Algorithmica (2000) 28: 242–254
DOI: 10.1007/s004530010038

Algorithmica
© 2000 Springer-Verlag New York Inc.

Finding the *k* Shortest Paths in Parallel[1]

E. Ruppert[2]

Abstract. A concurrent-read exclusive-write PRAM algorithm is developed to find the k shortest paths between pairs of vertices in an edge-weighted directed graph. Repetitions of vertices along the paths are allowed. The algorithm computes an implicit representation of the k shortest paths to a given destination vertex from every 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 pre-computed. The paths themselves can be extracted from the implicit representation in $O(\log k + \log n)$ time, and $O(n \log n + L)$ work, where L is the total length of the output.

Key Words. Parallel graph algorithms, Data structures, Shortest paths.

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 vertices of a graph also has a long history and many applications to a diverse range of problems. Many optimization problems may be formulated as the computation of a shortest path between two vertices in a graph. Often, the k best solutions to the optimization problem may then be found by computing the k shortest paths between the two vertices. A method for computing the k best solutions to an optimization problem may be useful if some constraints on the feasible solutions are difficult to specify formally. In this case, one can 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. Knowledge of the k best solutions to an optimization problem can also be helpful when determining whether the optimal solution is sensitive to small changes in the input. 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 instance would cause the sub-optimal solution to become optimal.

Sequential 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 directly to

[1] Financial support for this research was provided by the Natural Sciences and Engineering Research Council of Canada.
[2] Department of Computer Science, Brown University, Providence, RI 02912, USA. ruppert@cs.brown.edu.

Received July 2, 1997; revised June 18, 1998. Communicated by G. N. Frederickson.
Online publication May 18, 2000.

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.

A parallel algorithm is developed in Section 3 to compute the k shortest paths to a given vertex t from every vertex of an edge-weighted directed graph. It is assumed that the weights on the edges are positive, but the algorithm can easily be adapted to handle negative edge weights, as long as there are no negative cycles in the graph. The algorithm runs on a concurrent-read exclusive-write (CREW) PRAM. (See Karp and Ramachandran's survey [15] for definitions of PRAM models.) The algorithm finds the k shortest paths to t from every vertex in $O(\log^3 k \log^* k + \log n \log\log k + \log d \log^* d)$ time using $O(m + nk \log^2 k)$ work, where d is the maximum outdegree of any vertex in the graph, assuming that the shortest path to t from every other vertex is given. The algorithm computes an implicit representation of the paths, from which the paths themselves can be extracted in parallel using the techniques described in Section 3.2. New parallel algorithms for the weighted selection problem and the problem of selecting the kth smallest element in a matrix with sorted columns, which are used as subroutines, are outlined in Section 3.3. Some applications of the k shortest paths algorithm are described in Section 4.

Previous work. Dijkstra's sequential algorithm computes the shortest path to a given destination vertex from every other vertex in $O(m + n \log n)$ time [12]. In parallel, the shortest path between each pair of vertices can be found using a min/sum transitive closure computation in $O(\log^2 n)$ time and $O(n^3 \log n)$ work on an EREW PRAM [18]. More complicated implementations of the transitive closure computation run in $O(\log^2 n)$ time using $o(n^3)$ work on the EREW PRAM and in $O(\log n \log\log n)$ time on the CRCW PRAM [13]. There are no known polylogarithmic-time PRAM algorithms that find the shortest path from one particular vertex to another using less work than the all-pairs algorithm. This transitive closure bottleneck is not avoided by the algorithm presented here: the complexity bounds on the algorithm describe the amount of additional time and work to compute the k shortest paths, once the shortest paths are known.

The problem of finding the k shortest paths in sequential models of computation was discussed as early as 1959 by Hoffman and Pavley [14]. Fox presents an algorithm that can be implemented to run in $O(m + kn \log n)$ time [9]. Eppstein's recent sequential algorithm [7] is a significant improvement. It computes an implicit representation of the k shortest paths for a given source and destination in $O(m + n \log n + k)$ time. The k shortest paths to a given destination from every vertex in the graph can be found, using Eppstein's algorithm, in $O(m + n \log n + nk)$ time. The paths themselves can be extracted from the implicit representation in time proportional to the number of edges in the paths. A brief description of Eppstein's algorithm is given in Section 2.1. Kumar and Ghosh [17] independently developed a CREW PRAM algorithm for the all-pairs version of the k shortest paths problem. Their algorithm is an adaptation of the transitive closure algorithm for computing all-pairs shortest paths. They claim that the algorithm finds the k shortest simple paths, but repeated vertices may appear on the paths that they compute. There are some problems with their algorithm and their complexity analysis,

but it appears that they can be fixed so that the algorithm runs in $O((\log n + \log k)^4)$ time, performing $O(n^3k^2(\log k + \log n) + n^3(\log k + \log n)^3)$ work.

Sequential algorithms have been developed for other variations of the k shortest paths problem. Yen [24] gives an algorithm for the more difficult problem of finding the k shortest simple paths in $O(kn^3)$ time. Katoh et al. [16] describe an $O(kn^2)$ algorithm to find the k shortest simple paths in an undirected graph.

2. Preliminaries. Let $G = (V, E)$ be a directed graph with n vertices and m edges, where each edge (u, v) of E has 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 s to vertex t, $\text{dist}(s, t)$, is defined to be the weight of the path from s to t that minimizes this sum. A path that achieves this distance is a shortest path from s to t.

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. Here, paths are not restricted to being simple; a vertex may appear more than once on a path.

Let T be a tree with root t that is a subgraph of G and is constructed so that the (unique) path in T to t from any vertex v is a shortest v-t path in G. The tree T is called a shortest path tree of G rooted at t. 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 non-tree edges along the path. For example, in the graph shown in Figure 1(a), edges of T are shown as solid lines, and non-tree edges are shown as broken 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, $\langle (s, c), (a, f) \rangle$; the rest of the path can be filled in by following the edges of T. If p is any path, let $\text{sidetracks}(p)$ be the sequence of non-tree edges that occur along the path p.

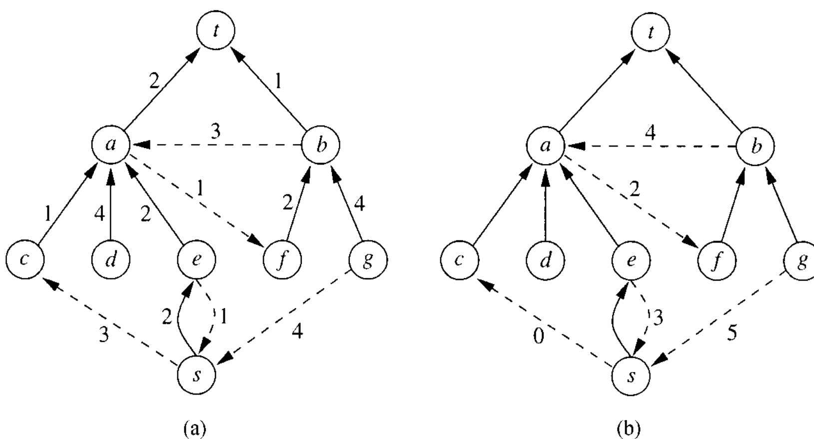

Fig. 1. An example graph. (a) Solid edges form the tree T. (b) Values of δ are shown for non-tree edges.

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

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

The following lemma describes some properties of this measure.

LEMMA 1 [7].

(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*,

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

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 to vertex t from any other vertex is 0. From now on, the weight function δ will be used instead of w.

2.1. *Eppstein's Sequential Algorithm.* Eppstein's sequential algorithm [7] computes an implicit representation of the k shortest paths. Each path's sidetracks sequence is represented as a modification of the sidetracks of a shorter path. Candidates for the kth shortest path are obtained from one of the shortest $k - 1$ paths either by adding a non-tree edge to the sidetracks sequence or by replacing the last non-tree edge by another one. The algorithm constructs a new weighted directed graph G' in which each path starting at a fixed vertex s' (and ending at any other vertex) corresponds to an s-t path of G. This correspondence is bijective and weight-preserving, so the k shortest s-t paths of G can be found by computing the k shortest paths that begin at s' in G', using Frederickson's algorithm [10]. It is possible that the ith shortest path is obtained by adding a non-tree edge to the $(i - 1)$th shortest path (for $2 \leq i \leq k$), so it appears that Eppstein's algorithm cannot be directly implemented in parallel with a running time that is polylogarithmic in k.

3. A Parallel Algorithm to Compute the k Shortest Paths.

The k shortest paths problem will be solved in stages. During the ith stage, paths with at most 2^i non-tree edges are considered and the k shortest of these paths to t from each vertex are computed. The following lemma shows that $\lceil \log k \rceil$ stages will be sufficient. (All logarithms have base 2.)

LEMMA 2. *There is a solution to the k shortest paths problem in which each path has 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 the weight of p, since δ is non-negative. So, the set of these paths is a correct solution to the problem, and each path has at most $k-1$ non-tree edges. □

The list of edges that make up each path will not be explicitly computed in each stage. Instead, each path is represented by a binary tree structure whose leaves represent the non-tree edges along the path. The ith stage constructs the implicit representations of paths by concatenating the sidetracks sequence of two paths that have been computed in previous stages. The result of such a concatenation is stored in the data structure by creating a new node, which will be the root of the tree representing the path, and setting its children pointers to point to the roots of the two smaller paths. Thus, the tree structures representing different paths may share common sub-trees, and the trees constructed during the ith stage have height at most i. Some additional information will be stored in each node of the tree structures to allow the the computations to be performed efficiently.

3.1. *The Data Structure Used by the Algorithm.* Let A_v^i be an array that will store the root nodes of the tree structures that represent 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.

Each array element stores the following information about the path p that it represents:

- pointers to the two previously computed paths whose sidetracks sequences were concatenated to form p, unless p contains only a single non-tree edge, in which case this edge is stored instead,
- the weight of the path (with respect to the weight function δ),
- the number of non-tree edges along the path,
- num, the number of edges on the path up to and including the last non-tree edge, and
- the head of the last non-tree edge on the path (this could be nil if all edges along the path are in T).

In the next section a parallel algorithm is given for extracting the k shortest paths from this implicit representation. This is done by allocating processors to traverse the leaves of the tree structures that represent the paths to obtain the sidetracks sequences and filling in the rest of the edges along the paths by traversing branches of the tree T. The actual construction of the data structure is described in Section 3.3.

3.2. *Extracting Information from the Data Structure.* First, some preprocessing is done to the shortest path tree, T. Each vertex uses pointer jumping to locate a pointer to its ancestor 2^i levels above itself, for $i = 1, 2, \ldots, \lceil \log(n-1) \rceil$. 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. This computation is done in $\lceil \log(n-1) \rceil$ steps: during the ith step, each vertex finds the ancestor 2^i levels above itself by following

two pointers computed in the previous step. This computation uses $O(\log n)$ time and $O(n \log n)$ work.

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 size $L = \sum_{j=1}^{k} l_j$. The starting location of each path in the array P can be found by performing a prefix sum (see [15]) on $l_1, \ldots, l_k$.

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 $\log(kn)$. To begin filling in its block of the array, the processor first determines which path it should be working on by doing a binary search of the prefix sums of $l_1, \ldots, l_k$. The processor then follows the pointers in the tree data structure that represents the path, starting from the root and going to the appropriate leaf to find the first edge it must write into P. At each node, the num field gives the number of edges in the sub-path represented by the sub-tree rooted at that node, so that the processor can determine whether to go left or right at each node on its way to the leaf. When it reaches a leaf, the processor begins filling in entries of P sequentially. 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 the tree data structure that represents the path, to the next leaf, which 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.

If the first edge that the processor 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 total 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 $O(n \log n + L_{\text{total}})$ work, where L_{total} is the total length of the output for all starting vertices v, since the preprocessing of T need only be done once.

In fact, some properties of the paths can be computed without explicitly listing the edges in the path, as observed by Eppstein [7]. Suppose each edge in the graph is assigned a value from a semi-group, and the value of a path is defined as the product of the values of the edges along the path. If the associative semi-group operation can be evaluated in constant time by a single processor, the values of the k shortest paths can be computed in the same way as the num field of the data structure, without affecting the performance bounds.

3.3. *Building the Data Structure.*

The following preliminary lemma describes a version of the common technique of parallel tree contraction which will be used during the construction of the data structure.

LEMMA 3 (Tree Contraction). *Let M be a monoid with an associative binary operation that can be computed in $O(S)$ time and $O(W)$ work on an EREW PRAM. Let T be a tree whose n vertices are labelled by elements of M. Let m_v be the result of combining*

the labels on the path from vertex v to the root using the binary operation. Then m_v can be computed for every vertex v in $O(S \log n)$ time and $O(Wn)$ work on an EREW PRAM.

PROOF. Reid-Miller et al. [19, page 163] describe how tree contraction can be used to compute the values m_v for binary trees, if the monoid is the set of natural numbers with the associative operation $x \cdot y = \max(x, y)$ and identity 0. However, their technique will work for any monoid. Their algorithm can be adapted to work for non-binary trees by replacing each vertex that has $c > 2$ children by a binary tree with $c - 1$ internal nodes whose leaves are the children of the original vertex. The newly created nodes are labelled by the identity element of M. This modification of the algorithm does not affect the asymptotic running time or work. □

The data structure will be constructed in stages. Stage i of the algorithm will compute A_v^i for each vertex v using the arrays computed in the previous stage.

The construction of A_v^0 is described first. 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 computed by finding the $k - 1$ shortest non-tree edges (with respect to δ) whose tails are on the path from v to t in T.

First, the $k - 1$ shortest non-tree edges whose tails are at vertex v are selected, for each vertex v in the graph. This can be done using $O(\log d_v \log^* d_v)$ time and $O(d_v)$ work, where d_v is the outdegree of v, using Cole's selection algorithm [5]. In total, this requires $O(\log d \log^* d)$ time and $O(m)$ work, where d is the maximum outdegree of any vertex in the graph. In addition, $O(\log n)$ time and $O(n)$ work is used to allocate the appropriate number of processors to each vertex using a prefix sum computation. Cole's parallel merge sort [6] can be used to sort the $k - 1$ smallest edges out of each vertex in $O(\log k)$ time using $n(k - 1)$ processors.

Lemma 3 can then be used to compute the array of the k shortest edges whose tails are on the path from each vertex v to the destination t. Here, the labels of the nodes are sorted arrays of k edges. The binary operation on the labels 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 Borodin and Hopcroft's merging algorithm [1], so the tree contraction takes $O(\log n \log \log k)$ time and $O(nk)$ work in total.

The num fields of the paths found during stage 0 can be filled in as follows. First, the depth of each vertex in T is computed using Lemma 3. Here, the label associated with each node is 1, except for the root, which has label 0, and the binary operation is addition. This computation uses $O(\log n)$ time and $O(n)$ work. The value of num for any path from v to t found during stage 0 of the algorithm can then be computed easily: if (x, y) is the (only) non-tree edge on the path, the value of num is $\text{depth}(v) - \text{depth}(x) + 1$.

Stage 0 of the algorithm uses $O(\log d \log^* d + \log k + \log n \log \log k)$ time in total, and performs $O(m + nk \log k)$ work.

Now, the computation of A_v^i, for $i > 0$, is described. The candidate paths for inclusion in A_v^i 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 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 $g \dashrightarrow s \dashrightarrow c \longrightarrow a \dashrightarrow f \longrightarrow b \longrightarrow t$.

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. 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 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 computation of A_v^0 described above ensures that the invariant is true for $i = 0$.

Assume 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. However, $p_1 p_3 \prec_v p_1 p_2 = p$, 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, $p_1 p_3$ 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 $r_1 \prec_w r_2 \prec_w \cdots \prec_w r_k \prec_w p_2$. However, this implies that $p_1 r_1 \prec_v p_1 r_2 \prec_v \cdots \prec_v p_1 r_k \prec_v p_1 p_2 = p$ (by the definition of the orders $\prec_v$ and $\prec_w$). This is a contradiction, so p_2 must be in A_w^{i-1}. □

It follows from Lemmata 2 and 4 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 CREW 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_1 and p_2, since the weight of each tree edge is 0. The num field is filled in similarly. 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 having l non-tree edges in A_v^{i-1} with the paths in two contiguous sub-arrays of A_w^{i-1}: the portions containing paths having l or $l-1$ non-tree edges. Each sub-array will already be sorted by $\prec_w$, so the paths that result from combining the path p_1 with the elements of the sub-array 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 computation of A_v^i amounts to the selection of the k smallest elements from a matrix with sorted columns. 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 will be the k smallest with respect to the ordering $\prec_v$.

Frederickson and Johnson's sequential algorithm for selection in a matrix with sorted columns [11] can be adapted to find the required k paths using $O(\log^2 k \log^* k)$ time and $O(k)$ work. The following lemma describes a parallel implementation of their algorithm for more general inputs.

LEMMA 5. *Let $k \le n$. There is an EREW PRAM algorithm that selects the* kth *smallest element of an $m \times n$ matrix with sorted columns using* $O(\log n \log^* n + \log \min(m,k) \log k \log^* k + \log k \log\log k)$ *time and* $O(n)$ *work.*

PROOF. Frederickson and Johnson's algorithm works by discarding elements that are not among the k smallest elements in the array until only $O(k)$ elements remain, at which time the kth smallest element can be selected directly. The elements that have not yet been discarded are called *active*. Let k' be the smallest power of two that is at least k. The algorithm first performs $1 + \min(\log k', \lfloor \log m \rfloor)$ iterations of a loop: for $i = 1, 2, 4, 8, \ldots$, it selects the (k'/i)th smallest active element, e_i, of row i and re-indexes the columns so that e_i appears in column k'/i and all smaller active elements in row i appear to the left of e_i. There are $i(k'/i) \ge k$ elements above and to the left of e_i, all of which are at most e_i. Thus, the elements below and to the right of e_i, which are at least e_i, may be discarded. Elements are not actually removed from the array; instead, the algorithm simply updates a variable that stores the row index of the last active element in the column. See Figure 2 for an illustration of the algorithm when $m = 10$, $n = 18$ and $k' = 16$. Elements discarded by this loop are shaded.

Cole's selection algorithm [5] is used to select the element e_i and a prefix sum computation is used to re-index the columns of the matrix. The first iteration of the loop is performed in $O(\log n \log^* n)$ time and $O(n)$ work. When the algorithm works on row $i > 1$, the row will contain $(2k'/i) - 1$ active elements, so the iteration can be completed

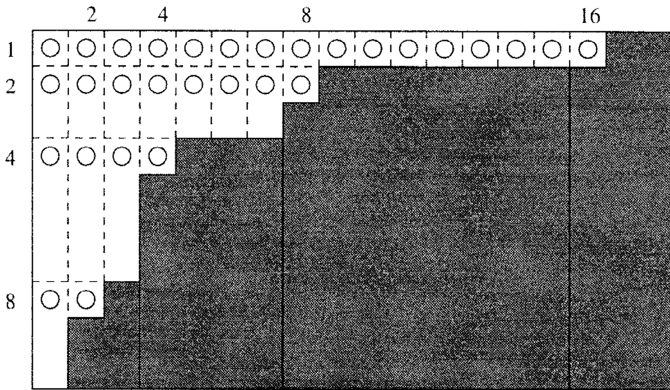

Fig. 2. Selection in a matrix with sorted columns.

in $O(\log(k'/i)\log^*(k'/i))$ time, performing $O(k'/i)$ work. The entire loop can therefore be completed in $O(\log n \log^* n + \log\min(m,k)\log k \log^* k)$ time and $O(n+k) = O(n)$ work.

Now, let Q be the set of active elements in the rows that were considered by the loop. These are the circled elements in Figure 2. The active elements of the array are partitioned into $O(k')$ sets. Each set is a contiguous block of a column that contains exactly one element of Q as its smallest element. This partition is indicated by broken lines in Figure 2. Each element of Q is assigned a weight equal to the size of its block. Lemma 6, below, is used to select the smallest element, q^*, of Q such that $\sum_{q\in Q, q\le q^*} \text{weight}(q) \ge 2k'$. This is done in $O(\log k \log\log k)$ time and $O(k)$ work. Those sets that contain an element of Q that is larger than q^* are discarded. Since each set's weight is at most k', the number of active elements remaining is at least $2k'$ and less than $3k' = O(k)$. In any column at least half of the elements that remain active are less than or equal to q^*: if the last element of Q in the column that remains active appears in row 2^j, then at most $2^j - 1$ of the active elements in the column are greater than q^*. Thus, at least $2k'/2 \ge k$ of the active elements are smaller than q^*. No element smaller than q^* is discarded in this step, so the k smallest elements in the array remain active.

Cole's selection algorithm can now be used to select the kth smallest element directly from the remaining $O(k)$ active elements in $O(\log k \log^* k)$ time and $O(k)$ work. □

LEMMA 6. *Suppose $[x_1, \ldots, x_n]$ is an unsorted array of n distinct integers. A nonnegative integer weight w_i is associated with each element x_i. Given a number K, one can compute $x^* = \min\{x_i \mid \sum_{x_j\le x_i} w_j \ge K\}$ using $O(\log n \log\log n)$ time and $O(n)$ work on an EREW PRAM.*

PROOF. Vishkin's parallel algorithm to select the kth smallest element of an unsorted array [23] can be adapted to provide the proof of this lemma. Vishkin's algorithm partitions the array of elements into groups and finds the median a of the medians of these groups. To reduce the size of the problem, the algorithm then discards either those elements smaller than a if the number of such elements is less than k, or those elements

larger than a, otherwise. The weighted selection algorithm is similar, except for the method of choosing elements to discard. If the total weight of the elements less than a is less than K, the algorithm discards them, and otherwise discards the elements greater than a. The resources required by the weighted selection algorithm are the same as for Vishkin's algorithm. □

A more detailed discussion of parallel algorithms for the weighted selection problem and the problem of selecting an element from a matrix with sorted columns may be found in [20].

Once the k elements of A_v^i are found using the algorithm of Lemma 5, they can be sorted in $O(\log k)$ time and $O(k \log k)$ work using Cole's parallel merge sort [6]. 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.

When the $\lceil \log k \rceil$ stages of the k shortest paths algorithm have been completed, the k shortest paths are stored implicitly in the data structure. This completes the proof of the following theorem.

THEOREM 7. *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 the destination vertex t, an implicit representation of the k shortest paths to t from every vertex 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.*

As described in Section 3.2, the paths can be explicitly listed using $O(\log k + \log n)$ time and $O(n \log n + L)$ work, where L is the length of the output.

The parallel algorithm developed here will also work on weighted directed multigraphs. The performance bounds proven above still 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. Lemma 1 still applies in this case. Therefore, the transformation used to reduce a general problem instance to one with non-negative edge weights (defined by δ), where the distance to vertex t from any other vertex is 0, also applies to graphs with negative edge weights that have no negative cycles.

4. Applications. Two applications of the k shortest paths problem will now be described. More detailed descriptions of these applications may be found in [20].

The Viterbi decoding problem is to estimate the state sequence of a discrete-time Markov process, given noisy observations of its state transitions. This problem has applications in communications (see [8]). The list Viterbi decoding problem is to compute the k state sequences that are most likely to have occurred, given a particular sequence of observations. Sequential algorithms exist that construct a weighted directed acyclic graph in which each path between two fixed vertices describes a possible state sequence [8]. The weight of the path corresponding to a state sequence is equal to the conditional probability that it occurred, given the observations. Sequential algorithms for the list Viterbi problem have appeared previously [21], [22], and a straightforward parallel implementation was

described by Seshadri and Sundberg [21]. The parallel k shortest paths algorithm can be applied to this graph to solve the list Viterbi decoding problem on a CREW PRAM using $O(\log^2 s + \log(sT) \log\log k + \log^3 k \log k)$ time and $O(s^3 T + sTk \log^2 k)$ work, where s is the size of the state space of the Markov process and T is the length of the observation sequence. This is the first parallel algorithm for this problem that runs in polylogarithmic time.

The quickest path problem [4] is a generalization of the shortest path problem. It is used to model the problem of transmitting data through a computer network. Each edge of a directed graph is assigned a positive capacity and a non-negative latency. The capacity $c(p)$ of a path p is defined to be the minimum capacity of any edge on the path. The latency $l(p)$ is the sum of the latencies of the edges of p. The time to transmit σ bits of data along a path p is $l(p) + \sigma/c(p)$. The k quickest paths problem is to compute the k paths that require the least time to transmit a given amount of data between a given pair of vertices. If all edge capacities are equal, the problem reduces to the k shortest paths problem. Sub-paths of quickest paths need not be quickest paths themselves, so the approaches used to solve shortest path problems are not directly applicable to quickest path problems. Sequential algorithms for the k quickest paths problem have been studied previously [2], [3], [20]. The problem can be solved by first finding the k shortest paths (with respect to latency) that have capacity at least c, for each edge capacity c, and then choosing from among them the k paths that have the shortest overall transmission time [3]. Repeated use of the parallel k shortest paths algorithm yields a CREW PRAM implementation of this approach that finds the k quickest paths to a given destination vertex t from every vertex v. For an n-node graph with m edges and r distinct edge capacities, the parallel algorithm runs in $O(\log^2 n \log r + \log^3 k + \log n \log\log k)$ time using $O(n^2 m \log n \log r + rnk \log^2 k)$ work.

Acknowledgements. I thank my adviser Faith Fich for many invaluable discussions throughout the development of this paper. I also thank Derek Corneil and the anonymous referee for their helpful comments.

References

[1] 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.

[2] G.-H. Chen and Y.-C. Hung. Algorithms for the constrained quickest path problem and the enumeration of quickest paths. *Computers and Operations Research*, 21(2):113–118, 1994.

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

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

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

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

[7] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, April 1999.

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

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

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

[11] G. N. Frederickson and D. 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.

[12] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[13] Y. Han, V. Y. Pan, and J. H. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. *Algorithmica*, 17:399–415, Apr. 1997.

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

[15] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 871–941. Elsevier, Amsterdam, 1990.

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

[17] N. Kumar and R. K. Ghosh. Parallel algorithm for finding first k shortest paths. *Computer Science and Informatics*: *Journal of the Computer Society of India*, 24(3):21–28, 1994.

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

[19] M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann, San Mateo, California, 1993.

[20] E. Ruppert. Parallel algorithms for the k shortest paths and related problems. Master's thesis, University of Toronto, 1996. Available from http://www.cs.utoronto.ca/∼ruppert.

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

[22] F. K. Soong and E.-F. 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.

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

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