O(n)$ work (or $O(\log k \log\log k)$ time and $O(k)$ work if $k \leq n$) using the weighted partition algorithm of Section 2.1, since Q has at most $2n$ elements. The computation of the new values of the array bottom can be computed in constant time with $2n$ processors by allocating 1 processor to each element of Q. Each processor checks whether its element, $M_{ij} \in Q$ is bigger than q^* and the next largest element of Q in the same column is less than or equal to q^*. If so, the processor writes $i - 1$ into bottom$[j]$.

Phase 3 $\tilde{k} := k$

Loop

  $N :=$ number of active elements in matrix

  Exit when $N < 2n$

  For $1 \leq j \leq n$, let N_j be the number of active elements in column j

  If $\tilde{k} \geq N/2$ then

    $i_j := \text{top}[j] + \lfloor N_j/4 \rfloor$ (for all $j \in \{1, \ldots, n\}$)

    Assign the weight N_j to the element $M_{i_j j}$ and find the $(N/4)$th-weighted element, e, of the set $\{M_{i_j j} \mid 1 \leq j \leq n\}$

    Let $\text{top}[j] := i_j + 1$ for those j for which $M_{i_j j} < e$ and $\text{top}[j] := i_j$ for the column j containing e.

    Reduce $\tilde{k}$ by the number of elements discarded in this iteration

  Else

    $i_j := \text{bottom}[j] - \lfloor N_j/4 \rfloor$ (for all $j \in \{1, \ldots, n\}$)

    Assign the weight N_j to the element $M_{i_j j}$ and find the $(3N/4)$th-weighted element, e of the set $\{M_{i_j j} \mid 1 \leq j \leq n\}$

    Let $\text{bottom}[j] := i_j - 1$ for those j for which $M_{i_j j} \geq e$

  End if

End loop

Phase 4 Select the $\tilde{k}$th-smallest active element using a standard selection algorithm.

Figure 2.6: Reducing the number of active elements to $O(n)$

2.4.3 Reducing the Number of Active Elements to $O(n)$

The third phase uses a loop to reduce the number of active elements to $O(n)$. During each iteration of the loop, the number of active elements is reduced by a constant factor. The elements removed during one iteration are either all smaller than the required element or all larger than it. If more than half of the active elements are smaller than the required element, the algorithm eliminates the top quarter of some columns. The largest element in the top quarter of each column is used as the representative of all of the elements above it. The weighted selection algorithm of Section 2.1 is used to find the group of columns whose top quarters contain about $N/4$ elements in total, and have the smallest representatives. The top quarters of these columns are then discarded. The case where less than half of the active elements are smaller than the required element is handled symmetrically, with elements being removed from the bottom quarters of columns. A more precise description of the algorithm appears in Figure 2.6, and the following lemma explains why the algorithm works.

Lemma 2.8 (Frederickson and Johnson [19]) *For each iteration of the loop in phase 3,*
(i) the number of active elements is reduced from N to at most $15N/16 + 1$.
(ii) If $\tilde{k} \geq N/2$ at the beginning of the iteration, only elements smaller than the $\tilde{k}$th-smallest element are discarded; otherwise only elements larger than the $\tilde{k}$th-smallest element are discarded. Hence, at the end of the iteration, the $\tilde{k}$th-smallest active element is the kth smallest element of the original matrix.

Proof: *(i)* The number of elements discarded is at least

$$\Big(\sum_{M_{i_j j}\le e} \lfloor N_j/4\rfloor + 1\Big) - 1 \ge (1/4)\Big(\sum_{M_{i_j j}\le e} N_j\Big) - 1 \ge (1/4)^2 N - 1.$$

So, the number of elements that remain active after the iteration is at most $15N/16 + 1$.
(ii) If $\tilde{k} \ge N/2$, consider the columns in which no elements are discarded, together with the column containing e. The number of elements in those columns that are larger than or equal to e is at least

$$\begin{aligned}
\sum_{M_{i_j j}\ge e} (N_j - \lfloor N_j/4\rfloor) &\ge \sum_{M_{i_j j}\ge e} 3N_j/4 \\
&= (3/4)\sum_{M_{i_j j}\ge e} N_j \\
&\ge (3/4)(3N/4) \qquad \text{(since } e \text{ is chosen so that } \sum_{M_{i_j j}< e} N_j < N/4\text{)} \\
&= 9N/16
\end{aligned}$$

So, there were at most $N - 9N/16 < N/2 \le \tilde{k}$ elements smaller than or equal to e at the start of the iteration. Thus, e is smaller than the $\tilde{k}$th-smallest element, and only elements smaller than e are discarded. Also, $\tilde{k}$ is reduced by the number of elements that were discarded. If at the beginning of the loop, the $\tilde{k}$th active element is the kth element of the original matrix, this will also be true at the end of the loop. It follows by mathematical induction that this is true at the end of each iteration, since it is clearly true at the start of the first iteration.

If $\tilde{k} < N/2$, the proof is symmetric. □

Part *(i)* of the lemma guarantees that at most $O(\log \frac{k}{n})$ iterations of the loop will reduce the number of active elements from $O(k)$ to $O(n)$. Part *(ii)* guarantees that the $\tilde{k}$th active element at the beginning of phase 4 is the kth element of the original matrix. The correctness of the algorithm follows.

Each iteration of the loop in phase 3 can be performed in $O(\log n \log\log n)$ time using $O(n)$ work using the algorithm for the weighted selection problem described in Section 2.1. The adjustments to N and $\tilde{k}$ can be performed by summing the number of elements that were discarded in each column in $O(\log n)$ time using $O(n)$ work.

Since the number of active elements has been reduced to $O(n)$, phase 4 can be accomplished using Cole's selection algorithm in $O(\log n \log^* n)$ time with $O(n)$ work.

The resources used by the algorithm are summarized in Table 2.2. The amount of work performed by the algorithm is the same as Frederickson and Johnson's original (sequential) version of the algorithm. This completes the proof of the following theorem.

	$k \le n$		$k > n$	
	Time	Work	Time	Work
Reducing number of columns if $k \le n$	$O(\log n \log^* n)$	$O(n)$	0	0
Phase 1	$O(\log m \log k \log^* k)$	$O(k)$	$O(\log n \log \frac{mn}{k} \log^* n)$	$O(n)$
Phase 2	$O(\log k \log\log k)$	$O(k)$	$O(\log n \log\log n)$	$O(n)$
Phase 3	0	0	$O(\log \frac{k}{n} \log n \log\log n)$	$O(n \log \frac{k}{n})$
Phase 4	$O(\log k \log^* k)$	$O(k)$	$O(\log n \log^* n)$	$O(n)$
Total	$O(\log n \log^* n +$ $\log k \log\log k +$ $\log m \log k \log^* k)$	$O(n)$	$O(\log \frac{k}{n} \log n \log\log n +$ $\log n \log \frac{mn}{k} \log^* n)$	$O(n \log \frac{k}{n})$
Total for algorithm of Section 2.3	$O(\log m \log n \log^* n)$	$O(n \log m)$	$O(\log m \log n \log^* n)$	$O(n \log m)$
Total for algorithm of Section 2.5	-	-	$O(\log k \log n \log^* n)$	$O(n \log \frac{k}{n})$

Table 2.2: Selecting the kth smallest element in an $m \times n$ matrix with sorted columns

Theorem 2.9 *The kth smallest element of an $m \times n$ matrix with sorted columns can be selected on an EREW PRAM in $O(\log n \log^* n + \log k(\log m \log^* k + \log\log k))$ time and $O(n)$ work if $k \le n$, and in $O(\log \frac{k}{n} \log n \log\log n + \log n \log \frac{mn}{k} \log^* n)$ time and $O(n \log \frac{k}{n})$ work if $k > n$.*

2.5 Combining the Two Selection Algorithms

For some values of the parameters, it is possible to improve the running time of the algorithm of Section 2.4 slightly, without affecting the amount of work performed, by combining it with the algorithm of Section 2.3.

The algorithm of Section 2.3 uses a recursive routine, Biselect, that selects two elements from a matrix with ordered columns. After $\log \frac{k}{n}$ levels of recursion, the recursive call to Biselect can be replaced by two calls to the parallel version of Frederickson and Johnson's algorithm. The amount of work performed before switching algorithms will be $O(n \log \frac{k}{n})$, and the running time will be $O(\log \frac{k}{n} \log n \log^* n)$, since each level of recursion contributes $O(n)$ work and uses $O(\log n \log^* n)$ time.

Each level of recursion preserves the ordered property of the columns but reduces the height of the matrix by approximately a factor of two; the number of columns remains unchanged. More precisely, after $\log \frac{k}{n}$ levels of recursion, the height of the reduced matrix, M', will be $\frac{m}{2^{log(k/n)}} + 1 = \frac{mn}{k} + 1$. The initial call to Biselect requires selection of the kth-smallest and the $(mn - k + 1)$th-largest elements of M. The size of these two parameters is also roughly halved at each level of the recursion. More precisely, after $\log \frac{k}{n}$ recursive calls, the size of these parameters, k_1 and k_2, will be at most

$\frac{k}{2^{\log(k/n)}}+1=n+1$ and at least $\frac{mn-k+1}{2^{\log(k/n)}}-1 \geq \frac{mn^2}{k}-n-1$, respectively. The k_1th-smallest element of M' can be selected in $O(\log^2 n \log^* n)$ time and $O(n)$ work using the algorithm of Section 2.4. Finding the k_2th largest element of M' is equivalent to finding the $(|M'|-k_2+1)$th-smallest of M'. But $|M'|-k_2+1 \leq (\frac{mn}{k}+1)n-(\frac{mn^2}{k}-n-1)=2n+1$. So, the algorithm described above can also be used to solve this subproblem in $O(\log^2 n \log^* n)$ time using $O(n)$ work.

The performance of this combined algorithm is summarized in the following theorem.

Theorem 2.10 *When $k>n$, the kth smallest element of an $m \times n$ matrix with sorted columns can be selected on an EREW PRAM using $O(\log k \log n \log^* n)$ time and $O(n \log \frac{k}{n})$ work.*

The time is strictly better than the parallel version of Frederickson and Johnson's algorithm when, for example, k is known to lie in the range $n^{c_1} \leq k \leq n^{c_2}$ for constants $c_2 \geq c_1 > 1$.

Chapter 3

The k Shortest Paths Problem

Let $G = (V, E)$ be a directed graph with n vertices and m edges. Each edge (u, v) of E is assigned a non-negative weight $w(u, v)$. The weight of a path in G is simply the sum of the weights of the edges that make up the path. The distance from vertex u to vertex v, $\text{dist}(u, v)$, is defined to be the weight of the path from u to v that minimizes this sum, and a path that achieves this distance is a shortest path from u to v.

The problem of finding the k shortest paths from vertex s to vertex t is to find a set $\mathcal{P}$ of k s-t paths such that the weight of any path in $\mathcal{P}$ is no larger than the weight of any s-t path not in $\mathcal{P}$. There may be several paths in $\mathcal{P}$ with the same weight. If there are fewer than k distinct paths from s to t, the solution set $\mathcal{P}$ should consist of all s-t paths.

3.1 Background

Sequential algorithms which compute shortest paths are well-known. Dijkstra's algorithm computes the shortest path from a given source vertex to all other vertices in $O(m + n \log n)$ time [21]. Applying Dijkstra's algorithm n times, once for each source vertex, yields a solution to the shortest path problem between all pairs of vertices in $O(nm + n^2 \log n)$ time.

In the PRAM model, the shortest path between each pair of vertices can be found using the well-known method of a min/sum transitive closure computation (see, for example, [34]). Suppose the vertex set of the graph is $\{1, \ldots, n\}$, and let A be the $n \times n$ matrix of edge weights: $A_{ij} = w(i, j)$. Min/sum matrix multiplication can be defined as follows: If B and C are $n \times n$ matrices, $(BC)_{ij} = \min_{1 \leq k \leq n}(B_{ik} + C_{kj})$. It can be proven inductively that $(A^l)_{ij}$ contains the length of the shortest path from vertex i to vertex j with at most l edges. Since the shortest path between two vertices is a simple path, and therefore contains at most $n-1$ edges, $(A^{n-1})_{ij} = \text{dist}(i, j)$. This matrix can be computed on an EREW PRAM in $O(\log^2 n)$ time using $O(n^3 \log n)$ work by repeatedly squaring matrix A to obtain $A^2, A^4, \ldots, A^{2^{\lceil \log(n-1) \rceil}}$. The computation of the minimum of n numbers is achieved using the prefix sum computation described in Section 1.3.1. Without altering the performance bounds, the computation of A^{n-1} can be augmented so that information about the paths themselves, as well as their lengths, is computed. For example, one can store, for each element $(A^l)_{ij}$, the second vertex along a path from i to j that consists of at most l edges and has weight $(A^l)_{ij}$. The computation of the matrix A^{n-1} solves the shortest path problem for all pairs. In contrast to the sequential case, there are no known PRAM algorithms that find the shortest path from one particular vertex to another using less work than the all-pairs algorithm. A more complicated implementation of the transitive closure computation, by Han, Pan and Reif, runs in $O(\log^2 n)$ time on the EREW PRAM while doing only $o(n^3)$ work [22]. Han *et al.* also give an $O(\log n \log\log n)$-time implementation on the CRCW PRAM which also does $o(n^3)$ work.

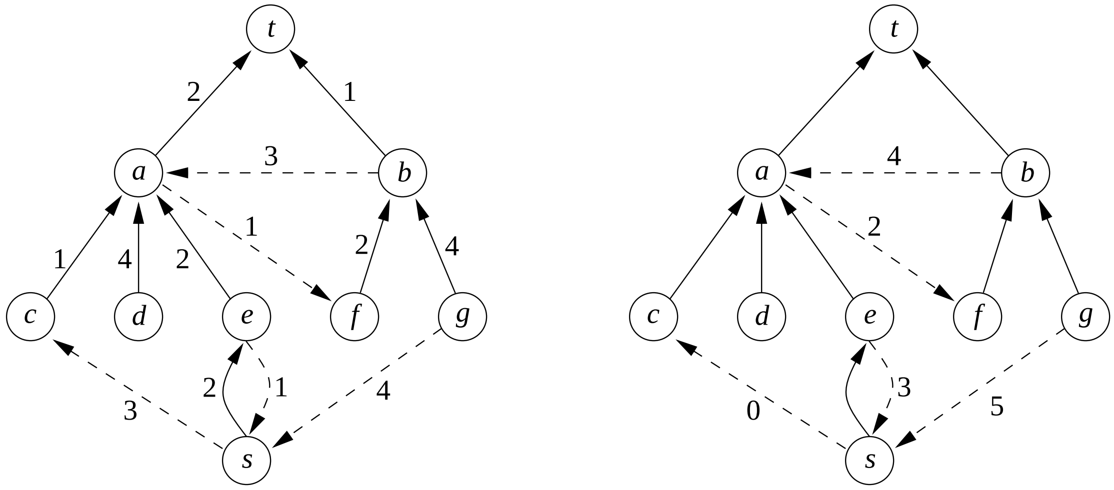
The problem of finding the k shortest paths between vertices in a graph has long been studied in sequential models of computation. The problem was discussed as early as 1959 by Hoffman and Pavley [23]. Fox presents an algorithm that can be implemented to run in $O(m + kn \log n)$ time [17]. Eppstein's recent sequential algorithm computes the k shortest paths for a given source and destination [13] in $O(m + n \log n + k)$ time. This algorithm will be described in Section 3.2, below.

Other variations of the k shortest path problem have been studied and sequential algorithms have been developed for them. Yen [45] gives an algorithm for the significantly harder problem of finding the k shortest simple paths in $O(kn^3)$ time. Katoh, Ibaraki and Mine [29] describe an $O(kn^2)$ algorithm to find the k shortest simple paths in an undirected graph. A generalization of the k shortest paths problem, called the k quickest paths problem, is discussed in Section 4.1.

3.2 A Sequential Algorithm to Compute the k Shortest Paths

Before describing Eppstein's sequential k shortest paths algorithm, some preliminaries will be presented. They will be used again in the development of a parallel algorithm for the k shortest paths problem.

Let T be a shortest path tree of G rooted at the destination vertex t. T is a subgraph of G, and the (unique) path in T from any vertex v to t is an optimal path from v to t in the input graph. The edges of the graph which do not appear in T are called non-tree edges. Any path from a fixed vertex s to vertex t can be represented by the sequence of edges along the path that are not in T. In the graph shown in Figure 3.1(a), edges of T are shown as solid lines, and non-tree edges are shown as dashed lines. In this graph, the path $s \dashrightarrow c \longrightarrow a \dashrightarrow f \longrightarrow b \longrightarrow t$ could be represented by the sequence of non-tree edges, $< (s,c),(a,f) >$. The rest of the path can be filled in by following the edges of T. If p is any path, let sidetracks(p) be the sequence of non-tree edges that occur along the path p.



(a) Solid edges are tree edges (b) Values of δ are shown for non-tree edges

Figure 3.1: An example graph

For each edge $(u, v) \in E$, one can define a measure $\delta(u, v)$ of the extra distance added to the length of a path from u to t if the edge (u, v) is used instead of taking the optimal path from u to t:

$$\delta(u, v) = w(u, v) + \mathrm{dist}(v, t) - \mathrm{dist}(u, t)$$

The following lemma describes some properties of this measure.

Lemma 3.1 (Eppstein [13])

(i) $\delta(u, v) \geq 0$ *for all* $(u, v) \in E$

(ii) $\delta(u, v) = 0$ *for all* $(u, v) \in T$

(iii) For any path p *from* s *to* t,

$$\mathrm{weight}(p) = \mathrm{dist}(s, t) + \sum_{(u,v)\in p} \delta(u, v) = \mathrm{dist}(s, t) + \sum_{(u,v)\in \mathrm{sidetracks}(p)} \delta(u, v)$$

Proof: *(i)* Since $\mathrm{dist}(u, t)$ is the weight of the shortest path from u to t, $\mathrm{dist}(u, t) \leq w(u, v) + \mathrm{dist}(v, t)$

(ii) If $(u, v) \in T$, a shortest path from u to t begins with the edge (u, v), and $\mathrm{dist}(u, t) = w(u, v) + \mathrm{dist}(v, t)$.

(iii) For any s-t path p,

$$\begin{aligned}
\mathrm{weight}(p) &= \sum_{(u,v)\in p} w(u, v) && \text{by definition of weight}(p) \\
&= \mathrm{dist}(s, t) - \mathrm{dist}(t, t) + \sum_{(u,v)\in p} (w(u, v) + \mathrm{dist}(v, t) - \mathrm{dist}(u, t))
\end{aligned}$$

$$
\begin{aligned}
&= \operatorname{dist}(s,t) + \sum_{(u,v)\in p} \delta(u,v) && \text{by definition of } \delta(u,v) \\
&= \operatorname{dist}(s,t) + \sum_{(u,v)\in \mathrm{sidetracks}(p)} \delta(u,v) && \text{by part } (ii) \qquad \square
\end{aligned}
$$

To find the k shortest paths from s to t, it is therefore sufficient to find the paths p which yield the k smallest values of $\delta(p) = \sum_{(u,v)\in p} \delta(u,v)$. If δ is viewed as a weight function on the edges of G, a general instance of the k shortest paths problem has now been transformed into an instance where the distance from any vertex to vertex t is 0. From now on, the weight function δ will be used instead of w.

Eppstein's sequential algorithm first computes the shortest path tree T, using Dijkstra's algorithm. It then constructs a new graph, G' of bounded outdegree. There will be a bijective mapping from s-t paths of G onto paths of G' that start at a particular node, s', and end at any other node of G'. (To avoid confusion, we refer to vertices of G and nodes of G'.) Weights are assigned to the edges of G' so that this correspondence is weight-preserving. This will reduce the problem of finding the k shortest s-t paths in G to the problem of finding the k shortest paths of G' that start at s'. The resulting problem can be solved directly.

Each node of G' (except for the distinguished node s') represents a non-tree edge of G. Each non-tree edge of G is represented by at least one node of G'. To construct G', one starts with an empty graph. An out-tree $H(v)$ is added to G' for each vertex v of the input graph. The nodes of $H(v)$ represent the non-tree edges of G whose tails are on the path from v to t in T. The nodes of $H(v)$ are heap-ordered according to the weights of the non-tree edges that they represent, using the weight function δ. (A tree is heap-ordered if the weight associated with each node is no larger than the weights associated with the node's children.) The tree $H(v)$ need not be balanced. In Eppstein's actual construction, subtrees that appear more than once in G' can be collapsed into a single copy of the subtree. This does not affect the following description, but allows G' to be constructed quickly. For each node α representing the non-tree edge (u, v) of G, a new edge, called a cross-edge, is added to G' connecting α to the root of $H(v)$. Finally, a new node, s' is added to G' which is connected to the root of $H(s)$ by an edge.

Figure 3.2 shows the graph G' for the graph G of Figure 3.1(b). The weights on the edges of G' will be explained below. The name and weight of the edge represented by each node are shown in the diagram. Cross-edges are shown as dotted lines. To simplify the diagram, some identical heap-ordered trees have been collapsed into a single tree. The tree $H(t)$ does not contain any nodes and is therefore not shown.

Consider any path p' in G' starting at s'. Suppose the tails of the cross-edges that appear in p', taken in order along p', represent the non-tree edges $(u_1, v_1), \ldots, (u_{l-1}, v_{l-1})$ of G, and that the last node of p' represents the non-tree edge (u_l, v_l). One can define a corresponding s-t path p in G that

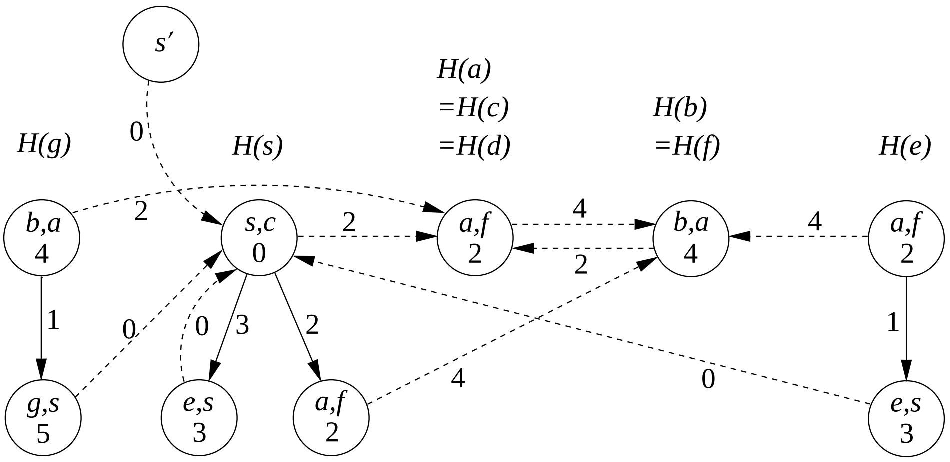


Figure 3.2: The graph G' constructed for the graph G shown in Figure 3.1(b)

has $< (u_1, v_1), \ldots, (u_l, v_l) >$ as its sidetracks sequence. (If p' consists of the single node s', then p is the path from s to t in T.) Intuitively, one can imagine travelling along the path p' to determine the corresponding s-t path, p, in G. The initial segment of the path p' is inside the tree $H(s)$. This tree contains a node for each possible choice of the first non-tree edge of p. The final node of this initial segment of p' determines which of the possible edges to use as the first non-tree edge in p. Once the first non-tree edge, (u_1, v_1), of p is found in $H(s)$, the path p' jumps to the root of the tree $H(v_1)$. The tree $H(v_1)$ contains a node for each possible choice of the second non-tree edge of p. This process is repeated to determine each of the non-tree edges of p in turn. It is not difficult to prove that the mapping $p' \mapsto p$ is a bijection from paths of G that begin at s' onto the paths of G that begin at s and end at t.

The edges of G' can now be assigned weights so that the correspondence $p' \mapsto p$ preserves path weights. Each cross-edge of G' that goes into node α is assigned the weight of the non-tree edge of G represented by α. The single edge out of s' is assigned the weight of the non-tree edge represented by the root of $H(s)$. The weight of any other edge (α, β) of G' is equal to the difference between the weights of the non-tree edges of G represented by the nodes α and β. For example, in Figure 3.2, the weight of the solid edge in $H(g)$ is $\delta(g, s) - \delta(b, a) = 5 - 4 = 1$. The weight of the segment of p' inside $H(v_i)$, together with the weight of the cross-edge which was used to jump into $H(v_i)$, is equal to the weight of the edge (u_{i+1}, v_{i+1}) of G. This ensures that the weight of p in G is the same as the weight of p' in G'.

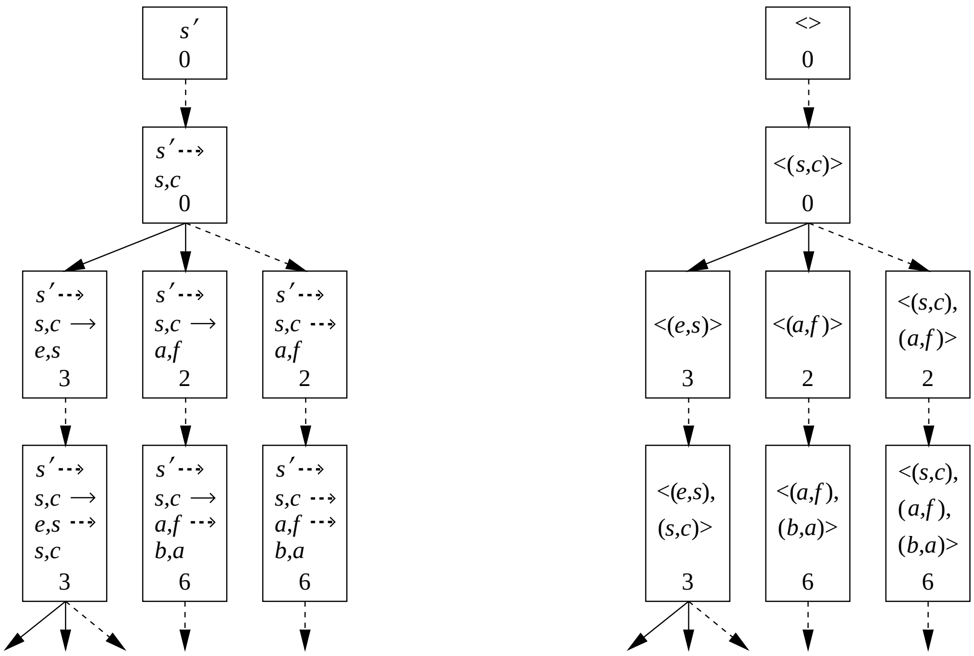
For example, the path $p' = s' \dashrightarrow (s, c) \rightarrow (a, f) \dashrightarrow (b, a) \dashrightarrow (a, f)$ in the graph G' of Figure 3.2 corresponds to the path $p = s \longrightarrow e \longrightarrow a \dashrightarrow f \longrightarrow b \dashrightarrow a \dashrightarrow f \longrightarrow b \longrightarrow t$ in the graph G shown in Figure 3.1(b), since sidetracks$(p) =< (a, f), (b, a), (a, f) >$. Both p' and p have weight 8.

Eppstein's algorithm constructs the trees $H(v)$ to have bounded outdegree, so that the graph G'

also has bounded outdegree. By allowing the trees to share common subtrees, the construction of G' can be carried out in $O(m + n \log n)$ time. (See [13] for details.)

The problem of finding the k shortest paths s-t paths in G has now been reduced to a problem of finding the k shortest paths that begin at node s' in G'. The latter problem can be solved by Frederickson's $O(k)$ algorithm that selects the k smallest elements in a heap-ordered out-tree F with bounded outdegree [18]. The nodes of F represent paths of G' that start at s'. The root of F represents the trivial path with only one node, s'. The node of F that represents path p' has one child for each path that can be obtained by adding one more node to the end of p'.

Figure 3.3(a) shows the top four levels of the tree F obtained from the graph G' of Figure 3.2. Each node of F is labelled with a path p' of G', and the path's weight. Figure 3.3(b) shows the sidetracks sequence of the path p in G which corresponds to each path p' that appears in Figure 3.3(a). In this example, the four shortest paths from s to t in G are defined by the sidetracks sequences $<>, <(s,c)>, <(a,f)>$ and $<(s,c),(a,f)>$. The actual paths are $s \longrightarrow e \longrightarrow a \longrightarrow t$, $s \dashrightarrow c \longrightarrow a \longrightarrow t$, $s \longrightarrow e \longrightarrow a \dashrightarrow f \longrightarrow b \longrightarrow t$, and $s \dashrightarrow c \longrightarrow a \dashrightarrow f \longrightarrow b \longrightarrow t$,which have weights 0, 0, 2 and 2, respectively (according to the weight function δ).



(a) The heap-ordered tree F, whose nodes represent paths of G'

(b) The sidetracks sequences of the corresponding paths of G

Figure 3.3:

The tree F may be of infinite height if the graph G' contains a cycle, but only G', and not F, must be explicitly constructed before applying Frederickson's algorithm. The value associated with a node of F is just the weight of the path that it represents. This allows the k shortest paths of G' that start at s' to be computed in $O(k)$ time. Once these shortest paths are known, one has an implicit representation of the k shortest s-t paths in G. Using this implicit representation, the paths themselves can be printed out in time proportional to the length of the output. The sidetracks sequence of each path in G can be obtained from the corresponding path in G' using the mapping $p' \mapsto p$. The path itself can then be obtained by traversing its sidetracks sequence, filling in any tree edges that appear in the path by following the edges of T.

The total time used to compute the implicit representation of the k shortest paths algorithm is $O(m + n \log n + k)$, since T and G' can be constructed in $O(m + n \log n)$ time, and Frederickson's selection algorithm runs in $O(k)$ time. Eppstein's algorithm can also be used to find the k shortest paths to a fixed vertex t from every vertex in the graph G. The construction of G' must be done only once, except for the placement of the node s'. Implicit representations of the k shortest paths to t from each vertex s can therefore be computed in $O(m+n \log n+nk)$ time, and the paths themselves can be extracted from the implicit representation in time proportional to the length of the output.

The following proposition shows that the problem of selection in a heap-ordered tree cannot be solved in $O(\text{polylog}(k))$ time, using a polynomial amount of work. This suggests that it would be difficult to implement Eppstein's algorithm directly on a PRAM. A different approach for a parallel algorithm will be described in the next section.

Proposition 3.2 *If at most P nodes of a binary heap-ordered tree F can be examined in each parallel step, $\Omega(k/\log P)$ steps are required to find the k smallest elements of F.*

Proof: Consider the case where F is a complete binary tree of height k. For a fixed value of P, let $S(k)$ be the number of parallel steps required (in the worst case) to find the kth element of F if at most P nodes can be examined during each step.

Consider the first parallel step. There are at least $P + 1$ nodes at level $l = \lceil \log(P + 1) \rceil$ of F, so there is at least one subtree, F', rooted at level l which contains none of the nodes that were examined in the first step. Suppose all of the nodes on the path from the root of F to the root of F' have a rank equal to their depth in F, and other nodes not in F' have ranks greater than k. Then, the kth smallest element of F is the $(k - l)$th element of F'. In order to compute the kth element of F, it would then be necessary to compute the $(k - l)$th element of F'. So, $S(k) \geq S(k - l) + 1$, for $k > l$, and hence $S(k) \geq k/l - 1 = \Omega(k/\log P)$. □

It follows from the proposition that a PRAM algorithm for the heap-ordered tree selection problem which uses $O(k^q)$ processors (where q is any constant) will require $\Omega(k/\log k)$ time.

3.3 A Parallel Algorithm to Compute the k Shortest Paths

In this section, an algorithm is developed to compute the k shortest paths from each vertex of a directed graph G to a fixed vertex t, given a shortest path tree T that describes the shortest path from any vertex to t. The algorithm runs on a concurrent-read, exclusive-write PRAM in $O(\log^3 k \log^* k + \log n \log\log k + \log d \log^* d)$ time and the amount of work done is $O(m + nk\log^2 k)$, where d is the maximum outdegree of any node in the graph. It computes an implicit representation of the paths, from which information about the paths can be extracted efficiently using the techniques to be described in Section 3.3.4. It is assumed that the graph G is initially stored as adjacency arrays: for each vertex v, there is an array containing one entry for each edge whose tail is at v. The array entry stores the head of the edge and its weight. (When designing sequential shortest path algorithms, it is usually assumed that the edges are given as adjacency lists, but traversal of such a list requires $\Omega(n)$ time in the worst case, and this would make polylogarithmic-time parallel algorithms impossible).

In the algorithm described in this section, it is assumed that the tree T is given as part of the input. T can be computed using a transitive closure computation (see Section 3.1). As described in Section 3.2, the given edge weights can be replaced by the weight function δ, satisfying Lemma 3.1.

The problem will be solved in stages. During the ith stage, the k shortest paths to t from each vertex, v, that have at most 2^i non-tree edges are computed. The following lemma shows that $\lceil \log k \rceil$ stages will be sufficient.

Lemma 3.3 *There is a solution to the k shortest paths problem in which all of the paths have at most $k-1$ non-tree edges.*

Proof: Suppose there is a solution set that contains a path p with at least k non-tree edges. Consider the paths from v to t whose sequences of non-tree edges are prefixes of the sequence sidetracks(p), where the prefixes are of length $0, 1, 2, \ldots, k-1$. There are k such paths, and each one has weight no greater than p, since δ is non-negative. So, the set of these paths is a correct solution to the problem, and all of them have at most $k-1$ non-tree edges. □

3.3.1 The Data Structure Used by the Algorithm

Let A_v^i be an array that will store the k shortest paths from v to t that have at most 2^i non-tree edges. If there are fewer than k such paths, some of the entries in the array will be nil. Elements of the array A_v^i will be formed by concatenating the sequences of non-tree edges of two paths with at most 2^{i-1} non-tree edges each.

When representing a path p in one of these arrays, only the following information about the path will actually be stored

- the weight of the path (with respect to the weight function δ)
- the number of non-tree edges along the path

- the head of the last non-tree edge on the path (this could be nil if all edges along the path are in T)
- information that describes how the path was formed by concatenating previously computed paths. For each path in A_v^i, the data structure will contain pointers to the two paths whose lists of sidetracks were concatenated to form p. (For paths in A_v^0 which have one non-tree edge, the non-tree edge is stored instead.)

Stage i of the algorithm will compute A_v^i using the arrays computed in the previous stage.

3.3.2 Computing A_v^0

The first entry of A_v^0 will be the path from v to t in T. It has weight 0 and contains no non-tree edges. The rest of the paths in A_v^0 will each contain exactly one non-tree edge, so the tail of each of these edges must lie on the path from v to t in the tree T. Thus, A_v^0 can be filled in by finding the k shortest non-tree edges (with respect to δ) whose tails are on the path from v to t in T.

To select the k smallest edges whose tails are at vertex v, degree$(v)/(\log d \log^* d)$ processors are assigned to each vertex v, where d is the maximum outdegree of any vertex in the graph. The allocation can be performed in $O(\log n)$ time using $O(n)$ work (see Section 1.3.1). The k smallest edges out of each vertex can then be chosen in $O(\log d \log^* d)$ time and $O(\sum_{v \in V} \text{outdegree}(v)) = O(m)$ work, using Cole's selection algorithm [9]. Cole's parallel merge sort [10] can be used to sort the k smallest edges out of each vertex in $O(\log k)$ time using nk processors.

Tree contraction is used to compute the array of the k shortest edges whose tails are on the path from each vertex v to the destination t. The tree contraction is performed as described in Section 1.3.5 where it is used to compute the minimum weight ancestor of each node in a node-weighted tree, except each primitive operation computes the smallest k elements in a pair of sorted arrays of k edges instead of finding the minimum of two node weights. This primitive operation is performed by merging the two sorted arrays, and then taking the first half of the resulting array. Ties between edge weights can be broken according to some arbitrary lexicographic order on the edges. Each merge step can be performed in $O(\log \log k)$ time and $O(k)$ work using the merging algorithm of Borodin and Hopcroft (see Section 1.3.2), so the tree contraction takes $O(\log n \log \log k)$ time and $O(nk)$ work in total.

The total time for stage 0 of the algorithm is $O(\log d \log^* d + \log k + \log n \log \log k)$, and $O(m + nk \log k)$ work is performed.

3.3.3 Computing A_v^i for $i > 0$

The candidate paths for inclusion in A_v^i (for $i > 0$) are those paths whose sidetracks sequence is obtained by concatenating sidetracks(p_1) and sidetracks(p_2), where p_1 is a path in the array A_v^{i-1}, p_2

is a path in A_w^{i-1}, and w is the head of the last non-tree edge of p_1. Any sidetracks sequence formed in this way represents a legal path, since there is a path in T from the head of the last non-tree edge of p_1 to the tail of the first non-tree edge in p_2. For example, in the graph of Figure 3.1, combining the paths $p_1 = g \dashrightarrow s \dashrightarrow c \longrightarrow a \longrightarrow t$ and $p_2 = c \longrightarrow a \dashrightarrow f \longrightarrow b \longrightarrow t$ produces the path $g \dashrightarrow s \dashrightarrow c \longrightarrow a \dashrightarrow f \longrightarrow b \longrightarrow t$ (where dashed arrows indicate non-tree edges).

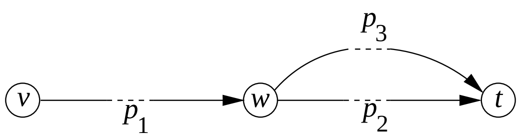
Paths p_1 containing l non-tree edges will be combined only with paths p_2 which have either l or $l-1$ non-tree edges. This ensures that each path considered is distinct, since any sequence of non-tree edges can be divided into such a pair p_1, p_2 in exactly one way. In fact, it is sufficient to consider only those pairs which, when concatenated, yield a path that has more than 2^{i-1} non-tree edges, since the k shortest paths with at most 2^{i-1} non-tree edges are already known.

From among these candidates, together with the paths in A_v^{i-1}, the shortest k are chosen to be the entries in A_v^i. In order to prove that this algorithm solves the problem correctly, it is helpful to define a total ordering $\prec_v$ on the paths from v to t so that an invariant describing the contents of A_v^i can be stated precisely. Paths are ordered by weight, and ties are broken in favour of the path with fewer non-tree edges. If two paths from v to t have the same weight and the same number of non-tree edges, then the lexicographic order of the sidetracks sequences is used to break the tie. This total ordering is used to select the k shortest paths for inclusion in A_v^i.

Lemma 3.4 *The paths in A_v^i are the k smallest paths (with respect to the order $\prec_v$) with at most 2^i non-tree edges. (If there are fewer than k such paths, all of them appear in A_v^i.)*

Proof: The previous section described the computation of A_v^0 that ensures that the invariant is true for $i = 0$.

Suppose that the invariant is true for $i-1$. Let p be any path from v to t with l_p non-tree edges, where $2^{i-1} < l_p \le 2^i$. Let p_1 be the portion of p up to and including the $\lceil l_p/2 \rceil$ th non-tree edge. Let w be the last vertex on p_1, and let p_2 be the remaining portion of p from w to t. Let p_3 be the path from w to t consisting only of tree edges. The paths $p_1 p_3$ and p_2 each have at most 2^{i-1} non-tree edges.



Suppose p is one of the k smallest paths (with respect to $\prec_v$) from v to t with at most 2^i non-tree edges. To prove the invariant, it is sufficient to show that $p_1 p_3$ appears in A_v^{i-1} and p_2 appears in A_w^{i-1}, since p will then be among the paths considered for inclusion in A_v^i.

Suppose $p_1 p_3$ does not appear in A_v^{i-1}. By the inductive hypothesis, there are k paths from v to t that are smaller than $p_1 p_3$ (with respect to $\prec_v$), each with at most 2^{i-1} non-tree edges. But

$p_1p_3 \preccurlyeq_v p_1p_2 = p$, since p_3 has no non-tree edges. This contradicts the fact that p is among the k smallest paths from v to t with at most 2^i non-tree edges. So, p_1p_3 must be in A_v^{i-1}.

Now suppose p_2 does not appear in A_w^{i-1}. Then there are k paths from w to t, each with at most 2^{i-1} non-tree edges, satisfying $r_1 \prec_w r_2 \prec_w \ldots \prec_w r_k \prec_w p_2$. But this implies that $p_1r_1 \prec_v p_1r_2 \prec_v \ldots \prec_v p_1r_k \prec_v p_1p_2 = p$ (by the definition of the orders $\prec_v$ and $\prec_w$). This is again a contradiction, so p_2 must be in A_w^{i-1}. □

It follows from Lemma 3.3 that $A_v^{\lceil \log k \rceil}$ will contain the k shortest paths from v to t. In the rest of this section, the implementation of the computation of A_v^i on a PRAM is described, and the analysis of the performance of the algorithm is completed.

Using the notation introduced in the previous proof, the weight of the path formed by combining p_1 and p_2 is just the sum of the weights of p_1p_3 and p_2, since the weight of each tree edge is 0. Suppose that the paths of A_v^{i-1} are sorted according to the number of non-tree edges in the path, with ties broken in accordance with the ordering $\prec_v$. Then, the candidates for inclusion in A_v^i can be found by combining each path p_1 of length l in A_v^{i-1} with the paths in two contiguous subarrays of A_w^{i-1}: the portions containing paths with l or $l-1$ non-tree edges. Each subarray will already be sorted by $\prec_w$, so the paths that result from combining the path p_1 with the elements of the subarray will be sorted according to $\prec_v$. Thus, the problem of picking the shortest k paths is now equivalent to selecting the k shortest paths from a set of $2k+1$ sorted arrays of paths, each with at most k elements (one of the arrays is A_v^{i-1}, and there are two arrays for each path p_1 in A_v^{i-1} whose elements are obtained by combining p_1 with other paths). If the arrays are considered to be the columns of a $k \times (2k+1)$ matrix, the algorithm of Section 2.4 will solve this problem in $O(\log^2 k \log^* k)$ time and $O(k)$ work. The matrix is not explicitly computed; entries are computed as they are accessed by the matrix selection algorithm. If ties between two paths with the same weight that appear in different columns are broken in favour of the element in the leftmost column, the elements selected from the matrix will be the k smallest with respect to the ordering $\prec_v$.

The k elements of A_v^i can then be sorted in $O(\log k)$ time and $O(k \log k)$ work using Cole's parallel merge sort [10]. The ith stage of the algorithm that computes the arrays A_v^i (for all vertices v) therefore uses $O(log^2 k \log^* k)$ time and $O(nk \log k)$ work.

Once the $\lceil \log k \rceil$ stages of the algorithm are complete, the k shortest paths are stored implicitly in the data structure. Summing up (see Table 3.1) demonstrates the following theorem.

Theorem 3.5 *Let G be a directed graph with n vertices, m edges. Let d be the maximum outdegree of any vertex. Given the tree of shortest paths rooted at vertex t, an implicit representation of the k shortest paths to t from every vertex v can be computed on a CREW PRAM using $O(\log^3 k \log^* k + \log n \log\log k + \log d \log^* d)$ time and $O(m + nk \log^2 k)$ work.*

A method for retrieving information about the paths from the implicit representation is described in Section 3.3.4.

		Time	Work
Stage 0	Selecting and sorting k smallest edges out of each vertex	$O(\log d \log^* d + \log k)$	$O(m + nk \log k)$
	Tree contraction to compute A_v^0	$O(\log n \log\log k)$	$O(nk)$
Stage i $(1 \le i \le \lceil \log k \rceil)$	Selecting k smallest elements from a $k \times (2k+1)$ matrix	$O(\log^2 k \log^* k)$	$O(nk)$
	Sorting each A_v^i array	$O(\log k)$	$O(nk \log k)$
	Total	$O(\log d \log^* d +$ $\log n \log\log k +$ $\log^3 k \log^* k)$	$O(m + nk \log^2 k)$

Table 3.1: Performance of k shortest paths algorithm on CREW PRAM

The parallel algorithm developed here will also work on directed *multigraphs*, where there can be more than one edge from one vertex to another. The performance bounds proven above apply unchanged. If some of the edge weights are negative, the algorithm will still work, provided there is no cycle in the graph whose total weight is negative. The proof of Lemma 3.1 applies unchanged to this case. Therefore, the reduction used to reduce a general problem instance to one with non-negative edge weights (defined by δ), where the distance from any vertex to vertex t is 0, also applies to graphs with negative edge weights that have no negative cycles. The algorithm may also be used to solve the single-source version of the problem, where the task is to find the k shortest paths from a given vertex s to all other vertices, simply by reversing the direction of the edges in the input graph.

3.3.4 Extracting Information From the Implicit Representation

Some properties of the paths stored in the data structure can be obtained without computing an explicit listing of the edges in the path. For example, the number of edges that appear in each path can be computed. First, the depth of each node in T is computed using tree contraction. The algorithm is similar to the computation of minimum ancestors described in Section 1.3.5, except that the minimization operations are replaced by additions, and each node in the tree is assigned a value of 1, except the root, which has value 0. The data structure described in Section 3.3.1 can be augmented to include an extra field, num, that stores the number of edges in the path up to and including the last non-tree edge. It is easy to compute the value of num for paths found during stage 0 of the algorithm, since the number of edges in any path consisting entirely of tree edges is the difference between the depths of the endpoints. When a path is formed by concatenating the sidetracks sequences of two paths, the num field can be computed by adding the num fields of the two concatenated paths. When the final results are obtained, the number of edges in any of the k shortest paths can be found by adding the value of num to the depth in T of the head of the last non-tree edge on the path. The preprocessing of T to compute depths of nodes uses $O(\log n)$ time and $O(n)$ work, which is small compared to the resources required by the k shortest paths algorithm. The additional computations to update the num fields do not affect the asymptotic bounds on the performance of the algorithm either, since they involve a constant amount of work each time two paths are combined.

In fact, this technique can be used to compute a variety of properties of the paths without explicitly listing the edges in the path, as observed by Eppstein [13]. Suppose each edge in the graph is assigned a value from a semigroup, which has an associative binary multiplication operator defined on it, and the value of a path is defined as the product of the values of the edges along the path. If the multiplication operation can be evaluated in constant time by a single processor, the values of the k shortest paths can be found in the same way as the number of edges in the paths can be computed, where the value of each edge was one, and the binary operator was ordinary integer addition. As a further example, suppose that each edge (u, v) of the graph is assigned a capacity $c(u, v)$, and the capacity of a path is defined to be the minimum capacity of any edge along the path. The algorithm can be modified to compute the capacity of each of the k shortest paths, again without increasing the asymptotic bounds on the performance of the algorithm. (Here, minimization is the associative operator.)

An explicit listing of the edges on the k shortest paths from a vertex v to the vertex t can also be extracted from the data structure.

First, some preprocessing is done to the shortest path tree, T. Each node locates a pointer to its ancestor 2^i levels above it, for $i = 1, 2, \ldots, \log n$ (some of these pointers may be nil). This is done so that portions of the k shortest paths that are made up exclusively of tree edges can be traversed quickly. The computation is done using the technique of pointer jumping, in $\log n$ stages. During the ith stage, each node simultaneously locates its 2^ith ancestor by finding the 2^{i-1}th ancestor of its 2^{i-1}th ancestor, which was computed in the previous stage. This computation uses $O(\log n)$ time and $O(n \log n)$ work in total.

Suppose that the jth shortest path from v to t contains l_j edges. The k shortest paths from v to t can then be explicitly stored one after another in an array P of length $L = \sum_{j=1}^{k} l_j$. The starting location of each path in the array P can be found by performing a prefix sum on $l_1, \ldots, l_k$.

The pointer structure that represents the jth shortest path from v to t can be thought of as a binary out-tree of height at most $\lceil \log k \rceil$. The root of the tree is the jth entry of $A_v^{\lceil \log k \rceil}$. Each internal node of the tree has a pair of pointers to the two subpaths that were combined to create the longer path that it represents, and the leaves of the tree represent the non-tree edges along the path. Assume that each entry in the arrays A_v^i has a field, num, associated with it that stores the number of edges in the path that it represents, up to and including the last non-tree edge, as described above.

Suppose $L/\log(kn)$ processors are available. Each processor is assigned the task of filling in a block of the output array P of length $\log(kn)$. The processor follows the out-tree edges in the pointer structure representing the path from the root down to the appropriate leaf, using the num fields of the nodes along the path to decide whether to go to the left or right child. When it reaches a leaf, the processor begins filling in entries of P using the following sequential algorithm. The portion of P that the processor must compute is made up of segments of branches of T, separated by non-tree edges. The processor can perform a linear traversal of the branches of T, copying the edges into P one by one. Whenever the processor reaches the end of a segment of tree edges, it traverses to the next leaf in the out-tree pointer structure representing the path. That next leaf represents the next non-tree edge on the path. Once the non-tree edge has been entered into P, the processor can again start copying a segment of a branch of T into P by doing a linear traversal of the branch. (If the first edge that the algorithm is required to enter into P is in the middle of a segment of tree edges in the path, the processor can jump to the correct point in T in $O(\log n)$ time using the ancestor pointers computed during the preprocessing of T). If a processor finishes entering one of the k shortest paths into P, it starts working on the next one.

The time spent traversing and copying branches of T is at most $O(\log(kn))$. Each processor traverses from leaf to leaf of the out-tree rooted in $A_v^{\lceil \log k \rceil}$ using the standard sequential left-right traversal of the leaves of a tree. The total amortized time for this traversal is proportional to the depth of the tree plus the number of leaves visited, which is $O(\log n + \log(kn)) = O(\log(kn))$. So, the total running time to compute the output array P, including the time to preprocess T, is $O(\log n + \log(kn)) = O(\log k + \log n)$, and the total work performed is $O(n \log n + L)$.

The k shortest paths to t from every vertex v can be extracted, in $O(\log k + \log n)$ time using $O(n \log n + L)$ work (where L is the total length of the output), since the preprocessing of T need only be done once.

Chapter 4

Applications

Many optimization problems can be solved by constructing a graph and then finding shortest paths in the graph. In this chapter, two examples of this method are described. The k shortest paths algorithm of the previous chapter is applied to obtain parallel algorithms for the k quickest paths problem and for the list Viterbi decoding problem.

4.1 Quickest Paths

The quickest path problem is a generalization of the shortest path problem. It was introduced by Y. Chen and Y. Chin [8] to find the quickest way of transmitting data through a computer network. The network is represented as a directed graph, $G = (V, E)$ with n vertices and m edges. Each edge $(u, v) \in E$ has a capacity, $c(u, v)$, measured in bits per second, and a latency, $l(u, v)$, measured in seconds. The transmission of σ bits of data along the edge (u, v) requires $l(u, v) + \sigma/c(u, v)$ seconds; $l(u, v)$ seconds after the transmission begins at u, data begins arriving at v, and continues arriving at a rate of $c(u, v)$ bits per second thereafter.

The capacity of a path, $P = v_0 \to v_1 \to \ldots \to v_L$, is defined to be $c(P) = \min_{1 \le i \le L} c(v_{i-1}, v_i)$, since the edge with the smallest capacity constrains the rate at which data can be transmitted along the path. The time required to send σ bits of data along a path P is $\sigma/c(P) + \sum_{i=1}^{L} l(v_{i-1}, v_i)$. The quickest path between two vertices (for a given value of σ) is the path that minimizes this quantity. When all edges have the same capacity, the problem of finding the quickest path reduces to the problem of finding the shortest path, using the latencies as the edge weights.

In the graph of Figure 4.1, the first number labelling each edge is its capacity and the second is its latency. The path $s \to u \to v$ requires $\sigma/2 + 2$ seconds to send σ bits of data. The path $s \to v$ requires $\sigma/3 + 3$ seconds, and is the quickest path from s to v when σ is at least 6. The quickest path from s to t is $s \to u \to v \to t$ for all values of σ. Subpaths of quickest paths need not be quickest

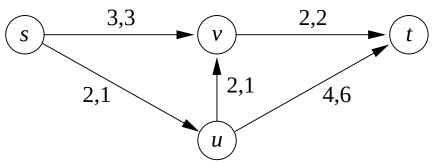


Figure 4.1: An example graph with capacities and latencies

paths themselves. For example, when $\sigma > 6$, the quickest path from s to t is $s \rightarrow u \rightarrow v \rightarrow t$ even though the quickest path from s to v is $s \rightarrow v$.

4.1.1 Sequential Algorithms

Y. Chen and Y. Chin [8] give a sequential algorithm to find the quickest path from vertex s to vertex t in $O(m^2 + mn \log n)$ time. If there are only r distinct capacities assigned to the edges of the graph, the running time of their algorithm reduces to $O(rm + rn \log n)$. A similar algorithm with the same worst-case running time is described by Rosen, Sun and Xue [37]. Kagaris *et al.* [28] give an algorithm which has the same worst-case running time for the general problem, but runs faster if the graph can be decomposed into a small number of subgraphs (called hammocks) satisfying certain properties. Kagaris *et al.* also study a dynamic version of the quickest paths problem. Y.-C. Hung and G.-H. Chen describe an $O(mn^2)$ algorithm to find the quickest path between all pairs of vertices [24] and also consider distributed algorithms in an asynchronous, message-passing model of computation [25].

The sequential algorithm described here, which is a slightly modified version of Chen and Chin's algorithm, uses a transformation of the graph to reduce the quickest path problem to a shortest path problem. Let the distinct edge capacities be denoted by $c_1 < c_2 < \ldots < c_r$. A new graph, $G' = (V', E')$ is defined containing r partial copies of G, called levels. G' will be defined so that each path in G with capacity c_i has a corresponding path in G' that begins in the bottom level, r, and ends in level i. Weights will be assigned to the edges of G' so that the time required to transmit data along a path in G is the same as the weight of the corresponding path in G'.

Each level of G' has a copy of every vertex of G. The copy of vertex v in level i will be denoted v^i. The ith level of G' includes copies of the edges of G that have capacity at least c_i. Edges are also added to connect levels of G': if (u, v) is an edge of G with capacity c_i, an edge (u^j, v^i) is added between the copies of u and v to level i from each higher-numbered level $j > i$. Finally, a new destination vertex t' is introduced, and each copy of the original destination, t, is connected to it by an edge.

For example, Figure 4.2 shows the graph G' obtained from the input graph G of Figure 4.1. The edges between levels of the graph are shown as dotted lines.

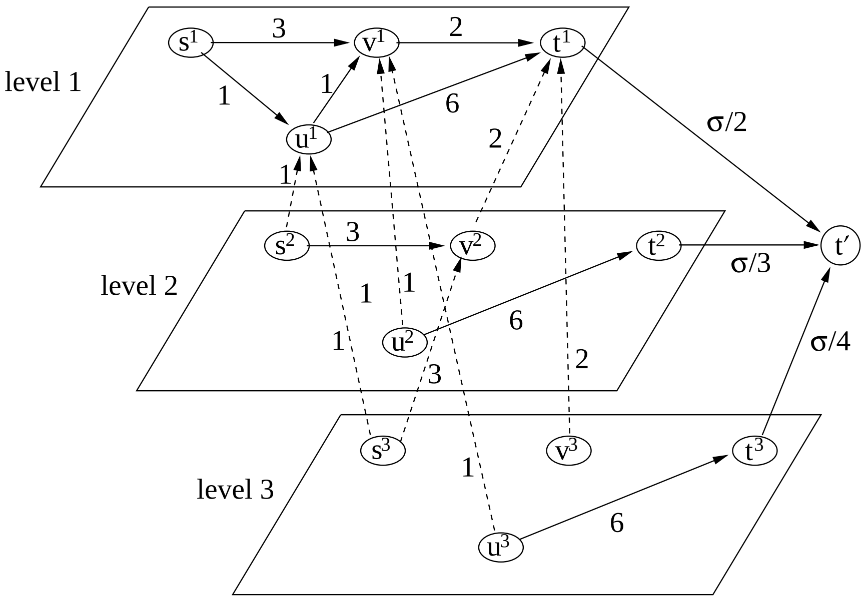


Figure 4.2: Converting the quickest path problem for the graph of Figure 4.1 into a shortest path problem

For any path p of G starting at vertex s, one can define a corresponding path p' in G' which starts at s^r and visits copies of each vertex along the path p. The path p' stays in the highest-numbered level of G' possible, only jumping to another level, i, when it is forced to do so because an edge of capacity c_i appears in p. This ensures that p' ends in level i whenever p has capacity c_i. Formally, if $p = s \to v_1 \to v_2 \to \ldots \to v_L$ in G, there is a corresponding path $p' = s^r \to v_1^{i_1} \to v_2^{i_2} \to \ldots \to v_L^{i_L}$ in G', where c_{i_j} is the capacity of the prefix $s \to v_1 \to \ldots \to v_j$ of p.

The mapping $p \mapsto p'$ is clearly injective. One can also show by mathematical induction on L that there exists a path p in G that maps to a given path p' in G' of the form $s^r \to v_1^{i_1} \to \ldots \to v_L^{i_L}$. If $L = 1$, the edge $(s^r, v_1^{i_1})$ must appear in G'. By the definition of G', the edge (s, v_1) must appear in G and have capacity c_{i_1}, and the path $p = s \to v_1$ maps to p'. Now suppose that the claim holds for paths with $L-1$ edges. Consider a path p' with L edges. By the induction hypothesis, there is a path $\tilde{p} = s \to v_1 \to \ldots \to v_{L-1}$ in G that maps to $s^r \to v_1^{i_1} \to \ldots \to v_{L-1}^{i_{L-1}}$. By the definition of G', i_L must be no bigger than i_{L-1}. If $i_L = i_{L-1}$, then (v_{L-1}, v_L) must be an edge of G with capacity at least c_{i_L}. If $i_L < i_{L-1}$, then (v_{L-1}, v_L) must be an edge of G with capacity exactly c_{i_L}. In either case, $p = \tilde{p} \to v_L$ is a path of G with capacity i_L that gets mapped to p'.

Thus, the mapping $p \mapsto p'$ is a bijection from paths of G from s to v onto paths of G' that start at s^r and end at v^i for some i. Consequently, the mapping $p \mapsto (p' \to t')$ is a bijection from s-t paths of G to paths of G' that start at s^r and end at t'.

Edges in the graph G' are not labelled with a capacity and a latency; instead, they are assigned a weight, which depends on the input σ, as follows: $w_\sigma(u^j, v^i) = l(u,v), w_\sigma(t^i, t') = \sigma/c_i$. For any s-t path p in G, the time to transmit σ bits along p is $w_\sigma(p' \to t')$, since $l(p) = w(p')$ and the weight of the final edge of $p' \to t'$ is $\sigma/c(p)$. So, the problem of finding the k quickest paths from s to t in G has been reduced to finding the k shortest paths from s^r to t' in the graph G'.

Since G' has $rn+1$ vertices and $r(m+1)$ edges, the required shortest path in G' can be found, using Dijkstra's algorithm, in $O(rm + rn\log(rn)) = O(rm + rn\log n)$ time [21].

Finding the k Quickest Paths

The problem of finding the k quickest paths from vertex s to vertex t for the transmission of σ bits of data has also been studied. A solution to this problem is a set $\mathcal{P}$ of paths from s to t such that the time required to transmit σ bits of data along any path in $\mathcal{P}$ is less than the time required along any s-t path not in $\mathcal{P}$. If fewer than k s-t paths exist, $\mathcal{P}$ should contain all of them.

Y. Chen [6] gives a sequential algorithm to find the k quickest paths between two vertices that runs in $O(m^2 + (m+k)n\log n + k^{3/2}\log k)$ time. This algorithm computes an implicit representation of the paths, from which an explicit listing can be extracted in time proportional to the length of the output. The candidate paths are grouped according to their capacities. The quickest path from each group is placed in a priority queue. The algorithm repeatedly removes and outputs the quickest path in the priority queue, and replaces it by the next quickest path with the same capacity. It is possible to improve Chen's algorithm to run in $O(rm + rn\log n + k\log k)$ time by examining fewer candidate paths and exploiting the fact that many edges may have the same capacity.

A faster, more straightforward algorithm to compute the k quickest paths from s to t can be obtained by reducing the problem to the k shortest paths problem, using the graph transformation $G \mapsto G'$ described in the previous section. The bijective mapping $p \mapsto (p' \to t')$ from s-t paths of G onto the s^r-t' paths of G' has the property that the time to transmit σ bits of data along p is equal to the weight $w_\sigma(p' \to t')$. Thus, the required quickest paths can be found by computing the k shortest s^r-t' paths in G' using Eppstein's algorithm (see Section 3.1). This establishes the following theorem.

Theorem 4.1 *Let s and t be vertices of an n-node graph with m edges that are labelled with latencies and r distinct capacities. There is a sequential algorithm that computes an implicit representation of the k quickest paths from s to t in $O(rm + rn\log n + k)$ time. The actual paths can then be printed out in time proportional to the length of the output.*

If $k = O(rm + rn\log n)$, the computation of the k quickest paths has the same time complexity as Chen and Chin's algorithm for finding the single quickest path between a pair of vertices. Finding implicit representations of the k quickest paths from every vertex to a fixed vertex t can be accomplished in $O(rm + rn\log n + kn)$ time using Eppstein's algorithm. Repeating this n times, once for each possible destination vertex t, yields an $O(rmn + rn^2 \log n + kn^2)$ algorithm for the all-pairs k quickest paths problem. If $k = \omega\left(\sqrt{\frac{r}{n} + \frac{r \log n}{m}}\right)$, this is better than the best known algorithm for the all-pairs version of the problem, due to G.-H. Chen and Y.-C. Hung [5], which runs in $O(k^2mn^2)$ time. In fact, Chen and Hung's algorithm can be improved to run in $O(kmn^2 \log k)$ time by replacing an $O(k^2)$ computation of the k smallest elements of a $k \times k$ Cartesian sum matrix (see Section 2.2) by the faster algorithm of Mirzaian and Arjomandi [33], but the algorithm presented here is still better when $k \log k = \omega\left(\sqrt{\frac{r}{n} + \frac{r \log n}{m}}\right)$.

The problem of finding the k quickest *simple* paths has also been studied ([7], [37]). This version of the problem appears to be significantly harder, and the best previously known algorithm, presented by Y. Chen [7], runs in $O(krn^3 + krn \log k)$ time, or $O(krn^2 + krn \log k)$ time for an undirected graph. An improved implementation which decreases the running time to $O(krn^3)$ (or $O(krn^2)$ for undirected graphs) is described here.

Let $G_i = (V, E_i)$ be the subgraph of G that contains only those edges with capacity greater than or equal to c_i (for $1 \leq i \leq r$). (G_i is just the ith level of the graph G' described above). The capacity of any path in G_i is at least c_i. To compute the k quickest simple s-t paths in G, one may first compute the set $\mathcal{P}_i$ of the k shortest simple paths (with respect to latency) in G_i and then choose the quickest k paths in $\cup_{i=1}^{r} \mathcal{P}_i$. This will yield a valid solution to the problem, since each path in $\mathcal{P}_i$ will have a transmission time that is no bigger than that of any path with capacity c_i that does not appear in $\mathcal{P}_i$. To avoid duplicate paths in the output, any of the paths in $\mathcal{P}_i$ which have capacity $c_j > c_i$ will be discarded before the k quickest paths are selected. If such a discarded path is indeed among the quickest k simple paths, it will also appear in $\mathcal{P}_j$. This step will ensure that each candidate path with capacity c_j appears only in $\mathcal{P}_j$.

The computation of each set $\mathcal{P}_i$ runs in $O(kn^3)$ time for each G_i [45], or $O(kn^2)$ time if the graph is undirected [29]. Removing candidates from these sets and choosing the best k from the remaining paths using a standard linear-time selection algorithm uses only $O(rk)$ time.

A summary of the sequential algorithms for quickest path problems is presented in Table 4.1.

4.1.2 Finding the k Quickest Paths in Parallel

In the previous section, the problem of finding the k quickest paths from vertex s to vertex t in a graph G was solved by applying Eppstein's algorithm to find the k shortest paths from s^r to t' in a larger graph, G'. This suggests that the parallel k shortest paths algorithm developed in Chapter 3 could be applied to G' to solve the k quickest paths problem in the graph G. However, to apply the parallel k shortest paths algorithm to G', one must first compute a shortest path tree in G'. This would require a great deal of work: the best known parallel algorithm to compute a shortest path tree performs only slightly less than $O((rn)^3)$ work (see Section 3.1). To avoid having to perform shortest path computations on such a large graph, the problem will again be broken up by looking

	Problem	Algorithm	Running Time
1	Single pair quickest path	Y. Chen and Y. Chin [8]	$O(rm + rn\log n)$
2	All pairs quickest path	Repeated application of (1)	$O(rmn + rn^2\log n)$
3		Y.-C. Hung and G.-H. Chen [25]	$O(mn^2)$
4	Single pair k quickest paths	Section 4.1.1	$O(rm + rn\log n + k)$
5	All pairs k quickest paths	Repeated application of (4)	$O(rmn + rn^2\log n + n^2k)$
6		Improved version of G.-H. Chen and Y.-C. Hung [5]	$O(kmn^2\log k)$
7	Single pair k quickest simple paths	Improved version of Y. Chen [7]	$O(krn^3)$
8	Single pair k quickest simple paths in undirected graph	Improved version of Y. Chen [7]	$O(krn^2)$

Table 4.1: Sequential algorithms for quickest path problems

for shortest paths in each subgraph $G_i = (V, E_i)$, which contains only those edges with capacity at least c_i. The approach is similar to the sequential algorithm to compute the k quickest simple paths in the previous section.

Instead of computing the shortest path tree rooted at t in each of the G_i's separately, some of the work can be shared by using an incremental approach, since $E_r \subset E_{r-1} \subset \ldots \subset E_1$.

Lemma 4.2 *Suppose the $n \times n$ array D contains the distances between each pair of vertices in an edge-weighted n-node graph G. If m' new weighted edges are added to the edge set of G, the matrix D can be updated in $O(\log^2 m')$ time using $(n^2 m' \log m')$ work on an EREW PRAM.*

Proof: A shortest path in the new graph will consist of a sequence of new edges and shortest paths of the original graph G. The algorithm uses $O(\log m')$ iterations to update D, and each iteration doubles the number of new edges allowed in the paths which are considered as candidates for shortest paths:

```
For l = 1 to ⌈log(m' + 1)⌉
    For all pairs of vertices v_i, v_j, in parallel
        Compute min{D(v_i, u) + w(u, w) + D(w, v_j) | (u, w) is a new edge}
        If this quantity is less than D(v_i, v_j), assign it to D(v_i, v_j)
    End for
End for
```

The following invariant is satisfied: At the beginning of the lth iteration of the main loop, the matrix element $D(v_i, v_j)$ contains the weight of the shortest path from v_i to v_j which uses at most $2^{l-1} - 1$ of the newly added edges. This is clearly true at the beginning of the first iteration. Suppose the invariant is true at the beginning of iteration l. Any path from v_i to v_j with between 1 and $2^l - 1$ new edges can be broken into three subpaths: a subpath with at most $2^{l-1} - 1$ new edges, followed by a subpath consisting of a single new edge, and another subpath with at most $2^{l-1} - 1$ new edges. Each of the three subpaths must be optimal if the entire path is. So, the weight of the optimal path from v_i to v_j with between 1 and $2^l - 1$ new edges will be found during iteration l, and the invariant will be true at the beginning of the following iteration. The correctness of the algorithm follows, since there exists an optimal path between each pair of vertices that uses each new edge at most once.

Suppose that there are $n^2m'/\log m'$ processors available. Then, $m'/\log m'$ processors can be assigned to each pair of vertices v_i, v_j to compute the minimum of m' values, so that each iteration of the main loop can be carried out in $O(\log m')$ time using the sum algorithm described in Section 1.3.1, with minimization playing the role of the addition operator. The algorithm therefore runs in $O(\log^2 m')$ time, performing $O(n^2m' \log m')$ work. □

The algorithm described in the above proof can be used to compute the shortest paths in all r graphs $G_r, \ldots, G_1$ in $1 + \log r$ phases. (Without loss of generality, it is assumed that r is a power of 2.) During phase j, for $j = 0, \ldots, \log r$, the shortest paths in every $(r/2^j)$th G_i are computed by adding edges to a graph whose distance matrix is known. Let $c_{r+1} = \infty$ and let G_{r+1} be the graph with no edges. Initially the distance matrix for G_{r+1} is known (every entry on the diagonal is 0, and the remaining elements are infinite). In phase j, the algorithm of Lemma 4.2 is applied, starting with each graph G_i whose distance matrix is known, to obtain the distance matrix for $G_{i-r/2^j}$ (provided $i - r/2^j > 0$) by adding the edges with capacities in the half-open interval $[c_{i-r/2^j}, c_i)$. Figure 4.3 illustrates how the algorithm works. The arrows show how the distance matrix of each G_i is obtained from another distance matrix by applying Lemma 4.2.

The resources used by each phase are now considered. During any phase, no two applications of Lemma 4.2 add the same edge to one of the graphs. Each phase therefore involves a total of at most m edges being added to different G_i's. Since the number of edges added to any graph is at most $m \leq n^2$, each phase can be performed using $O(n^2 m \log n)$ work and $O(\log^2 n)$ time. Since there are $1 + \log r$ phases, the distance matrices for all of the G_i's can be computed in $O(\log^2 n \log r)$ time and $O(n^2 m \log n \log r)$ work.

If r is small enough that $rn < m \log n \log r$, then a more straightforward approach can be used without performing any more work. A separate transitive closure operation can be performed on

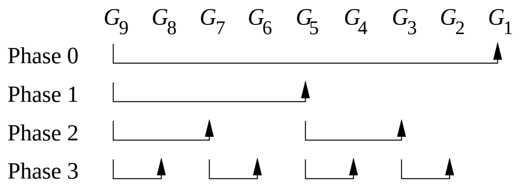


Figure 4.3: Computing distance matrices incrementally (r=8)

each G_i to compute the distance matrices using $O(\log^2 n)$ time and $O(rn^3)$ work. (See Section 3.1 for a description of the transitive closure computation.)

The computation described so far computes only the distances between vertices in each graph G_i. However, the computation can be augmented so that it also computes the shortest path in-tree rooted at vertex t by keeping track of the second vertex on each of the shortest paths whose weights are stored in the distance matrix D. This does not affect the running time or the amount of work performed.

When the shortest path tree rooted at t is known for each G_i, the parallel k shortest paths algorithm of Chapter 3 can be applied to each G_i, using the latencies as the weights of edges to compute $\mathcal{P}_i$, the set of the k shortest paths (with respect to latency) that have capacity at least c_i. $\mathcal{P}_i$ is also a set of the k quickest paths that have capacity at least c_i. $\mathcal{P}_i$ will be computed as an array of paths. The k shortest paths algorithm can be augmented to compute the capacity of the paths that it finds, without explicitly listing the edges of the path, as described in Section 3.3.4. From this information, the transmission times of each of the paths can be computed easily. Paths in $\mathcal{P}_i$ with capacity strictly greater than c_i will not be considered in the remainder of the computation, to avoid duplicate paths in the output. This can be done simply by assigning them a transmission time of ∞. The paths can then be sorted by transmission time using Cole's merge sort (see Section 1.3.3). The sort uses $O(\log(rk))$ time and $O(rk\log(rk))$ work. The first k paths are then the k quickest from s to t. The performance of the algorithm is summarized in Table 4.2.

Theorem 4.3 *An implicit representation of the k quickest paths from vertex s to vertex t of an n-node graph with m edges can be computed in $O(\log^2 n \log r + \log^3 k + \log n \log\log k)$ time and $O(n^2 m \log n \log r + rnk \log^2 k)$ work using an CREW PRAM.*

	Time	Work
Computing shortest path trees in G_i	$O(\log^2 n \log r)$	$O(n^2 m \log n \log r)$
Computing k shortest paths in G_i	$O(\log^3 k + \log n \log^* n + \log n \log\log k)$	$O(rm + rnk \log^2 k)$
Sorting the candidate paths	$O(\log(rk))$	$O(rk\log(rk)) = O(rk\log n + rk\log k)$
Total	$O(\log^2 n \log r + \log^3 k + \log n \log\log k)$	$O(n^2 m \log n \log r + rnk \log^2 k)$

Table 4.2: Performance of the parallel quickest paths algorithm

The paths themselves can be computed from the implicit representation, as described in Section 3.3.4 for the k shortest paths problem. This will not add to the running time of the algorithm, but the amount of work will increase by an amount proportional to the length of the output.

4.2 Adapting Dynamic Programming Algorithms to Find the k Best Solutions in Parallel

Dynamic programming is a widely used technique for the design of sequential algorithms to solve optimization problems. (See [12] for a general discussion of dynamic programming.) A dynamic programming algorithm starts with a recurrence relation that defines the value of the optimal solution for a given problem instance in terms of the values of optimal solutions to smaller problem instances. The value of the optimal solution is computed, starting with small subproblems and working up to larger ones, using the recurrence relation. Dynamic programming can often be viewed as the computation of the shortest path through a graph, as described in Section 4.2.1, below. In such cases, the k best solutions to the optimization problem can be obtained by finding the k shortest paths through the graph. In this section, this approach is described, using the Viterbi decoding problem as a concrete example. Other examples are described by Eppstein [13]. The parallel k shortest paths algorithm of Chapter 3 will be applied to obtain a parallel algorithm that computes the k best solutions to the Viterbi decoding problem.

4.2.1 Dynamic Programming and Shortest Paths

It is possible to set up a general framework that allows a large class of dynamic programming algorithms to be viewed as shortest path computations. Let $f^*(P)$ represent the cost of an optimal solution to the problem instance P. Suppose that the solution for a problem instance P can be described by a series of choices, each of which adds to the cost of the solution and reduces the problem instance to a smaller one. When the first choice is made, the problem instance P will be reduced to a smaller instance, P', from some set, $S(P)$, and the cost incurred by the choice is denoted by $w(P', P)$. A recurrence defining f^* will have the form

$$f^*(P) = \min_{P' \in S(P)} (f^*(P') + w(P', P)), \tag{4.1}$$

with suitable boundary conditions.

Finding the optimal solution to P can be viewed as a problem of finding a shortest path in an edge-weighted acyclic graph. There is a vertex in the graph for each possible problem instance that is no bigger than P. There is an edge (P', P) with weight $w(P', P)$ for each $P' \in S(P)$. In addition, there is a new vertex s and an edge (s, P_0) with weight $f^*(P_0)$ for each problem instance P_0 whose optimal cost is defined by the boundary conditions of the recurrence relation. Then, the problem of computing $f^*(P)$ can be solved by finding the shortest path from vertex s to vertex P. The choices that yield a solution with this optimal cost can be recovered by examining the shortest path itself.

Similarly, the k best solutions to the problem instance P may be obtained by finding the k shortest paths from s to P.

4.2.2 The Viterbi Decoding Problem

The Viterbi decoding problem is to estimate the state sequence of a discrete-time Markov process, given noisy observations of its state transitions. A Markov process is a probabilistic model for a system which is in one of s possible states at each time $\tau \in \{0, ..., T\}$. The behaviour of the system can be completely described by the sequence of states $\boldsymbol{X} = (X_0, \ldots, X_T) \in \{1, \ldots, s\}^{T+1}$; the system enters state X_τ at time τ. A Markov process does not remember its history: the probability of changing into a particular state at time τ may depend on the current state, $X_{\tau-1}$, and on τ but not on the previous states of the system, $X_1, \ldots, X_{\tau-2}$. The probabilities $p_{ij}(\tau) = P(X_\tau = j \mid X_{\tau-1} = i)$ are known, and are called the transition probabilities of the system at time $\tau > 0$. The probability p_i that the system begins in state i at time $\tau = 0$ is also known. At time τ, there is a transition from state $X_{\tau-1}$ to state X_τ. This transition of the Markov process produces an observable output, Y_τ. The observation Y_τ is a random variable that depends probabilistically on $X_{\tau-1}, X_\tau$, and possibly on τ. Each observation comes from a finite set of possible observations and occurs with known probability $q_{ij}(y, \tau) = P(Y_\tau = y \mid X_{\tau-1} = i \text{ and } X_\tau = j)$. This problem has applications in communications, and the randomness in the observations is used to model a Markov process whose transitions are being observed over a noisy channel (see [16]).

The Viterbi decoding problem is to compute the most likely state sequence that could produce the given observations of the system's transitions. Given a sequence of observations $\boldsymbol{y} = (y_1, \ldots, y_T)$, one must find the state sequence $\boldsymbol{x} = (x_1, \ldots, x_T)$ that maximizes the conditional probability $P(\boldsymbol{x} \mid \boldsymbol{y})$. This conditional probability can be expressed in terms of $P(\boldsymbol{y})$ and known quantities, using some elementary properties of probabilities:

$$\begin{aligned}
P(\boldsymbol{x} \mid \boldsymbol{y}) &= P(\boldsymbol{y} \mid \boldsymbol{x}) P(\boldsymbol{x}) / P(\boldsymbol{y}) \\
&= \Big(\prod_{\tau=1}^{T} q_{x_{\tau-1} x_\tau}(y_\tau, \tau) \Big) \Big(p_{x_0} \prod_{\tau=1}^{T} p_{x_{\tau-1} x_\tau}(\tau) \Big) / P(\boldsymbol{y}) \\
&= p_{x_0} \Big(\prod_{\tau=1}^{T} q_{x_{\tau-1} x_\tau}(y_\tau, \tau) p_{x_{\tau-1} x_\tau}(\tau) \Big) / P(\boldsymbol{y})
\end{aligned}$$

To maximize $P(\boldsymbol{x} \mid \boldsymbol{y})$ for a given sequence of observations, $\boldsymbol{y}$, it suffices to maximize the product $f(\boldsymbol{x}, \boldsymbol{y}) = p_{x_0} \prod_{\tau=1}^{T} q_{x_{\tau-1} x_\tau}(y_\tau, \tau) p_{x_{\tau-1} x_\tau}(\tau)$.

Viterbi presented a dynamic programming solution to this problem in 1967 [44]. The Viterbi algorithm runs in $O(s^2 T)$ time. In 1973, Forney suggested that the algorithm be augmented to output the k most likely state sequences [16]. This problem is called the list Viterbi decoding problem, and is of practical importance in a number of areas, including communications and speech recognition (see [16], [40], [41]). Seshadri and Sundberg describe a list Viterbi algorithm that lists the best k state sequences in $O(s^2T + ksT + k\log k)$ time [40]. They also present a naive parallel algorithm whose running time is $\Omega(T)$.

4.2.3 The Viterbi Algorithm

First, Viterbi's sequential algorithm for the Viterbi decoding problem will be described briefly. In Section 4.2.4 the k shortest paths algorithm of Chapter 3 will be used to obtain a parallel list Viterbi algorithm.

The selection of the optimal state sequence $\boldsymbol{x}$, given the sequence of observations $\boldsymbol{y}$, can be viewed as a series of choices: each state x_τ of the sequence $\boldsymbol{x}$ must be chosen from the set $\{1, \ldots, s\}$. Let $f^*(j, \boldsymbol{y})$ denote the maximum value of $f(\boldsymbol{x}, \boldsymbol{y})$ if the system is required to end in state $x_T = j$. A recurrence relation can be developed for the function f^*. The Viterbi algorithm [44] is then obtained by applying the technique of dynamic programming to compute the values of f^* using the recurrence. The optimal value of $f(\boldsymbol{x}, \boldsymbol{y})$ for the given observation sequence $\boldsymbol{y}$ will be computed as $\max_{1\le j\le s} f^*(j, \boldsymbol{y})$.

If $\boldsymbol{y}$ is a null sequence, then no transitions have occurred, and the only corresponding state sequence that ends in state j is the sequence of one element $\boldsymbol{x} = (j)$. This state sequence will produce the null sequence of observations with probability 1, and the probability of this state sequence occurring is just p_j, so $f^*(j, \text{null}) = p_j$.

Now consider the problem of computing $f^*(j, \boldsymbol{y})$ for an observation sequence $\boldsymbol{y} = (y_1, \ldots, y_T)$ of length $T > 0$. Any candidate state sequence must end in state j, and the second last state can be any one of the s possible states. If the second last state of $\boldsymbol{x}$ is chosen to be i, then $f(\boldsymbol{x}, \boldsymbol{y}) = f(\boldsymbol{x}', \boldsymbol{y}')\cdot q_{ij}(y_T, T)p_{ij}(T)$, where $\boldsymbol{y}' = (y_1, \ldots, y_{T-1})$ and $\boldsymbol{x}' = (x_0, \ldots, x_{T-1})$ are obtained by deleting the last element of $\boldsymbol{x}$ and $\boldsymbol{y}$. This probability is maximized when $\boldsymbol{x}'$ is chosen to maximize $f(\boldsymbol{x}', \boldsymbol{y}')$, subject to the constraint that the last state of $\boldsymbol{x}'$ is i. Maximizing over all choices for the second last state of $\boldsymbol{x}$ yields the recurrence

$$f^*(j, \boldsymbol{y}) = \max_{1\le i\le s} (f^*(i, \boldsymbol{y}') \cdot q_{ij}(y_T, T)p_{ij}(T)). \tag{4.2}$$

Now the recurrence (4.2) is used to reduce the Viterbi decoding problem to a shortest path computation. First, the recurrence for f^* must be manipulated into the form of Equation (4.1). This can be done by taking logarithms of both sides and multiplying by -1:

$$\begin{aligned} -\log f^*(j, \boldsymbol{y}) &= -\log \min_{1\le i\le s} (f^*(i, \boldsymbol{y}') \cdot q_{ij}(y_T, T)p_{ij}(T)) \\ &= \min_{1\le i\le s} (-\log f^*(i, \boldsymbol{y}') - \log q_{ij}(y_T, T) - \log p_{ij}(T)) \end{aligned} \tag{4.3}$$

Now, as described in Section 4.2.1, a graph (sometimes called a *trellis*) can be defined with a vertex, $[j,\tau]$, to represent the problem of computing $-\log(f^*(j,\boldsymbol{y}^\tau))$, for each state j and for each prefix $\boldsymbol{y}^\tau = (y_1,\ldots,y_\tau)$ of $\boldsymbol{y}$. Recurrence (4.3) is encoded in the graph by adding an edge from the vertex $[i,\tau-1]$ to vertex $[j,\tau]$, for all states i and j and for all values of $\tau \in \{1,2,\ldots,T\}$. In accordance with (4.3), the weight of such an edge is $-\log q_{ij}(y_\tau,\tau) - \log p_{ij}(\tau)$, which is a non-negative value. (If $q_{ij}(y_\tau,\tau)=0$ or $p_{ij}(\tau)=0$, the weight is taken to be infinite, and the edge can be left out of the graph.) The boundary conditions, $f^*(j,\text{null}) = p_j$, are encoded in the graph by creating a new vertex s, and joining it to the vertex $[j,0]$ by an edge with weight $-\log p_j$ for each state j. The minimal value of $-\log f^*(j,\boldsymbol{y})$ is then just the weight of the shortest path from vertex s to vertex $[j,T]$. An extra destination vertex, t, is also added to reflect the fact that the minimum value of $-\log f^*(j,\boldsymbol{y})$ over all possible states j is the value of the best state sequence; the vertex representing $[j,T]$ is connected to t by an edge of weight 0 for each state j. Then, the solution to the Viterbi decoding problem can be found by computing the shortest path from s to t. Similarly, the problem of computing the k best decodings may be solved by finding the k shortest paths from s to t.

For example, Figure 4.5 shows the trellis graph obtained for the Markov process of Figure 4.4, where vertices represent states, and possible state transitions are indicated by edges. The transition probabilities are not shown. The weight of edge e in Figure 4.5 is $-\log q_{32}(y_1,1) - \log p_{32}(1)$.

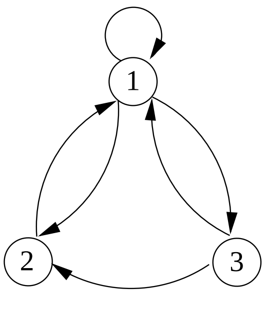


Figure 4.4: A 3-state Markov process

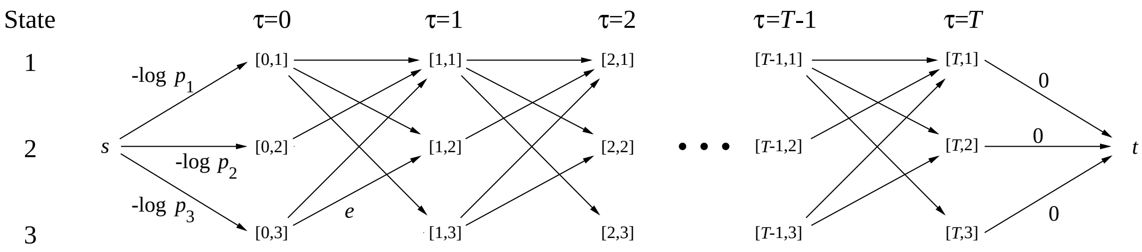


Figure 4.5: Trellis for the 3-state Markov process of Figure 4.4.

4.2.4 A Parallel Algorithm for the List Viterbi Problem

The parallel algorithm developed in Chapter 3 to solve the k shortest paths problem can now be applied to the trellis graph constructed in the previous section to solve the list Viterbi decoding problem.

The k shortest paths algorithm requires a shortest path tree rooted at t. Instead of using a standard transitive closure computation to build the tree (see Section 3.1), which would perform slightly less than $O(s^3T^3)$ work, the special structure of the graph can be exploited to make the computation more efficient. Let W_τ be the $s \times s$ matrix of weights of edges between vertices in columns $\tau - 1$ and τ; *i.e.* $(W_\tau)_{ij} = w([i, \tau - 1], [j, \tau])$. Let $D_\tau = \prod_{l=\tau+1}^{T} W_l$, where matrix products are computed using min/sum matrix multiplication (see Section 3.1). It is easy to see that $\mathrm{dist}([i,\tau],[j,T]) = (D_\tau)_{ij}$, and that $\mathrm{dist}([i,\tau],t)$ is the ith component of the product $D_\tau[0\ 0\ \cdots\ 0]^T$. Also, $\mathrm{dist}(s,t) = [-\log p_1\ \ -\log p_2\ \ \cdots\ \ -\log p_s]D_0[0\ 0\ \cdots\ 0]^T$. Each matrix multiplication can be carried out on an EREW PRAM using $O(\log s)$ time and $O(s^3)$ work. The matrix products D_τ can therefore be computed in $O(\log^2 s)$ time and $O(s^3T)$ work, using the prefix sum algorithm of Section 1.3.1, since matrix multiplication is associative. This computation will provide the distance to t from every vertex in the trellis. The computation of distances can be augmented by storing the second vertex on each of the optimal paths to vertex t to obtain a tree of shortest paths to the destination t. This algorithm for finding the shortest paths will, in fact, work for any layered directed graph. (A graph is layered if the vertices can be partitioned into a sequence of layers and each edge goes from a vertex in one layer to a vertex in the next layer.)

The trellis graph has $sT+2$ vertices, each with out-degree at most s, and at most s^2T+2s edges. If the k shortest paths algorithm of Chapter 3 is applied to this graph, an implicit representation of the k best decodings can be found on an CREW PRAM in $O(\log s \log^* s + \log(sT)\log\log k + \log^3 k \log k)$ time and $O(s^2T + sTk\log^2 k)$ work. The paths themselves can be extracted from the implicit representation in $O(\log k + \log(sT))$ time and $O(sT + kT)$ work. Summing up yields the following theorem.

Theorem 4.4 *The k best solutions to the Viterbi decoding problem can be output by a CREW PRAM using $O(\log^2 s + \log(sT)\log\log k + \log^3 k \log k)$ time and $O(s^3T + sTk\log^2 k)$ work.*

Chapter 5

Open Problems

There are many open problems related to the problems considered in this thesis.

A parallel algorithm for the k shortest paths was presented in Chapter 3. Not very much is known about the parallel complexity of finding the best k solutions to other optimization problems. Can the methods used here be extended to obtain parallel algorithms to find the k shortest *simple* paths, or the k smallest spanning trees of a graph, for example? Since most sequential algorithms for such problems generate the kth best solution using knowledge of the first $k-1$, it may be challenging to develop other parallel algorithms for this kind of problem that run in $O(\text{polylog}(k))$ time. In recent years, much attention has been focussed on dynamic versions of problems, where solutions must be quickly recomputed after small changes in the input. Can efficient algorithms be found for dynamic versions of k-best optimization problems?

Chapter 2 contained a parallel algorithm for selection in a matrix with ordered columns. Sarnath and He [38] gave a PRAM algorithm for selection in a sorted matrix, and gave lower bounds on the required resources. It would be interesting to develop a similar lower bound for selection in a matrix with ordered columns. Little work has been done on parallel algorithms for selection problems in other partially ordered structures.

The quickest path problem was discussed in Section 4.1. Most of the existing algorithms for this problem amount to repeated applications of shortest path algorithms, and consequently perform much more work than solutions to shortest path problems. Are quickest path problems really much more difficult than shortest path problems, or could a less naive algorithm solve the quickest path problem more efficiently than existing algorithms?

Bibliography

[1] K. Abrahamson, N. Dadoun, D. A. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. In *Proc. 25th Annual Allerton Conference on Communication, Control and Computing*, volume II, pages 624–633, 1987.

[2] Paul Beame and Johan Hastad. Optimal bounds for decision problems on the CRCW PRAM. In *Proc. 19th ACM Symposium on Theory of Computing*, pages 83–93, 1987.

[3] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[4] A. Borodin and J. E. Hopcroft. Routing, merging and sorting in parallel models of computation. *Journal of Computer and System Sciences*, 30:130–145, 1985.

[5] Gen-Huey Chen and Yung-Chen Hung. Algorithms for the constrained quickest path problem and the enumeration of quickest paths. *Computers and operations research*, 21(2):113–118, 1994.

[6] Y. L. Chen. An algorithm for finding the k quickest paths in a network. *Computers and Operations Research*, 20(1):59–65, 1993.

[7] Y. L. Chen. Finding the k quickest simple paths in a network. *Information Processing Letters*, 50(2):89–92, April 1994.

[8] Y. L. Chen and Y. H. Chin. The quickest path problem. *Computers and Operations Research*, 17(2):153–161, 1990.

[9] R. Cole. An optimally efficient selection algorithm. *Information Processing Letters*, 26:295–299, January 1988.

[10] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, August 1988.

[11] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Proc. 27th IEEE Symposium on Foundations of Computer Science*, pages 478–491, 1986.

[12] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 16. MIT Press, 1990.

[13] David Eppstein. Finding the k shortest paths. In *Proc. 35th IEEE Symposium on Foundations of Computer Science*, pages 154–165, 1994.

[14] Faith E. Fich. The complexity of computation on the parallel random access machine. In [36], Chapter 20.

[15] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, March 1975.

[16] G. David Forney, Jr. The Viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, March 1973.

[17] B. L. Fox. Calculating kth shortest paths. *INFOR; Canadian Journal of Operational Research*, 11(1):66–70, 1973.

[18] Greg N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104:197–214, 1993.

[19] Greg N. Frederickson and Donald B. Johnson. The complexity of selection and ranking in $X+Y$ and matrices with sorted columns. *Journal of Computer and System Sciences*, 24:197–208, 1982.

[20] Greg N. Frederickson and Donald B. Johnson. Generalized selection and ranking: Sorted matrices. *SIAM Journal on Computing*, 13(1):14–30, 1984.

[21] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[22] Y. Han, V. Pan, and J. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 353–362, 1992.

[23] Walter Hoffman and Richard Pavley. A method of solution of the Nth best path problem. *Journal of the ACM*, 6:506–514, 1959.

[24] Yung-Chen Hung and Gen-Huey Chen. On the quickest path problem. In *Advances in Computing and Information – International Conference on Computing and Information*, number 497 in Lecture Notes in Computer Science, pages 44–46, May 1991.

[25] Yung-Chen Hung and Gen-Huey Chen. Distributed algorithms for the quickest path problem. *Parallel Computing*, 18(7):823–834, 1992.

[26] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[27] Donald B. Johnson and Tetsuo Mizoguchi. Selecting the Kth element in $X + Y$ and $X_1 + X_2 + \ldots + X_n$. *SIAM Journal on Computing*, 7(2):147–153, 1978.

[28] Dimitrios Kagaris, Grammati E. Pantziou, Spyros Tragoudas, and Christos D. Zaroliagis. On the computation of fast data transmissions in networks with capacities and delays. In *4th Workshop on Algorithms and Data Structures*, pages 291–302, 1995.

[29] N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for K shortest simple paths. *Networks*, 12:411–427, 1982.

[30] S. Rao Kosaraju and Arthur L. Delcher. Optimal parallel evaluation of tree-structured computations by raking. In *VLSI Algorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, volume 319 of *Lecture Notes in Computer Science*, pages 101–110. Springer, 1988.

[31] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[32] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *Proc. 26th IEEE Symposium on Foundations of Computer Science*, pages 478–489, 1985.

[33] A. Mirzaian and E. Arjomandi. Selection in X + Y and matrices with sorted rows and columns. *Information Processing Letters*, 20(1):13–17, 1985.

[34] Richard C. Paige and Clyde P. Kruskal. Parallel algorithms for shortest paths problems. In *Proceedings of the International Conference on Parallel Processing*, pages 14–20, 1985.

[35] Margaret Reid-Miller, Gary L. Miller, and Francesmary Modugno. List ranking and parallel tree contraction. In [36], Chapter 3.

[36] John H. Reif, editor. *Synthesis of Parallel Algorithms.* Morgan Kaufmann, San Mateo, California, 1993.

[37] J. B. Rosen, S.-Z. Sun, and G.-L. Xue. Algorithms for the quickest path problem and the enumeration of quickest paths. *Computers and Operations Research*, 18(6):579–584, 1991.

[38] R. Sarnath and Xin He. On parallel selection and searching in partial orders. In *6th International Parallel Processing Symposium*, pages 108–111, 1992.

[39] A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *Journal of Computer and System Sciences*, 13:184–199, October 1976.

[40] Nambirajan Seshadri and Carl-Erik W. Sundberg. List Viterbi decoding algorithms with applications. *IEEE Transactions on Communications*, 42(2/3/4 Part I):313–323, 1994.

[41] Frank K. Soong and Eng-Fong Huang. A tree-trellis based fast search for finding the N best sentence hypotheses in continuous speech recognition. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 705–708, 1991.

[42] Leslie G. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4(3):348–355, 1975.

[43] Uzi Vishkin. An optimal parallel algorithm for selection. In *Advances in Computing Research*, volume 4, pages 79–86. JAI Press, 1987.

[44] Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13(2):260–269, April 1967.

[45] Jin Y. Yen. Finding the K shortest loopless paths. *Management Science*, 17(11):712–716, July 1971.

Index

active elements, definition, 19
acyclic graph, 3
ancestor in a tree, 4

capacity of a path, 41
Cartesian sum of vectors, 12
Cole's parallel merge sort, 7
cross-edge, 30
cycle, definition, 3

depth in trees, 4
descendant in a tree, 4
dynamic programming, 49

graph definitions, 3

head of an edge, 3
heap-ordered tree, 30
height of a tree, 4

in-tree, 3
indegree, 3

k shortest paths
- data structure for parallel algorithm, 34
- definition, 27
- extracting from data structure, 38
- parallel algorithm, 33
- sequential algorithm, 30

k shortest simple paths, 28

list Viterbi decoding
- definition, 51
- parallel algorithm, 53

Markov process, 50
merging on a PRAM, 6
min/sum matrix multiplication, 27
minimum ancestors, computing on a PRAM, 9

non-tree edge, definition, 28

out-tree, 4
outdegree, 3

path, definition, 3
PRAM, definition, 4
PRAM, measuring performance, 5
prefix sums, 5
processor allocation, 6

quickest paths
- definition, 41
- parallel algorithm, 45
- sequential algorithms, 42

rank, definition, 13

selection in matrix with sorted columns, 13
selection on a PRAM, 7
shortest path computation on a PRAM, 27
shortest path tree, 28
shunt operation, 8
sibling, 4
sidetracks, definition, 28
sorted matrix, definition, 12
sorting on a PRAM, 7

tail of an edge, 3
transitive closure, 27
transmission time, 41
tree contraction, 8
tree definitions, 3

Viterbi algorithm, 51
Viterbi decoding problem, 50

weighted selection, 10
work, definition, 5