Nondecreasing Paths in a Weighted Graph or: How to Optimally Read a Train Schedule

Virginia Vassilevska*

Abstract

A travel booking office has timetables giving arrival and departure times for all scheduled trains, including their origins and destinations. A customer presents a starting city and demands a route with perhaps several train connections taking him to his destination as early as possible. The booking office must find the best route for its customers. This problem was first considered in the theory of algorithms by George Minty [Min58], who reduced it to a problem on directed weighted graphs: find a path from a given source to a given target such that the consecutive weights on the path are nondecreasing and the last weight on the path is minimized. Minty gave the first algorithm for the single source version of the problem, in which one finds minimum last weight nondecreasing paths from the source to every other vertex. In this paper we give the first *linear* time algorithm for this problem. We also define an all pairs version for the problem and give a strongly polynomial truly subcubic algorithm for it.

1 Introduction

We consider the problem of planning a train trip so that we arrive at our destination as early as possible. The travel booking office has a timetable with departure and arrival times for all scheduled trains, together with their origins and destinations. From this data, we wish to extract the best choice of train connections possible, given our start city and preferred departure time. While this is an old problem, it is of course still interesting and relevant today. To our knowledge, the problem was first studied in the theory of algorithms in 1958 in a paper by Minty [Min58]. Minty reduced it to a problem on edge-weighted graphs as follows. The train stops are mapped to vertices. If there is a train scheduled from city A to city B leaving A at time t_1 and arriving at B at time t_2, one creates a vertex v for the scheduled train and adds an edge from A to v with weight t_1 and an edge from v to B with weight t_2. A train trip from a start S to a destination T is only valid if one can take each consecutive train on the trip without missing a connection. In particular, this means that for any two consecutive trains, the arrival time of the first one must be no later than the scheduled departure time of the second one. In the graph this corresponds to a *nondecreasing* path from S to T, *i.e.* a path, the weights on which form a nondecreasing sequence. In order to find the route which gets us to T at the earliest possible time, we must find a nondecreasing path minimizing the last edge weight on the path.

Minty [Min58] also defined a generalization of the problem: given an edge weighted graph $G = (V, E, w)$ and a source $s \in V$, report for every $t \in V$, the minimum last edge weight on a

*Computer Science Department, Carnegie Mellon University, Pittsburgh PA. Supported by a Computer Science Department PhD Scholarship. Email: virgi@cs.cmu.edu

nondecreasing path from s to t (∞ if such a path does not exist). We call this the single source minimum nondecreasing paths problem (SSNP).

SSNP is a variant of the popular single source shortest paths problem (SSSP). In SSNP the sum operation from SSSP is replaced by an inequality ($\leq$) constraint. SSNP was studied alongside SSSP and many of the algorithms for the two problems are quite similar. Minty gave an algorithm for SSNP running in $O(mn)$ time, where $n = |V|$ and $m = |E|$. In the following year, Moore [Mo59] gave a similar $O(mn)$ time algorithm for SSSP and showed how to transform it into one for SSNP. Incidentally, Moore's algorithm strikingly resembles the Bellman-Ford algorithm [Bell58, Ford56].

Nowadays much faster solutions are known for both problems. Dijkstra's algorithm [Dij59] using Fredman and Tarjan's implementation [FT87] solves SSSP with nonnegative weights in $O(m+n \log n)$ time. A folklore modification of that same algorithm also gives an $O(m + n \log n)$ time algorithm for SSNP in the comparison-based model of computation.

The restriction of SSNP for nonnegative weights has been studied in the algorithmic applications community under the name the *earliest arrival problem*. It seems that most of the research has been devoted to analyzing different graph representations and various heuristics to optimally implement Dijkstra's algorithm (*e.g.* [Brod04, Pyr07, Schu02]). Yet no asymptotically better algorithm for the problem has been presented, to our knowledge.

In the word RAM (transdichotomous) model of computation, Thorup [Tho99] gave a linear time algorithm for SSSP on *undirected* graphs with nonnegative weights. In this paper we how that in the word RAM model of computation there is a *linear* time algorithm for SSNP, even for *directed* graphs with *arbitrary* weights. In contrast, there is no known linear time algorithm for SSSP on weighted directed graphs. Our algorithm can be implemented to run in $O(m \log \log n)$ time in the comparison based model of computation. This is better than Dijkstra's algorithm for $m = o(n \log n / \log \log n)$.

We also consider an all pairs version of the minimum nondecreasing paths problem. In the all pairs nondecreasing paths problem (APNP) one is given an edge-weighted graph and one must return for all pairs of vertices s and t the minimum last weight on a nondecreasing path from s to t. We give a truly subcubic (in n) algorithm for APNP. Our runtime is $O(n^{2.896})$. Although some truly subcubic algorithms (*i.e.* $O(n^{3-\delta})$ for constant $\delta > 0$) are known for certain interesting restrictions of the all pairs shortest paths (APSP) problem [Sei95, GM97, SZ99, Zwi02, Chan07], there is no known truly subcubic algorithm for APSP. Recent work [VWY07] has shown that the related all pairs maximum bottleneck paths problem (APBP) is solvable in truly subcubic time. In retrospect, one can say that the algorithm for APBP in [VWY07] was based on solving a special case of APNP. In particular, one can show that APNP is at least as hard as APBP. Karger, Koller, and Phillips [KKP93] considered a particular type of algorithm, the "path comparison" algorithm, which is restricted in that it accesses the edge weight function only by comparing the weights of two paths in the graph. Many algorithms for APSP and APBP, such as Floyd's and Dijkstra's (when run on all vertices), have this property. Karger, Koller and Phillips show that any "path comparison" algorithm requires $\Omega(n^3)$ time to compute APSP and APBP. Since APBP can be reduced to APNP without violating the "path comparison" model, this lower bound applies to APNP as well.

APNP is also at least as hard as finding dominating pairs in $\mathbb{R}^n$. The best known algorithm for the latter problem is by Matousek [Mat91] and runs in $O(n^{2.688})$. Any improvement to this running time would imply better algorithms for many problems such as computing the most significant bits of the distance product [VW06].

2 Preliminaries

Throughout this paper, when we consider a graph $G = (V, E)$ we let $m = |E|$ and $n = |V|$. We identify V with the integers $\{1, \ldots, n\}$, so that each vertex is mapped to an integer in a one-to-one fashion. G is always a directed graph. It is always connected, so that $m \geq n - 1$. We will use the terms *vertex* and *node* interchangeably.

As is typical, we define $\omega \geq 2$ to be the smallest real number such that matrix multiplication over a ring can be done using $O(n^{\omega+\varepsilon})$ arithmetic operations, for all $\varepsilon > 0$. The best known upper bound for ω is 2.376, given by Coppersmith and Winograd [CW90].

We use a special matrix product in our algorithm for APNP.

Definition 2.1 *Given two $n \times n$ matrices A and B over a totally ordered set, the* $(\min, \leq)$ product *matrix $C = A \circledcirc B$ is defined as*

$$C[i,j] := \begin{cases} \min_k \{B[k,j] \mid A[i,k] \leq B[k,j]\} & \text{if } \exists k,\ A[i,k] \leq B[k,j], \\ \infty & \text{otherwise.} \end{cases}$$

The ∞ values we consider can always be replaced by suitably large finite values. We ignore this technicality in the rest of the paper.

Model of computation. In the SSNP portion of the paper we employ the word RAM model of computation with word size $w = \Omega(\log n)$. In the APNP portion we use the standard addition-comparison computational model, along with random access to registers. In the APNP algorithms of this paper, the only operations we perform on real numbers are comparisons.

3 Single Source Nondecreasing Paths

The best known algorithm for SSNP is the following folklore modification of Dijkstra's algorithm.

Algorithm `A`: Initialize $d[s] = -\infty$ and $d[v] = \infty$ for all $v \neq s$. Extract the vertex u minimizing $d[u]$, setting it as completed. Then for all uncompleted neighbors v of u, if $w(u, v) \geq d[u]$, set $d[v] = \min\{d[v], w(u, v)\}$. Recurse on the remaining uncompleted vertices until there are no more.

It is not hard to verify that at the completion of the algorithm $d[v]$ contains the minimum last weight edge on a nondecreasing path from s to v. Using Fibonacci heaps this algorithm runs in $O(m + n \log n)$.

We mention one more algorithm which resembles DFS. It takes a node v and a weight wt corresponding to an edge into v. Initially, $(v, wt) = (s, -\infty)$. For each vertex v, a value $d[v]$ is stored as before. Initially, $d[v] = \infty$ for $v \neq s$ and $d[s] = -\infty$.

Algorithm `B`(v, wt): Go through the edges (v, u) going out of v, and if $w(v, u) \geq wt$, remove (v, u) from the graph, set $d[u] = \min\{d[u], w(v, u)\}$ and call `B`$(u, w(v, u))$.

It can be verified that algorithm `B` also computes SSNP correctly. Its running time is $O(mn)$, but this can be improved by using good data structures. The bottleneck of the algorithm is going through the outgoing edges of v every time v is accessed. Yet we only need to access those edges whose weight is greater than the incoming weight. If we first store a successor search data structure

at every vertex to maintain the outgoing edge weights, we could directly access only the necessary edges at each access of v. For instance, if before running the algorithm, we insert the outgoing edges of each vertex in a binary search tree achieving $O(\log n)$ time per operation, we can reduce the running time to $O(m \log n)$. [1]

Our idea is to employ the right data structures so that interleaving algorithms `A` and `B` yields a linear time algorithm for SSNP. In particular we prove the following.

Theorem 3.1 *In the word RAM model of computation there is a linear time algorithm for SSNP on directed weighted graphs.*

Q-Heaps. The data structure which we use to store the weights of the edges going out of each node is a successor search data structure based on the Q-Heaps of Fredman and Willard [FW94]. As pointed out by Thorup [Tho99], the Q-Heaps result can be summarized as follows.

Lemma 3.1 *([FW94]) Given $O(n)$ preprocessing time and space for the construction of tables, there are data structures called Q-Heaps maintaining a family $\{S_i\}$ of word-sized integers multisets, each of size at most $O(\log^{1/4} n)$, so that each of the following operations can be done in constant time: insert x in S_i, delete x from S_i, and find the rank of x in S_i, returning the number of elements of S_i that are stricly smaller than x.*

Using a 4-level indirection (bucketing elements similarly to the AF-heaps of Fredman and Willard) and a small modification of the above we can obtain a data structure for successor queries which can dynamically maintain $O(\log n)$ elements with $O(1)$ amortized time per operation.

Corollary 3.1 *Given $O(n)$ preprocessing time and space for the construction of tables, there are data structures maintaining a family (U_i) of disjoint sets, each of size at most $O(\log n)$, so that each of the following operations can be done in constant amortized time: insert x in U_i, delete x from U_i, and given some integer y (not necessarily in $\bigcup_i U_i$) find $x \in U_i$ such that $x \geq y$ and x is closest to y from all elements of U_i.*

We call the data structures presented in the corollary FW-heaps (for Fredman and Willard).

Linear Time Algorithm for SSNP. This section is devoted to proving Theorem 3.1. For our linear time algorithm we first transform every edge weight $w(u, v)$ into a weight $w'(u, v) = Nw(u, v) + v$, where N is the smallest power of 2 greater than n. As noted in the preliminaries vertex v is identified with a unique integer from 1 to n. These new weights allow us to distinguish between edges with the same weight. The size of the integers in consideration increases only by $O(\log n)$, as in the binary representation we simply concatenate $w(u, v)$ and v. From now on we consider both w' and w weights where the w weights are assumed to have $\lceil \log n \rceil$ zero bits concatenated at the end. The w' weights are used solely by the FW-heaps data structure.

The idea of the algorithm is to handle *low* and *high* outdegree nodes differently. Define a *low* degree node to be one with outdegree at most $\log n$. A *high* degree node has outdegree at least $\log n$. Note that there are at most $m/\log n$ high degree nodes. We put all high degree nodes into a Fibonacci heap F with weights ∞. We insert s in F with weight $-\infty$. The source s is treated as a high degree node. Its key in F will never be decreased and it will be extracted from F first. We create FW-heaps maintaining the sets of outgoing edge weights for each low degree vertex. In

[1] Alternatively, one can presort the outgoing edges in nonincreasing order and then access this list when a node is processed.

particular, for a low degree vertex v we store each edge (v, u) with key $w'(v, u)$. The FW-heaps data structure allows us, when given any incoming edge, to extract the successor edge in the sorted order of the outgoing edges in (amortized) constant time. Since insertions also take constant amortized time, preparing these FW-heaps takes $O(m)$ time overall.

The nondecreasing paths algorithm works in stages. First an algorithm `AlgoLow` just like algorithm `B` is run on the low degree nodes starting from the neighbors of the source until no more low degree nodes can be reached via nondecreasing paths. Algorithm `AlgoLow` can visit nodes more than once, but no edge is accessed more than once since edges are removed from the graph once they are visited. During the run of the algorithm, every reached high degree node is updated in the Fibonacci heap F just like in algorithm `A`, *i.e.* if the edge taken to reach it has a smaller weight than the value stored in F, the value is replaced with this weight.

When no more low degree nodes can be reached, just as in algorithm `A`, the minimum weight high degree node is extracted from the Fibonacci heap and a new run of algorithm `AlgoLow` is started from it. We call this part of our algorithm `AlgoHigh`. This process alternates between `AlgoLow` and `AlgoHigh` until we have either gone through all edges, or the Fibonacci heap is empty.

Let M be the set of visited nodes, initially $\emptyset$. Let for a low degree node v, N_v be the subset maintained in the FW-heaps data structure storing its incident out-edges. For each $t \in V$ and weight w, $N_t.successor(w)$ returns the lexicographically smallest neighbor v of t for which $w(t, v) \geq w$ and there is no other neighbor u for which $w(t, v) > w(t, u) \geq w$. Let for any vertex v, $d[v]$ represent the current miminum weight of an edge used to get to v. For the current unvisited high degree nodes, $d[v]$ is stored in the Fibonacci heap F. Recall that at the start, $d[s] = -\infty$ and $d[v] = \infty$ for $v \neq s$. Let for every vertex $v \neq s$, $\pi[v]$ be the predecessor of v on the current best nondecreasing path from s to v. The initial value is $\pi[v] =$`null` for every v; $\pi[v]$ remains `null` if v is not reachable from s via a nondecreasing path. Pseudocode for `AlgoLow`, `AlgoHigh` and the final algorithm for SSNP is given in Figure 1.

Running time. We first handle the getMin calls on the Fibonacci heap, because they are the most expensive unit operations. The number of elements in F is at most $m/\log n$ so there can be at most $m/\log n$ getMin calls. Their overall cost is thus $O(\log n \times m/\log n) = O(m)$. Other than that, each edge is examined at most once in constant amortized time since FW-heaps allow each edge to be found quickly. When an edge is looked at, a constant number of other $O(1)$ time operations are performed. Thus the overall time is linear: $O(m)$. We note that if instead of FW-heaps one uses binary search trees, then the algorithm would run in $O(m \log\log n)$ time in the comparison model.

Correctness. We need to show that at the end of the algorithm, a vertex v is in M if and only if there is a nondecreasing path from s to v, and that furthermore $d[v]$ is the minimum weight of the last edge on a nondecreasing path from s to v. We prove three claims which imply the final proof of correctness.

Claim 1 *If a node v is in M, then it can be reached by a nondecreasing path, and $d[v] = w(\pi(v), v)$ is the weight of a last edge of some path from s to v.*

Proof. Follows by induction and because the value of a node in F gives a weight of an edge which is the end of a nondecreasing path from s to the node. □

The last two claims show that if a node $z \neq s$ can be reached by a nondecreasing path from s, then it is in M. For any nondecreasing path from s to z with last edge weight w, $d[z] \leq w$. Our

```
AlgoLow(t,w):
   Add t to M
   If t is a low degree node:
     u_0<- N_t.successor(w)
     do until u_0 = null:
       remove (t,u_0) from the graph
       remove u_0 from N_t
       if w(t,u_0)<d[u_0] then
         d[u_0] = w(t,u_0)
         pi[u_0]=t
         AlgoLow(u_0, w(t,u_0))
       u_0<- N_t.successor(w(t,u_0))
   Else t is a high degree node:
     If w<d[t], then
       F.decreaseKey(t, w).
```

```
AlgoHigh():
   z<- F.getMin()
   for all neighbors u of z:
     remove (z,u) from graph
     if w(z,u)>=d[z], then
       if w(z,u)<d[u] then
         d[u] = w(z,u)
         pi[u]=z
         AlgoLow(u, w(z,u))

findSSNP(s):
   M={}; d[s]=-infty
   d[v]= infty for all v!=s
   PrepareDataStructures()
   While F and E are nonempty:
     AlgoHigh()
```

Figure 1: The `AlgoLow` and `AlgoHigh` algorithms handle low and high degree nodes respectively. The final algorithm FindSSNP finds the minimum nondecreasing paths from s.

algorithm maintains an invariant that for any $v \neq s$, $d[v] = w(\pi(v), v)$. Hence our proofs disregard $\pi(v)$. We note that using the π values one can recover actual minimum nondecreasing paths from s to any vertex t in time linear in their lengths.

Claim 2 *Consider a run of algorithm* `AlgoLow` *starting from a high degree vertex z with weight $d[z]$ and finishing when no more vertices can be reached via a nondecreasing path using only low degree vertices. If v is reachable from z via a path of only low degree vertices, then at the end of this run $d[v]$ is at most the minimum last edge weight on any nondecreasing path from z to v with first edge at least $d[z]$.*

Proof. Let us first prove the claim for low degree vertices. This setting is almost the same as running algorithm `B` with z as a source and ignoring edges out of z with weights smaller than $d[z]$. The only real difference is that the algorithm might never recurse on some low degree vertices v reachable from z via such a nondecreasing path because their $d[v]$ values are already smaller or equal to the last edge weight of any nondecreasing path from z to v with first weight $\geq d[z]$. However, this can only happen if there is a path from s to v which is at least as good as any nondecreasing path from z to v with first weight $\geq d[z]$. Hence v need not be considered in the search. Furthermore, any low degree paths going through such a v would have been considered when the best path to v was traversed. Hence the claim holds for all low degree vertices. Because the $d[v]$ values are also updated when a high degree vertex can be reached from z, the claim also holds for high degree vertices reachable from z via a low degree nondecreasing path with first weight at least $d[z]$. □

The remainder of the argument bears similarities to the proof of correctness for Dijkstra's SSSP algorithm.

Claim 3 *At any point of our algorithm let S contain all high degree vertices which have already been*

extracted from F. For each high degree node $v \in S$ the value of $d[v]$ when v is extracted from F is equal to the minimum last edge weight of a nondecreasing path from s to v. Furthermore, for all high degree nodes $v \notin S$, $d[v]$ is the minimum last edge weight of a nondecreasing path using only vertices in S and low degree vertices in M.

Proof. We do induction on the size of S. The base case is when $|S| = 1$. Then S consists only of s, and $d[s]$ is $-\infty$. Inductive step: suppose the statement holds for $|S| < k$. After the $k-1$st high degree node z is added to S, `AlgoLow` is called on all of z's neighbors whose edge weight is greater than $d[z]$. Until the kth high degree node is to be added, all low degree nodes reachable from z by a nondecreasing path with edge weights greater than $d[z]$ are added to M. Their $d[v]$ value is at most the the best value for paths first using vertices in S and low degree nodes in M, then going through z and finally using only low degree nodes. Moreover, for every high degree $v \notin S$, if there is a nondecreasing path from s to v using only nodes from S and low degree nodes from M, going through z and using only low degree nodes, then the ending edge weight of that path is at least $d[v]$.

Suppose now that there is a nondecreasing path P from s to v using vertices in S, low degree vertices in M and z such that there is some high degree node (from S) between z and v on P. Let $y \in S$ be the last such high degree node. By the induction hypothesis when y was extracted from F it had the minimum value Y of a last edge on any nondecreasing path from s to a high degree node. After y was extracted, all low degree nodes reachable from it with nondecreasing paths with edge weights $\geq Y$ were explored (either by `AlgoLow` on y or by a previous run of `AlgoLow`). In particular, if v has a low degree predecessor in P, then `AlgoLow`(v, wt) was run, where wt is at most the minimum end weight of a nondecreasing path from y with weights $\geq Y$. Otherwise, y is v's predecessor in P. Either way, there is a path from s to v using only high degree nodes in $S \setminus \{z\}$ and low degree nodes in M of last edge weight at most that of P. And hence P does not need to be considered and $d[v]$ is indeed the minimum last edge weight of a nondecreasing path using only vertices in S and low degree vertices in M.

Now we need to show that when the kth high degree node v is added to S, $d[v]$ is equal to the minimum last edge weight of a nondecreasing path from s to v. Suppose not and let $Q := s \to y_1 \to \ldots \to y_{k-1} \to v$ be a minimum nondecreasing path from s to v that contains high degree vertices not in S. Let y_i be the first high degree vertex on Q which is not in S. $d[y_i]$ is the minimum last edge weight of a nondecreasing path using high degree vertices only from S. Because y_i appears before v in Q, we must have that $d[y_i] \leq w(y_{i-1}, y_i) \leq w(y_{k-1}, v)$. But since v is to be extracted from F before y_i, $d[v] \leq d[y_i]$ and hence $d[v] \leq w(y_{k-1}, v)$ and there is a path Q' from s to v using high degree vertices only from S such that the last edge weight of Q' is at most that of Q, making $d[v]$ equal to the minimum value of the last edge weight of a nondecreasing path from s to v. The induction step is completed. □

Claim 3 ensures that the values of the high degree nodes are minimized. By Claim 2 all values of low degree nodes reachable by nondecreasing paths are also minimized. Inductively, at the end of the algorithm all $d[\cdot]$ and $\pi[\cdot]$ values are correct.

4 All Pairs Nondecreasing Paths

In the all pairs version of the nondecreasing paths problem, we are interested in an algorithm with running time which is subcubic in terms of n, preferably $O(n^{3-\delta})$ for some constant $\delta > 0$. Observe that obtaining a cubic running time is easy - just run the SSNP algorithm on all vertices.

We begin with a motivating example. Consider the variant of APNP in which the weights are on the vertices instead of on the edges. In this case, the last weight on a path from s to t is defined as the weight of the node just before t on the path, or $-\infty$ if $s = t$. This is clearly a restriction of APNP since we can convert the node weights into edge weights by setting the weight of an edge (u, v) to the weight of u. A subcubic algorithm for vertex weighted APNP is as follows: first go through the graph and create a restricted adjacency matrix A such that $A[i, j] = 1$ if $w(i) \leq w(j)$ and $(i, j) \in E$, and $A[i, j] = 0$ otherwise. Compute the transitive closure of A in $O(n^\omega)$ time. This gives a matrix T such that $T[i, j] = 1$ if and only if there is a nondecreasing path from i to j with the property that the last weight on the path is $\leq w(j)$. Now, create the actual adjacency matrix B of the graph such that $B[i, j] = 1$ if $(i, j) \in E$ and $B[i, j] = 0$ otherwise. W.L.O.G. assume that the vertices are sorted by their weights so that $i \leq j$ iff $w(i) \leq w(j)$. Now use the minimum witness algorithm for boolean matrix multiplication [KL05] to find for every pair of vertices i, j for which $(TB)_{ij} = 1$ the minimum value $w(k)$ such that $T_{ik}B_{kj} = 1$. The overall algorithm runs in $O(n^{2.575})$ time. Furthermore, vertex weighted APNP is at least as hard as the problem of finding minimum witnesses of boolean matrix multiplication, since the latter is just the restriction of vertex weighted APNP on tripartite graphs with $-\infty$ weights in the first partition. Hence, the above runtime cannot be improved unless there is a better algorithm for minimum witnesses.

Now we consider the more general edge weighted case. We need the following theorem, the proof of which can be obtained by a slight modification of a result in previous work [VWY07].

Theorem 4.1 ([VWY07]) *Given $n \times n$ matrices A and B, the $(\min, \leq)$ product matrix C as in Definition 2.1 can be computed by a strongly-polynomial algorithm in $O(n^{2+\frac{\omega}{3}}) = O(n^{2.792})$ time. Moreover, for each pair i, j, if $C[i, j] < \infty$, the algorithm returns a witness k such that $\min_k\{B[k, j] \mid A[i, k] \leq B[k, j]\} = C[i, j]$.*

Before we can give our final algorithm, we give two useful lemmas.

Lemma 4.1 *Given an edge weighted directed graph $G = (V, E, w)$ and a vertex $s \in V$ there is an algorithm which in $O(n^2 \log n)$ time computes for all pairs of vertices i, j the minimum last edge weight of a nondecreasing path from i to j passing through s. Moreover, the algorithm can give predecessor/successor information for each node so that the best path can be recovered in time linear in its length.*

Proof. We first compute for every vertex $t \in V$ the minimum weight last edge on a nondecreasing path from t to s. The algorithm is very similar to algorithm `B` presented in the previous section. However, here we consider the graph with all edge directions reversed. We create for each vertex a balanced binary search tree data structure for the outedges and insert them according to their weights. This process takes $O(m \log n)$ time. In sorted order of the edges out of s start a search just as in algorithm `B`. Here instead of keeping track of the best last edge into a vertex t, record the first edge *out of* s for which one can reach t via a *nonincreasing* path. Similar to before, for every vertex u and incoming edge weight w accessing the predecessor weight outgoing edge takes $O(\log n)$ time. Hence the overall running time of this is $O(m \log n)$.

Now we again consider the original graph. For every vertex u we create a balanced binary tree on $n - 1$ leaves as follows. The leaves from left to right contain the edges out of s sorted by their weight. Each node in the tree contains two numbers. The first number of a tree node t represents the minimum over all weights of edges stored in the subtree rooted at t. This number can be filled in at the creation of the tree which takes $O(n \log n)$ for each vertex. The second number of a tree

node t represents the minimum weight of a last edge on a nondecreasing path from s to u beginning at an edge stored in the subtree rooted at t. The second number is initialized as ∞ at creation.

In *reverse* sorted order out of s we start another search for nondecreasing paths as in algorithm B. When we reach a vertex u using first edge (s, j), access the leaf containing (s, j) in the tree for u. If the current stored minimum end weight is greater than the one used to access u in the current path, update the stored value. Go up the tree to the root updating minimum weights if necessary.

At the end of this process, for each node u, take the minimum last edge weight w over all nondecreasing paths from u to s (computed in the first part of the algorithm). For each node v, go down the binary search tree for v from the root to the leaf holding the first edge with weight w. Look at all right children of the nodes on this path together with the leaf ending the path. Find the minimum out of the second numbers of these nodes. This finds the minimum last edge weight of a nondecreasing path from u to v going through s. If this weight corresponds to some edge (x, u), one can also set the predecessor of v on the path from u to be x. The overall running time of the algorithm is $O(n^2 \log n)$. □

The next lemma is used in many APSP algorithms, *e.g.* [Zwi02, GM97, Chan07]. One way to prove the lemma is by random sampling, another is by the greedy approximation to hitting set.

Lemma 4.2 *Given a collection of N subsets of $\{1, \ldots, n\}$, each of size ℓ, one can find in $O(N\ell)$ time a set of $\frac{n \log n}{\ell}$ elements of $\{1, \ldots, n\}$ hitting every one of the subsets.*

We are now prepared to combine Theorem 4.1 and Lemmas 4.1 and 4.2 into an algorithm for APNP.

Theorem 4.2 *There is a strongly polynomial algorithm computing APNP in $O(n^{\frac{15+\omega}{6}} \log n) = O(n^{2.896})$ time.*

Proof. Let ℓ be a parameter to be set later. Let A be the matrix for which $A[i, i] = -\infty$, and if $i \neq j$, $A[i, j] = w(i, j)$ for $(i, j) \in E$ and $A[i, j] = \infty$ otherwise. First set $B = A$ and then repeat $B = B \otimes A$ for $\ell - 1$ times. This gives for each i, j the minimum weight of a last edge on a nondecreasing path from i to j of length at most ℓ. The running time for this portion of the algorithm is $O(\ell n^{2+\frac{\omega}{3}})$.

When running the above, keep for each pair of vertices i, j, an actual minimum nondecreasing path of length $\leq \ell$ found between i and j. A subset L of these paths are of length exactly ℓ (otherwise we have found all minimum nondecreasing paths). Using Lemma 4.2, in $O(n^2\ell)$ time obtain a set S of $O(\frac{n \log n}{\ell})$ vertices hitting each of the $\leq n^2$ paths in L. For each $s \in S$ run the algorithm from Lemma 4.1 to obtain for every pair i, j the minimum last edge on a nondecreasing path from i to j going through S. This part of the algorithm takes $O(n^2\ell + \frac{n^3 \log^2 n}{\ell})$ time.

Taking the minimum of the above two path values for each pair gives the result. The running time is minimized when $\ell = n^{\frac{3-\omega}{6}} \log n$ and is $O(n^{\frac{15+\omega}{6}} \log n)$. We note that our algorithm above is strongly polynomial because the only operations we perform on the weights are comparisons. □

5 Conclusion

We have provided the first linear time algorithm for the single source minimum nondecreasing paths problem and a truly subcubic strongly polynomial algorithm for its all pairs version. As our linear time algorithm exploits the power of the RAM, one interesting open question is whether there

exists a linear time algorithm for SSNP in the comparison based model of computation. The current best running time in this model is $O(\min\{m + n \log n, m \log \log n\})$.

A slight variant of the problem presented in this paper is to find nondecreasing paths of minimum *weight sum*. Tarjan [Tar07] has shown that the single source version of this problem can be solved in $O(m \log n)$ time, even in the presence of negative weight edges. In contrast, the best known running time for SSSP in the presence of negative edge weights is $O(m\sqrt{n} \log w_{\max})$ by Goldberg [Gold93] (where $w_{\max} = \max_{e \in E} |w(e)|$). It is an interesting question whether the sum variant of SSNP also has a linear time algorithm, or whether sorting edges is necessary.

6 Acknowledgments

I would like to thank Alexei Kitaev from whom I first learned about the single source nondecreasing paths problem. I would also like to thank Manuel Blum for asking whether a linear time algorithm exists for SSNP, and my advisor Guy Blelloch, as well as Bob Tarjan for communicating with me on the topic. Last but not least, I am really indebted to Ryan Williams for his continued support and insightful suggestions on improving the quality of the paper.

References

[Bell58] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1), 87–90, 1958.

[Brod04] G. Brodal and R. Jacob. Time-dependent networks as models to achieve fast exact timetable queries. In *Proceedings of ATMOS Workshop 2003*, Electronic Notes in Theoretical Computer Science 92, 2004.

[Chan07] T. M. Chan. More Algorithms for All-Pairs Shortest Paths in Weighted Graphs. In *Proc. of STOC*, 2007.

[CW90] D. Coppersmith, S. Winograd, Matrix multiplication via arithmetic progressions. *J. Symbolic Computation* 9(3):251–280, 1990.

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. *Numer. Math.*, 269–271, 1959.

[Ford56] L. R. Ford. Network flow theory. *Technical Report* Paper P-923, The Rand Corporation, Santa Monica, 1956.

[FT87] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM* 34(3):596–615, 1987.

[FW94] M. L. Fredman and D. E. Willard. Transdichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *JCSS* 48:533–551, 1994.

[GM97] Z. Galil and O. Margalit. All pairs shortest paths for graphs with small integer length edges. *JCSS* 54:243–254, 1997.

[Gold93] A. Goldberg. Scaling algorithms for the shortest paths problem. In *Proc. of SODA*, 222–231, 1993.

[Han06] Y. Han. An $O(n^3(\log\log n/\log n)^{5/4})$ Time Algorithm for All Pairs Shortest Paths. In *Proc. of ESA*, Springer-Verlag LNCS 4168:411–417, 2006.

[KKP93] D. Karger, D. Koller, and S. Phillips. Finding the Hidden Path: Time Bounds for All-Pairs Shortest Paths. *SIAM J. Computing* 22(6):1199–1217, 1993.

[KL05] M. Kowaluk and A. Lingas. LCA queries in directed acyclic graphs. In *Proc. of ICALP*, Springer-Verlag LNCS 3580:241–248, 2005.

[Mat91] J. Matousek. Computing dominances in E^n. *Information Processing Letters* 38(5):277–278, 1991.

[Min58] G. J. Minty. A Variant on the Shortest-Route Problem, *Operations Research*, 6(6):882–883, 1958.

[Mo59] E. F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching*, 285–292, 2-5 April 1957, Part II, 1959.

[Pyr07] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient models for timetable information in public transportation systems. *J. Exp. Algorithmics* 12, 2007.

[Schu02] F. Schulz, D. Wagner, and C. D. Zaroliagis. Using Multi-level Graphs for Timetable Information in Railway Systems. In *Revised Papers From the 4th international Workshop on Algorithm Engineering and Experiments*. D. M. Mount and C. Stein, Eds. Lecture Notes In Computer Science 2409, 43–59, 2002.

[Sei95] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *JCSS* 51:400–403, 1995.

[SYZ07] A. Shapira, R. Yuster and U. Zwick. All-Pairs Bottleneck Paths in Vertex Weighted Graphs. In *Proc. of SODA*, 978–985, 2007.

[SZ99] A. Shoshan and U. Zwick. All Pairs Shortest Paths in Undirected Graphs with Integer Weights. In *Proc. of FOCS*, 605–614, 1999.

[Tar07] R. E. Tarjan, Personal communication.

[Tho99] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM* 46(3):362–394, 1999.

[VW06] V. Vassilevska and R. Williams. Finding a maximum weight triangle in $n^{3-\delta}$ time, with applications. In *Proc. of STOC*, 225–231, 2006.

[VWY07] V. Vassilevska, R. Williams and R. Yuster. All-Pairs Bottleneck Paths For General Graphs in Truly Sub-Cubic Time. In *Proc. of STOC*, 2007.

[Zwi02] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *JACM* 49(3):289–317, 2002.