Improved Shortest Paths on the Word RAM

Torben Hagerup

Fachbereich Informatik
Johann Wolfgang Goethe-Universität Frankfurt
D–60054 Frankfurt am Main, Germany
hagerup@informatik.uni-frankfurt.de

Abstract. Thorup recently showed that single-source shortest-paths problems in undirected networks with n vertices, m edges, and edge weights drawn from $\{0, \ldots, 2^w - 1\}$ can be solved in $O(n + m)$ time and space on a unit-cost random-access machine with a word length of w bits. His algorithm works by traversing a so-called *component tree*. Two new related results are provided here. First, and most importantly, Thorup's approach is generalized from undirected to directed networks. The resulting time bound, $O(n + m \log w)$, is the best deterministic linear-space bound known for sparse networks unless w is superpolynomial in $\log n$. As an application, all-pairs shortest-paths problems in directed networks with n vertices, m edges, and edge weights in $\{-2^w, \ldots, 2^w\}$ can be solved in $O(nm + n^2 \log\log n)$ time and $O(n + m)$ space (not counting the output space). Second, it is shown that the component tree for an undirected network can be constructed in deterministic linear time and space with a simple algorithm, to be contrasted with a complicated and impractical solution suggested by Thorup. Another contribution of the present paper is a greatly simplified view of the principles underlying algorithms based on component trees.

1 Introduction

The *single-source shortest-paths* (*SSSP*) problem asks, given a network $\mathcal{N}$ with real-valued edge lengths and a distinguished vertex s in $\mathcal{N}$ called the *source*, for shortest paths in $\mathcal{N}$ from s to all vertices in $\mathcal{N}$ for which such shortest paths exist. It is one of the most fundamental and important network problems from both a theoretical and a practical point of view. Actually, the more fundamental and important problem is that of finding a shortest path from s to a single given vertex t, but this does not appear to be significantly easier than solving the complete SSSP problem with source s.

This paper considers mainly the important special case of the SSSP problem in which all edge lengths are nonnegative. The classic algorithm for this special case is *Dijkstra's algorithm* [5,6,24]. Dijkstra's algorithm maintains for every vertex v in $\mathcal{N}$ a *tentative distance* from s to v, processes the vertices one by one, and always selects as the next vertex to be processed one whose tentative distance is minimal. The operations that need to be carried out on

U. Montanari et al. (Eds.): ICALP 2000, LNCS 1853, pp. 61–72, 2000.
© Springer-Verlag Berlin Heidelberg 2000

the set of unprocessed vertices—in particular, identifying a vertex with minimal *key* (= tentative distance)—are supported by the *priority-queue* data type, and therefore an efficient implementation of Dijkstra's algorithm essentially boils down to an efficient realization of the priority queue.

Suppose that the graph underlying $\mathcal{N}$ is $G = (V, E)$ and take $n = |V|$ and $m = |E|$. An implementation of Dijkstra's algorithm with a running time of $O(n + m \log n)$ is obtained by realizing the priority queue through a binary heap or a balanced search tree. Realizing the priority queue by means of a *Fibonacci heap*, Fredman and Tarjan [8] lowered the time bound to $O(m + n \log n)$. The priority queues mentioned so far are comparison-based. In reality, however, edge lengths are numeric quantities, and very frequently, they are or can be viewed as integers. In the remainder of the paper, we make this assumption, which lets a host of other algorithms come into play.

Our model of computation is the *word RAM*, which is like the classic unit-cost random-access machine, except that for an integer parameter $w \geq 1$ called the *word length*, the contents of all memory cells are integers in the range $\{0, \ldots, 2^w - 1\}$, and that some additional instructions are available. Specifically, the available unit-time operations are assumed to include addition and subtraction, (noncyclic) bit shifts by an arbitrary number of positions, and bitwise boolean operations, but not multiplication (the *restricted instruction set*). Our algorithm for undirected networks in addition assumes the availability of a unit-time "most-significant-bit" (MSB) instruction that, applied to a positive integer r, returns $\lfloor \log r \rfloor$ (all logarithms are to base 2). When considering an instance of the SSSP problem with n vertices, we assume that $w > \log n$, since otherwise n is not a representable number. In the same vein, when nothing else is stated, edge weights are assumed to be integers in the range $\{0, \ldots, 2^w - 1\}$.

We now discuss previous algorithms for the SSSP problem that work on the word RAM, focusing first on deterministic algorithms. A well-known data structure of van Emde Boas et al. [23] is a priority queue that allows insertion and deletion of elements with keys in $\{0, \ldots, C\}$ as well as the determination of an element with minimal key in $O(\log \log C)$ time per operation. This implies an SSSP algorithm with a running time of $O(n + m \log w)$. In more recent work, Thorup [18] improved this to $O(n+m \log \log n)$, the best bound known for sparse networks. Both algorithms, however, use $2^{\Theta(w)}$ space, which makes them impractical if w is larger than $\log n$ by a nonnegligible factor. A different algorithm by Thorup [20] achieves $O(n + m(\log \log n)^2)$ time using linear space, $O(n + m)$. Algorithms that are faster for denser networks were indicated by Ahuja et al. [2], Cherkassky et al. [4], and Raman [14,15,16]; their running times are of the forms $O(m + n(\log n)^{\Theta(1)})$ and $O(m + nw^{\Theta(1)})$. Some of these algorithms employ randomization, multiplication, and/or superlinear space. Using randomization, an expected running time of $O(n + m \log \log n)$ can be achieved in linear space [19].

Our first result is a new deterministic algorithm for the SSSP problem that works in $O(n + m \log w)$ time. The time bound is never better than that of Thorup [18]. The new algorithm, however, works in linear space. For sparse networks, the new algorithm is faster than all previous deterministic linear-

space solutions unless w is superpolynomial in $\log n$. We actually prove a more general result: If the edges can be partitioned into $b \geq 2$ groups such that the lengths of the edges within each group differ by at most a constant factor, the SSSP problem can be solved in $O(n \log\log n + m \log b)$ time. Our construction implies that the all-pairs shortest-paths (*APSP*) problem in networks with n vertices, m edges, and edge weights drawn from $\{-2^w, \ldots, 2^w\}$ can be solved in $O(nm + n^2 \log\log n)$ time and $O(n + m)$ space (not counting the output space). No faster APSP algorithm is known for any combination of n and m. If $m = \omega(n)$ and $m = o(n \log n)$, the new algorithm is faster than all previous algorithms.

In a remarkable development, Thorup [21] showed that the SSSP problem can be solved in linear time and space for undirected networks. His algorithm is all the more interesting in that it is not a new implementation of Dijkstra's algorithm. The vertices are still processed one by one, but the strict processing in the order of increasing tentative distance is abandoned in favor of a more permissive regime, the computation being structured with the aid of a so-called *component tree*. Thorup provides two algorithms for constructing the component tree. One uses the *Q-heap* data structure of Fredman and Willard [9] and works in $O(n + m)$ time, but is complicated and utterly impractical. The other one is simple and conceivably practical, but its running time is $O(m\alpha(m, n))$, where α is an "inverse Ackermann" function known from the analysis of a union-find data structure [17]. Our second result is a procedure for computing the component tree of an undirected network that is about as simple as Thorup's second algorithm, but works in linear time and space.

2 Shortest Paths in Directed Networks

This section proves our main result:

Theorem 1. *For all positive integers n, m and w with $w > \log n \geq 1$, single-source shortest-paths problems in networks with n vertices, m edges, and edge lengths in the range $\{0, \ldots, 2^w - 1\}$ can be solved in $O(n + m \log w)$ time and $O(n + m)$ space on a word RAM with a word length of w bits and the restricted instruction set.*

Let us fix a network $\mathcal{N}$ consisting of a directed graph $G = (V, E)$ and a length function $c : E \to \{0, \ldots, 2^w - 1\}$ as well as a source $s \in V$ and take $n = |V| \geq 2$ and $m = |E|$. We assume without loss of generality that G is strongly connected, i.e., that every vertex is reachable from every other vertex. Then $m \geq n$, and we can define $\delta(v)$ as the length of a shortest path in G from s to v, for all $v \in V$. It is well-known that knowledge of $\delta(v)$ for all $v \in V$ allows us, in $O(m)$ time, to compute a shortest-path tree of G rooted at s (see, e.g., [1, Section 4.3]), so our task is to compute $\delta(v)$ for all $v \in V$.

Dijkstra's algorithm for computing $\delta(v)$ for all $v \in V$ can be viewed as simulating a fire starting at s at time 0 and propagating along all edges at unit speed. The algorithm maintains for each vertex v an upper bound $d[v]$ on the (simulated) time $\delta(v)$ when v will be reached by the fire, equal to the time when

v will be reached by the fire from a vertex u already on fire with $(u, v) \in E$ (∞ if there is no such vertex). Whenever a vertex u is attained by the fire (u is *visited*), the algorithm reconsiders $d[v]$ for each unvisited vertex v with $(u, v) \in E$ and, if the current estimate $d[v]$ is larger than the time $d[u] + c(u, v)$ when v will be hit by the fire from u, decreases $d[v]$ to $d[u] + c(u, v)$; in this case we say that the edge (u, v) is *relaxed*. The simulated time is then stepped to the time of the next visit of a vertex, which is easily shown to be the minimal d value of an unvisited vertex.

Consider a distributed implementation of Dijkstra's algorithm in which each vertex v is simulated by a different processor P_v. The relaxation of an edge (u, v) is implemented through a message sent from P_u to P_v and specifying the new upper bound on $\delta(v)$. For each $v \in V$, P_v receives and processes all messages pertaining to relaxations of edges into v, then reaches the simulated time $d[v]$ and visits v, and subsequently sends out an appropriate message for each edge out of v that it relaxes. The implementation remains correct even if the processors do not agree on the simulated time, provided that each message is received in time: For each vertex v, a message specifying an upper bound of t on $\delta(v)$ should be received by P_v before it advances its simulated time beyond t. If such a message corresponds to the relaxation of an edge $e = (u, v)$, it was generated and sent by P_u at its simulated time $t - c(e)$. Provided that messages have zero transit times, this shows that for all $e = (u, v) \in E$, we can allow the simulated time of P_u to lag behind that of P_v by as much as $c(e)$ without jeopardizing the correctness of the implementation. In order to capitalize on this observation, we define a *component tree* as follows.

Take the *level* of each edge $e \in E$ to be the integer i with $2^{i-1} \leq c(e) < 2^i$ if $c(e) > 0$, and 0 if $c(e) = 0$. For each integer i, let G_i be the subgraph of G spanned by the edges of level at most i. A *component tree* for $\mathcal{N}$ is a tree T, each of whose nodes x is marked with a *level* in the range $\{-1, \ldots, w\}$, $level(x)$, and a *priority* in the range $\{0, \ldots, n-1\}$, $priority(x)$, such that the following conditions hold:

1. The leaves of T are exactly the vertices in G, and every inner node in T has at least two children.
2. Let x and y be nodes in T, with x the parent of y. Then $level(x) > level(y)$.
3. Let u and v be leaf descendants of a node x in T. Then there is a path from u to v in $G_{level(x)}$.
4. Let u and v be leaf descendants of distinct children y and z, respectively, of a node x in T. Then $priority(y) < priority(z)$ or there is no path from u to v in $G_{level(x)-1}$.

The component tree is a generalization of the component tree of Thorup [21]. Let $T = (V_T, E_T)$ be a component tree for $\mathcal{N}$ and, for all $x \in V_T$, let G_x be the subgraph of $G_{level(x)}$ spanned by the leaf descendants of x. The conditions imposed above can be shown to imply that for every $x \in V_T$, G_x is a strongly connected component (SCC) of $G_{level(x)}$, i.e., a maximal strongly connected subgraph of $G_{level(x)}$.

We carry out a simulation of the distributed implementation discussed above, imagining the processor P_v as located at the leaf v of T, for each $v \in V$. The rest of T serves only to enforce a limited synchronization between the leaf processors, as follows: For each integer i, define an *i-interval* to be an interval of the form $[r \cdot 2^i, (r+1) \cdot 2^i)$ for some integer $r \geq 0$. Conceptually, the algorithm manipulates *tokens*, where an i-token is an abstract object labeled with an i-interval, for each integer i, and a token is an i-token for some i. A nonroot node x in T occasionally receives from its parent a token labeled with an interval I, interpreted as a permission to advance its simulated time across I. If x has level i and is not a leaf, it then splits I into consecutive $(i-1)$-intervals $I_1, \ldots, I_k$ and, for $j = 1, \ldots, k$, steps through its children in an order of nondecreasing priorities and, for each child y, sends an $(i-1)$-token labeled with I_j to y and waits for a completion signal from y before stepping to its next child or to the next value of j. Once the last child of x has sent a completion signal for the last token to x, x sends a completion signal to its parent.

The root of T behaves in the same way, except that it neither generates completion signals nor receives tokens from a parent; we can pretend that the root initially receives a token labeled with the interval $[0, \infty)$. A leaf node v, upon receiving a token labeled with an interval I from its parent, checks whether $d[v] \in I$ and, if so, visits v and relaxes all edges leaving v that yield a smaller tentative distance. No "relaxation messages" need actually be generated; instead, the corresponding decreases of d values are executed directly. Similarly, although the simulation algorithm was described above as though each node in T has its own processor, it is easily turned into a recursive or iterative algorithm for a single processor.

Consider a relaxation of an edge $(u, v) \in E$ and let x, y, and z be as in condition (4) in the definition of a component tree. Then either $\mathit{priority}(y) < \mathit{priority}(z)$, in which case the simulated time of P_u never lags behind that of P_v, or $c(u, v) \geq \lfloor 2^{i-1} \rfloor$, where $i = \mathit{level}(x)$. Since the synchronization enforced by x never allows the simulated times of two processors at leaves in its subtree to differ by more than 2^{i-1}, our earlier considerations imply that the simulation is correct, i.e., the value of $\delta(v)$ is computed correctly for all $v \in V$.

As described so far, however, the simulation is not efficient. It is crucial not to feed tokens into a node x in T before the first token that actually enables a leaf descendant of x to be visited, and also to stop feeding tokens into x after the last leaf descendant of x has been visited. Thus each node x of T initially is *dormant*, then it becomes *active*, and finally it becomes *exhausted* (except for the root of T, which is always active). The transition of x from the dormant to the active state is triggered by the parent of x producing a token labeled with an interval that contains $d[x]$, defined to be the smallest d value of a leaf descendant of x. When the last leaf descendant of x has been visited, on the other hand, x notifies its parent that it wishes to receive no more tokens and enters the exhausted state. If x is an inner node, this simply means that x becomes exhausted when its last child becomes exhausted.

The following argument of Thorup [21] shows the total number of tokens exchanged to be $O(m)$: The number of tokens "consumed" by a node x in T of level i is at most 1 plus the ratio of the diameter of G_x to 2^i. The "contribution" of a fixed edge in E to the latter ratio, at various nodes x on a path in T, is bounded by $\sum_{j=0}^{\infty} 2^{-j} = 2$. Since T has fewer than $2n$ nodes, the total number of tokens is bounded by $2n + 2m$. In order to supply its children with tokens in the right order without violating the constraints implied by the children's priorities, each inner node in T initially sorts its children by their priorities. Using two-pass radix sort, this can be done together for all inner nodes in $O(n)$ total time. Taking the resulting sequence as the universe, each inner node subsequently maintains the set of its active children in a sorted list and, additionally, in a van Emde Boas tree [23]. The sorted list allows the algorithm to step to the next active child in constant time, and the van Emde Boas tree allows it to insert or delete an active child in $O(\log\log n)$ time. As there are $O(n)$ insertions and deletions of active nodes over the whole simulation, the total time needed is $O(m + n\log\log n) = O(m\log w)$. Since it is not difficult to implement the van Emde Boas tree in space proportional to the size of the universe [22], the total space needed by all instances of the data structure is $O(n)$.

If we ignore the time spent in constructing T and in discovering nodes that need to be moved from the dormant to the active state, the running time of the algorithm is dominated by the contribution of $O(m\log w)$ identified above. We now consider the two remaining problems.

2.1 Constructing the Component Tree

We show how to construct the component tree T in $O(m\min\{n, \log w\})$ time, first describing a simple, but inefficient algorithm.

The algorithm maintains a forest F that gradually evolves into the component tree. Initially $F = (V, \emptyset)$, i.e., F consists of n isolated nodes. Starting from a network $\mathcal{N}_{-1}$ that also contains the elements of V as isolated vertices and no edges, the algorithm executes $w+1$ *stages*. In Stage j, for $j = 0, \ldots, w$, a network $\mathcal{N}_j$ is obtained from $\mathcal{N}_{j-1}$ by inserting the edges in $\mathcal{N}$ of level j, computing the SCCs of the resulting network, and contracting the vertices of each nontrivial SCC to a single vertex. In F, each contraction of the vertices in a set U is mirrored by creating a new node that represents U, giving it level j, and making it the parent of each node in U. Suitable priorities for the vertices in U are obtained from a topological sorting of the (acyclic) subgraph of $\mathcal{N}_{j-1}$ spanned by the vertices in U. So that the remaining edges can be inserted correctly later, their endpoints are updated to reflect the vertex contractions carried out in Stage j. The resulting tree is easily seen to be a component tree.

Assuming that $w \leq m$, we lower the construction time from $O(mw)$ to $O(m\log w)$ by carrying out a preprocessing step that allows each stage to be executed with only the edges essential to that stage. For each $e = (u, v) \in E$, define the *essential level* of e to be the unique integer $i \in \{-1, \ldots, w-1\}$ such that u and v belong to distinct SCCs in G_i, but not in G_{i+1}. Starting with $E_{-1} = E$ and $E_0 = E_1 = \cdots = E_{w-1} = \emptyset$, the following recursive algorithm,

which can be viewed as a batched binary search, stores in E_i the subset of the edges in E of essential level i, for $i = -1, \ldots, w-1$. The outermost call is *BatchedSearch*$(-1, w)$.

procedure *BatchedSearch*(i, k):
  if $k - i \geq 2$ **then**
    $j := \lfloor (i+k)/2 \rfloor$;
    Let $\mathcal{N}_j$ be the subnetwork of $\mathcal{N}$ spanned by the edges in E_i of level $\leq j$;
    Compute the SCCs of $\mathcal{N}_j$;
    Move from E_i to E_j each edge with endpoints in distinct SCCs of $\mathcal{N}_j$;
    Contract each SCC of $\mathcal{N}_j$ to a single vertex
      and rename the endpoints of the edges in E_j accordingly;
    BatchedSearch(i, j);
    BatchedSearch(j, k);

The calls of *BatchedSearch* form a complete binary call tree of depth $O(\log w)$. If a call *BatchedSearch*(i, k) is associated with its lower argument i, each edge can be seen to belong to only one set E_i whose index i is associated with a call at a fixed level in the call tree. Since all costs of a call *BatchedSearch*(i, k), exclusive of those of recursive calls, are $O(1 + |E_i|)$, the execution time of the algorithm is $O(w + m \log w) = O(m \log w)$. Moreover, it can be seen that at the beginning of each call *BatchedSearch*(i, k), E_i contains exactly those edges in E whose endpoints belong to distinct SCCs in G_i, but not in G_k. Applied to the leaves of the call tree, this show the output of *BatchedSearch* to be as claimed.

We now use the original, simple algorithm, with the following modifications: (1) Instead of renaming edge endpoints explicitly, we use an efficient union-find data structure to map the endpoints of an edge to the nodes that resulted from them through a sequence of node contractions. Over the whole construction, the time needed for this is $O(m\alpha(m, n)) = O(m + n \log \log n) = O(m \log w)$ [17]. (2) In Stage j, for $j = 0, \ldots, w$, $\mathcal{N}_j$ is obtained from $\mathcal{N}_{j-1}$ by inserting each edge in $\mathcal{N}$ that was not inserted in an earlier stage, whose endpoints were not contracted into a common node, and whose level and essential level are both at most j. By the definition of the essential level of an edge, each edge disappears through a contraction no later than in the stage following its insertion, so that the total cost of the algorithm, exclusive of that of the union-find data structure, comes to $O(m)$. On the other hand, although the insertion of an edge may be delayed relative to the original algorithm, every edge is present when it is needed, so that the modified algorithm is correct.

We now sketch how to construct the component tree in $O(nm)$ time when $w \geq n$. Again, the basic approach is as in the simple algorithm. Starting with a graph that contains the elements of V as vertices and no edges, we insert the edges in E in an order of nondecreasing levels into a graph H, keeping track of the transitive closure of H as we do so. The transitive closure is represented through the rows and columns of its adjacency matrix, each of which is stored in a single word as a bit vector of length n. It is easy to see that with this representation, the transitive closure can be maintained in $O(n)$ time per edge insertion. After

each insertion of all edges of a common level, we pause to compute the strongly connected components, contract each of these and insert a corresponding node in a partially constructed component tree. This can easily be done in $O(kn)$ time, where k is the number of vertices taking part in a contraction. Over the whole computation, the time sums to $O(nm)$.

2.2 Activating the Nodes in the Component Tree

In order to be activated at the proper time, each nonroot node y in T needs to place a "wakeup request" with its parent x. To this effect, each active node x in T, on level i, say, is equipped with a *calendar* containing a slot for each $(i-1)$-interval of simulated time during which x is active. The calendar of x is represented simply as an array of (pointers to) linked lists, each list containing entries of all children of x requesting to be woken up at the corresponding $(i-1)$-interval. Since the total number of tokens exchanged is $O(m)$, calendars of total size $O(m)$ suffice, and the calendar of a node x can be allocated when x becomes active.

The wakeup mechanism requires us to maintain $d[y]$ for each dormant node y in T with an active parent x; let us call such a node *pre-active*. We describe below how to compute $d[y]$ at the moment at which y becomes pre-active. Subsequent changes to $d[y]$, up to the point at which y becomes active, are handled as follows: Whenever $d[v]$ decreases for some vertex $v \in V$, we locate the single pre-active ancestor y of v (how to do this is also described below) and, if appropriate, move the entry of y in the calendar of its parent to a different slot (in more detail, the entry is deleted from one linked list and inserted in another).

We list the leaves in T from left to right, calling them *points*, and associate each node in T with the interval consisting of its leaf descendants. When a node becomes pre-active, it notifies the last point v in its interval of this fact—we will say that v becomes a *leader*. Now the pre-active ancestor of a point can be determined by finding the leader of the point, the nearest successor of the point that is a leader. In order to do this, we divide the points into *intervals* of w points each and maintain for each interval a bit vector representing the set of leaders in the interval. Moreover, we keep the last point of each interval permanently informed of its current leader. Since T is of depth $O(w)$ and the number of intervals is $O(n/w)$, this can be done in $O(n)$ overall time, and now each point can find its current leader in $O(\log w)$ time.

In order to compute $d[y]$ when y becomes pre-active, we augment the data structure described so far with a complete binary tree planted over each interval and maintain for each node in the tree the minimum d value over its leaf descendants. Decreases of d values are easily executed in $O(\log w)$ time, updating along the path from the relevant leaf to the root of its tree. When a segment of length r is split, we compute the minima over each of the new segments by following paths from a leaf to the root in the trees in which the segment begins and ends and inspecting the roots of all trees in between, which takes $O(\log w + r/w)$ time. Since there are at most m decreases and $n-1$ segment splits and the total

length of all segments split is $O(nw)$, the total time comes to $O(m \log w)$. This ends the proof of Theorem 1.

2.3 Extensions

If only $b \geq 2$ of the $w+1$ possible edge levels $0, \ldots, w$ occur in an input network $\mathcal{N}$ with n vertices and m nodes, we can solve SSSP problems in $\mathcal{N}$ in $O(n \log\log n + m \log b)$ time and $O(n+m)$ space. For this, the "search space" of the algorithm *BatchedSearch* should be taken to be the actual set of edge levels (plus, possibly, one additional level needed to ensure strong connectivity), and the activation of nodes in the component tree should use intervals of size b rather than w. This changes all bounds of $O(m \log w)$ in the analysis to $O(m \log b)$, and all other time bounds are $O(m + n \log\log n)$. This also takes care of the case $w > m$ that was ignored in Section 2.1.

As observed by Johnson [12], the APSP problem in a strongly connected network $\mathcal{N}$ with n vertices, m edges, edge lengths in $\{-2^w, \ldots, 2^w\}$, and no negative cycles can be solved with an SSSP computation in $\mathcal{N}$ and $n-1$ SSSP computations in an auxiliary network $\mathcal{N}'$ with n vertices, m edges, and edge lengths in $\{0, \ldots, n2^w\}$. The SSSP computation in $\mathcal{N}$ can be carried out in $O(nm)$ time with the Bellman-Ford algorithm [3,7]. The SSSP computations in $\mathcal{N}'$ can be performed with the new algorithm, but constructing the component tree for $\mathcal{N}'$ only once. Disregarding the construction of the component tree and the activation of its nodes, the new algorithm works in $O(m + n \log\log n)$ time. The node activation can be done within the same time bound by appealing to a decrease-split-minimum data structure due to Gabow [10]; this connection was noted by Thorup [21], who provides details. Since the component tree can be constructed in $O(nm)$ time, this proves the following theorem.

Theorem 2. *For all positive integers n, m and w with $w > \log n \geq 1$, all-pairs shortest-paths problems in networks with n vertices, m edges, edge lengths in the range $\{-2^w, \ldots, 2^w\}$, and no negative cycles can be solved in $O(nm + n^2 \log\log n)$ time and $O(n+m)$ space (not counting the output space) on a word RAM with a word length of w bits and the restricted instruction set.*

3 Shortest Paths in Undirected Networks

When the algorithm of the previous section is applied to an undirected network, it is possible to eliminate the bottlenecks responsible for the superlinear running time. As argued by Thorup [21], the node activation can be done in $O(n+m)$ overall time by combining the decrease-split-minimum data structure of Gabow [10] with the Q-heap of Fredman and Willard [9]. The second bottleneck is the construction of the component tree, for which we propose a new algorithm.

3.1 The Component Tree for Undirected Networks

In the interest of simplicity, we will assume that there are no edges of zero length. This is no restriction, as all connected components of G_0 can be replaced by single vertices in a preprocessing step that takes $O(n+m)$ time.

We begin by describing a simple data structure that consists of a bit vector indexed by the integers $1,\ldots,w$ and an array, also indexed by $1,\ldots,w$, of stacks of edges. The bit vector typically indicates which of the stacks are nonempty. In constant time, we can perform a push or pop on a given stack and update the bit vector accordingly. Using the MSB instruction, we can also determine the largest index, smaller than a given value, of a nonempty stack or, by keeping the reverse bit vector as well, the smallest index, larger than a given value, of a nonempty stack. In particular, treating the stacks simply as sets, we can implement what [11] calls a *neighbor dictionary* for storing edges with their levels as keys that executes each operation in constant time. Since only the level of a key is relevant to the component tree, in the remainder of the section we will assume that the length of an edge is replaced by its level and denote the resulting network by $\mathcal{N}'$.

Following Thorup [21], we begin by constructing a minimum spanning tree (MST) of $\mathcal{N}'$. Instead of appealing to the MST algorithm of Fredman and Willard [9], however, we simply use Prim's MST algorithm [6,13], which maintains a subtree T of $\mathcal{N}'$, initially consisting of a single node, processes the edges in $\mathcal{N}'$ one by one, and always chooses the next edge to process as a shortest edge with at least one endpoint in T. Aided by an instance of the dictionary described above, we can execute Prim's algorithm to obtain an MST T_{M} of $\mathcal{N}'$ in $O(m)$ time. We root T_{M} at an arbitrary node. The significance of T_{M} is that a component tree for T_{M} (with the original edge lengths) is also a component tree for $\mathcal{N}$.

The next step is to perform a depth-first search of T_{M} with the aim of outputting a list of the edges of T_{M}, divided into *groups*. Whenever the search retreats over an edge $e=\{u,v\}$ of length l, with u the parent of v, we want, for $i=1,\ldots,l-1$, to output as a new group those edges of length i that belong to the subtree of v, i.e., the maximal subtree of T_{M} rooted at v, and that were not output earlier. In addition, in order to output the last edges, we pretend that the search retreats from the root over an imaginary edge of length ∞. In order to implement the procedure, we could use an initially empty instance of the dictionary described in the beginning of the section and, in the situation above, push e on the stack of index l after popping each of the stacks of index $1,\ldots,l-1$ down to the level that it had when e was explored in the forward direction (in order to determine this, simply number the edges in the order in which they are encountered). Because of the effort involved in skipping stacks that, although nonempty, do not contain any edges sufficiently recent to be output, however, this would not work in linear time. In order to remedy this, we stay with a single array of stacks, but associate a bit vector with each node in T_{M}. When the search explores an edge $\{u,v\}$ of length l in the forward direction, with u the parent of v, the dictionary of v is initialized to all-zero (denoting an

empty set), and when the search retreats over $\{u, v\}$, the bit of index l is set in the bit vector of u, which is subsequently replaced by the bitwise OR of itself and the bit vector of v. Thus the final bit vector of v describes the part of the array of stacks more recent than the forward exploration of $\{u, v\}$, so that, when the search retreats over $\{u, v\}$, the relevant edge groups can be output in constant time plus time proportional to the size of the output. Overall, the depth-first search takes $O(n)$ time.

Define the length of a group output by the previous step as the common length of all edges in the group (their former level). We number the groups consecutively and create a node of level equal to the group length for each group and a node of level -1 for each vertex in V. Moreover, each group output when the search retreats over an edge e, except for the longest group output at the root, computes its parent group as the shortest group longer than itself among the group containing e and the groups output when the search retreats over e, and each vertex $v \in V$ computes its parent group as a shortest group containing an edge incident on v. This constructs a component tree for $\mathcal{N}$ in $O(n+m)$ time.

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1993.
2. R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan, Faster algorithms for the shortest path problem, *J. ACM* **37** (1990), pp. 213–223.
3. R. Bellman, On a routing problem, *Quart. Appl. Math.* **16** (1958), pp. 87–90.
4. B. V. Cherkassky, A. V. Goldberg, and C. Silverstein, Buckets, heaps, lists, and monotone priority queues, *SIAM J. Comput.* **28** (1999), pp. 1326–1346.
5. G. B. Dantzig, On the shortest route through a network, *Management Sci.* **6** (1960), pp. 187–190.
6. E. W. Dijkstra, A note on two problems in connexion with graphs, *Numer. Math.* **1** (1959), pp. 269–271.
7. L. R. Ford, Jr. and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.
8. M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM* **34** (1987), pp. 596–615.
9. M. L. Fredman and D. E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, *J. Comput. System Sci.* **48** (1994), pp. 533–551.
10. H. N. Gabow, A scaling algorithm for weighted matching on general graphs, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1985), pp. 90–100.
11. T. Hagerup, Sorting and searching on the word RAM, in Proc. 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1998), Lecture Notes in Computer Science, Vol. 1373, Springer, Berlin, pp. 366–398.
12. D. B. Johnson, Efficient algorithms for shortest paths in sparse networks, *J. ACM* **24** (1977), pp. 1–13.
13. R. C. Prim, Shortest connection networks and some generalizations, *Bell Syst. Tech. J.* **36** (1957), pp. 1389–1401.

14. R. Raman, Priority queues: Small, monotone and trans-dichotomous, in Proc. 4th Annual European Symposium on Algorithms (ESA 1996), Lecture Notes in Computer Science, Vol. 1136, Springer, Berlin, pp. 121–137.
15. R. Raman, Recent results on the single-source shortest paths problem, *SIGACT News* **28**:2 (1997), pp. 81–87.
16. R. Raman, Priority queue reductions for the shortest-path problem, in Proc. 10th Australasian Workshop on Combinatorial Algorithms (AWOCA 1999), Curtin University Press, pp. 44-53.
17. R. E. Tarjan, Efficiency of a good but not linear set union algorithm, *J. ACM* **22**, (1975), pp. 215–225.
18. M. Thorup, On RAM priority queues, in Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1996), pp. 59–67.
19. M. Thorup, Randomized sorting in $O(n \log \log n)$ time and linear space using addition, shift, and bit-wise boolean operations, in Proc. 8th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA 1997), pp. 352–359.
20. M. Thorup, Faster deterministic sorting and priority queues in linear space, Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998), pp. 550–555.
21. M. Thorup, Undirected single-source shortest paths with positive integer weights in linear time, *J. ACM* **46** (1999), pp. 362–394.
22. P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Inform. Process. Lett.* **6** (1977), pp. 80–82.
23. P. van Emde Boas, R. Kaas, and E. Zijlstra, Design and implementation of an efficient priority queue, *Math. Syst. Theory* **10** (1977), pp. 99–127.
24. P. D. Whiting and J. A. Hillier, A method for finding the shortest route through a road network, *Oper. Res. Quart.* **11** (1960), pp. 37–40.