A summary of shortest-paths results

Rajeev Raman

December 1996

Abstract

This note attempts to summarize the best of the currently existing theoretical results on the single-source shortest paths problem for graphs with non-negative edge weights. The main purpose of this note is to make available the statements of all of these results, as well as the building blocks they use, together in one document. We also note some new theoretical implications of a recent result due to Cherkassky, Goldberg and Silverstein (1996).

1 Introduction

There has been a recent renewal of interest in the theory of data structures for the single-source shortest paths problem on graphs with non-negative edge weights. The results combine elements from classical data structures, data structures which use bit-level parallelism and bucketing data structures designed specifically for this problem. The aim of this note is to summarize the best known theoretical results for this problem and to pin-point the ingredients used by these, with the hope that the techniques used may somehow be unified, simplified or used to get further theoretical improvements in complexity of this problem. This note also points out that a recent algorithm due to Cherkassky, Goldberg and Silverstein (1996) actually gives better theoretical bounds for this problem than claimed by the authors.

All known algorithms for this problem are based on the approach given by Dijkstra, with improvements concentrating on the data structure used to implement Dijkstra's algorithm. The data structure has to support the following operations on an (initially empty) collection of items, each of which is associated with a key drawn from an ordered universe:

insert(i, x): Add item i to the collection with key x.

delete(i): Delete item i from the collection.

delete-min(): Returns the item with smallest key from the collection and deletes it from the collection.

decrease-key(i, x): Sets the key of item i to x; x is assumed to be no greater than the current key value of item i. Note that decrease-key can always be implemented as a delete followed by an insert.

Dijkstra's algorithm, when run on a graph with n vertices and m edges, performs a sequence of n insert, n delete-min and at most m decrease-key operations. Since, for all

but the sparsest graphs, m is asymptotically bigger than n, we would like to support the decrease-key operation cheaply (in $O(1)$ time if possible). We distinguish between two kinds of data structures, depending on whether or not the decrease-key operation is inexpensive:

- A *priority queue (PQ)* supports the operations insert, delete and delete-min, and does not support decrease-key significantly more efficiently than a delete followed by an insert. (Note that a data structure which supports delete and insert in $O(1)$ time each would still be called a PQ by this definition, even though it would support decrease-key in $O(1)$ time!)
- An *F-heap* supports the insert, delete, delete-min and decrease-key operations, and a decrease-key is significantly cheaper than a delete followed by an insert.

A sequence of operations is called *monotone* if the values returned by successive delete-mins are non-decreasing. Dijkstra's algorithm performs a monotone sequence of operations.

The data structures fall into two categories: either they only assume that the keys can be compared in constant time, i.e. the comparison-based setting, or they assume that the keys are integers in a certain range.

In the latter case we use Random Access Machine (RAM) model, with a word size of w bits. In this model we assume that addition, subtraction, bitwise logical operations, comparison, arbitrary left/right bit shifts and multiplication can be performed in constant time on $O(w)$-bit operands. Furthermore, we assume that the number of words of memory that can be addressed (in constant time) is 2^w. This means that we require w to be at least $\log(m+n)+c$ for some constant $c > 1$, so that the input can fit into the addressable memory while leaving at least linear working space for the algorithms. We assume that the edge costs are integers in the range $0..C$ for some $C \leq 2^w - 1$, i.e., edge costs are single-precision integers.

An important motivation for preferring this model to the (much more elegant) comparison-based one is that many real-life instances of this problem deal with integer or floating-point data. This model is clearly well-suited for handling integer data, but it turns out that all the data structures in this model would work without modification even if the keys were floating-point numbers conforming to the common IEEE 754 standard (Henessy and Patterson (1994)).

For the sake of simplicity we will concentrate on amortized running time as the primary measure of complexity, and whether or not the algorithm is deterministic as the secondary measure. All other factors are suppressed. As some technical conditions are omitted from he statement of the lemmas, in the interests of simplicity, the reader is urged to look up the original source before using one of the lemmas stated here.

The rest of this note is organized as follows. In Section 2 we mention the best-known comparison-based result for this problem. In Section 3 we cover the building blocks used by the implementations of Dijkstra's algorithm for the integer weights case. In Section 4 we state the current best complexity bounds on implementations of Dijkstra's algorithm. In Section 5 we describe some open problems and directions for further research.

2 Comparison-based structures

The first comparison-based F-heap supporting a constant-time decrease-key was the Fibonacci heap whose time bounds are as follows:

Lemma 1 (Fredman and Tarjan (1987)) *There is an F-heap which supports* insert *and* decrease-key *in* $O(1)$ *time each and* delete-min *and* delete *in* $O(\log n)$ *time each.*

This gives an optimal $O(m + n \log n)$ comparison-based implementation of Dijkstra's algorithm.

3 Integer Data

3.1 General Reductions

In this section we cover general reductions from one data-structuring problem to another (or, in one case, to sorting).

An *atomic PQ* with *capacity* M is a PQ which supports all operations in $O(1)$ time, but can hold at most M keys.

Lemma 2 (Fredman and Willard (1994)) *Assuming the existence of an atomic PQ with capacity* $M \geq (\log n)^2$*, there is an F-heap which supports* insert *and* decrease-key *in* $O(1)$ *time each and* delete-min *and* delete *in* $O(1 + \log n / \log M)$ *time each.*

If the the resulting F-heap is used for monotone operations, then the operations performed on the atomic PQs are also monotone.

Lemma 3 (Thorup (1996)) *If there is a non-decreasing function* $T(n)$ *such that for all* n*,* n *keys can be sorted in* $nT(n)$ *time then there is a PQ which supports all operations in* $O(T(n))$ *time each, provided the operation sequence is monotone.*

Remark: Lemma 3 holds in the RAM model as defined above.

Lemma 4 (Cherkassky and Goldberg (1996), Cherkassky et al. (1996)) *If the edge costs are integers in the range* $0..C$*, then for any integers* $1 \leq k \leq \log C$ *and* $t > 0$*, if there is an F-heap which can perform* monotone *operations* insert *and* decrease-key *in* $O(1)$ *time and* delete-min *in* $T(t)$ *time on keys in the range* $0..C$*, then Dijkstra's algorithm can be implemented in* $O(m + n(k + \frac{kC^{1/k}}{t} + T(t)))$ *time.*

3.2 Integer PQs

The following result comes from the well-known van Emde Boas data structure:

Lemma 5 (van Emde Boas et al. (1977)) *There is a PQ which performs all operations in* $O(\log \log C)$ *time if the keys are integers in the range* $0..C$*.*

The following result combines elements of the above result with Lemma 7 below. Note that although the van Emde Boas data structure allows predecessor and successor queries, this data structure does not.

Lemma 6 (Thorup (1996)) *There is a PQ which performs all operations in* $O(\log \log n)$ *time each.*

3.3 Packed Data Structures

These assume that the size of the keys is significantly shorter than w bits (say by a factor f), pack several keys into a single word and use bit-level parallelism to obtain speedup (by a factor of almost f) over conventional data structures.

Lemma 7 (Thorup (1996)) *For any $1 < k \leq w$, there is a PQ which performs all operations in $O(1 + \frac{\log n \log k}{k})$ time, provided all keys are $O(w/k)$ bits long.*

An immediate consequence is that:

Corollary 8 *For any $1 < k \leq w$, there is a PQ which performs all operations in $O(1)$ time, provided all keys are $O(w/k)$ bits long, and the maximum number of keys is $2^{k/\log k}$.*

As Corollary 8 gives an atomic PQ with capacity $2^{k/\log k}$ for $O(w/k)$-bit keys, Lemma 2 implies:

Corollary 9 (Raman (1996)) *For any integer $1 < k \leq w$, there is an F-heap which performs* insert *and* decrease-key *in $O(1)$ time and* delete-min *and* delete *in $O(1 + \frac{\log n \log k}{k})$ time provided all keys are $O(w/k)$ bits long.*

3.4 Multi-level Buckets

The multi-level bucket data structure was first introduced by Denardo and Fox (1979) and further developed and analyzed by Ahuja et al. (1990).

The data structure consists of a two-dimensional $k \times C^{1/k}$ array of "buckets", each of which contains a pointer to a list of keys. k is an integer parameter, $1 \leq k \leq \log C$, which is chosen later so as to maximize efficiency. Each key x in the data structure is associated with an *index*, which is simply the array index of the bucket in which it is stored.

The most important operation in the data structure is that of keeping track of the non-empty bucket with (lexicographically) smallest array index, while keys are moved around among the buckets. One way to achieve this is to maintain the multiset $\{index(x)\}$ where x ranges over the keys in the data structure. It turns out that a run of Dijkstra's algorithm results in n inserts and delete-mins and at most $(m + kn)$ decrease-keys on this multiset (this operation sequence is not monotone). The bookkeeping cost for the rest of the data structure is negligible:

Lemma 10 (Ahuja et. al. (1990)) *Let k be as above. A monotone sequence of n* insert *and* delete-min *operations and m* decrease-key *operations can be processed using the multi-level bucket data structure in $O(m + kn + \tau)$ time, where τ is the time to perform a (non-monotone) sequence of n* insert*s, n* delete-min*s and at most $(m + kn)$* decrease-key*s on the multiset of indices.*

3.5 Integer F-heaps

Going back to the previous section, note that an index can be viewed as an integer in the range $1..kC^{1/k}$. This can be exploited in several ways. Firstly, if C is small enough, the number of *distinct* keys in the index multiset would be much smaller than n. Ahuja et al. (1990) exploited this by giving an F-heap which supports insert and decrease-key in $O(1)$

time and delete-min and delete in $O(\log D)$ time, where D is the number of distinct keys in the collection. It can easily be seen that this gives a bound of $O(m + n\sqrt{\log C})$ overall.

Another way to exploit it is by noticing that an index is $\log(kC^{1/k}) = O(\log\log C + \frac{\log C}{k}) = O(w/k)$ bits long (provided we do not choose k to be greater than $w/\log w$, as will be the case). Now invoking Corollary 9 gives that $\tau = O(m + kn + n\frac{\log n \log k}{k})$ yielding an overall running time of:

$$O(m + kn + n\tfrac{\log n \log k}{k}) = O(m + n\sqrt{\log n \log\log n}),$$

by choosing $k = \sqrt{\log n \log\log n}$ (note that $w \geq \log n$ so k is not too large.) Hence we get:

Lemma 11 (Raman (1996)) *There is an F-heap which performs* insert *and* decrease-key *in $O(1)$ time and* delete-min *in $O(\sqrt{\log n \log\log n})$ time, provided the operation sequence is monotone.*

Another integer F-heap is obtained by combining Lemma 3 with fast sorting algorithms:

Lemma 12 (Andersson et al. (1995)) *If $w \geq (\log n)^{2+\epsilon}$ for some fixed $\epsilon > 0$, n keys can be sorted in linear expected time.*

Rewriting the condition on w in the above lemma as a condition on n and plugging this into Lemma 3 we get:

Corollary 13 *There is a PQ which supports all operations in $O(1)$ expected time per operation, provided the number of keys is bounded by $2^{w^{1/2-\epsilon}}$ for some fixed $\epsilon > 0$, and provided the operation sequence is monotone.*

This gives an atomic PQ for monotone applications with capacity $2^{w^{1/2-\epsilon}}$. Plugging Corollary 13 into Lemma 2 we get:

Corollary 14 *There is an F-heap which performs* insert *and* decrease-key *in $O(1)$ expected time each and* delete-min *and* delete *in $O(1+\frac{\log n}{w^{1/2-\epsilon}})$ expected time each, for any fixed $\epsilon > 0$, provided the operation sequence is monotone.*

4 Current Results

All the results mentioned in this section are for integer data.

4.1 Randomized Results

From Corollary 14 we get:

Theorem 15 *Dijkstra's algorithm can be implemented in $O(m + n\frac{\log n}{w^{1/2-\epsilon}})$ expected time, for any fixed $\epsilon > 0$.*

Remark: Among the interesting consequences is that for $w \geq (\log n)^{2+\epsilon}$ the expected running time of Dijkstra's algorithm is $O(m+n)$. Also, the assumption $w \geq \log n$ implies that in all cases, the expected running time is bounded by $O(m + n(\log n)^{1/2+\epsilon})$, for any fixed $\epsilon > 0$, and only. Thorup (1996) notes both the above consequences but does not explicitly state Corollary 15.

Now, going to Lemma 4 and choosing $k = (\log C)^{1/4}$, $t = kC^{1/k}$, we obtain that Dijkstra's algorithm can be implemented in $O(m + n((\log C)^{1/4} + T(t)))$ time. Invoking Corollary 14 gives us:

$$T(t) = \frac{\log t}{w^{1/2-\epsilon}} \leq \frac{\log\log C + (\log C)^{3/4}}{w^{1/2-\epsilon}} = O((\log C)^{1/4+\epsilon})$$

for any fixed $\epsilon > 0$, since $w \geq \log C$. We conclude:

Theorem 16 *Dijkstra's algorithm can be implemented in $O(m + n(\log C)^{1/4+\epsilon}) = O(m + nw^{1/4+\epsilon})$ expected time, for any fixed $\epsilon > 0$.*

Note that for $w \leq (\log n)^{4/3}$, Theorem 16 gives a bound of $O(m + n(\log n)^{1/3+\epsilon})$ expected time, for any fixed $\epsilon > 0$. On the other hand, for $w > (\log n)^{4/3}$. Theorem 15 gives a bound of $O(m + n(\log n)^{1/3+\epsilon})$ expected time, for any fixed $\epsilon > 0$. Hence:

Corollary 17 *Dijkstra's algorithm can be implemented in $O(m + n(\log n)^{1/3+\epsilon})$ expected time, for any fixed $\epsilon > 0$.*

Remark: Cherkassky and Goldberg (1996) used the weaker form of Thorup's result to get a bound of $O(m+n(\log C)^{1/3+\epsilon})$; this implies a expected running time of $O(m+n(\log n)^{2/5+\epsilon})$ for Dijkstra's algorithm.

4.2 Deterministic Results

If we restrict attention to deterministic algorithms we have the following results as a direct consequence of Lemmas 5 and 6, which are the fastest for very sparse graphs:

Corollary 18 *Dijkstra's algorithm can be implemented in $O(m \log\log n)$ time.*

Corollary 19 *Dijkstra's algorithm can be implemented in $O(m \log\log C)$ time.*

As a consequence of Lemma 11 we get:

Corollary 20 *Dijkstra's algorithm can be implemented in $O(m + n\sqrt{\log n \log\log n})$ time.*

Now, going to Lemma 4 and choosing $k = (\log C \log\log C)^{1/3}$, $t = kC^{1/k}$, we obtain that Dijkstra's algorithm can be implemented in $O(m+n((\log C \log\log C)^{1/3}+T(t)))$ time. Invoking Corollary 14 gives us:

$$T(t) = \sqrt{\log t \log\log t} = O\left(\sqrt{\frac{\log C \log\log C}{k}}\right) = O((\log C \log\log C)^{1/3}),$$

(noting that $k \leq C^{1/k}$ and hence $\log t = O(\frac{\log C}{k})$). We conclude:

Theorem 21 *Dijkstra's algorithm can be implemented in $O(m + n(\log C \log\log C)^{1/3}) = O(m + n(w \log w)^{1/3})$ time.*

5 Open Problems

Here is a list of possible lines of research.

- Experimental evidence shows that Fibonacci heaps are slow in practice, but they do not make use of the monotonicity property. Can monontonicity be exploited to give a practical variant of this data structure? The same remark applies to the data structure of Lemma 2.
- Note that Lemma 6 is asymptotically as good as Lemma 5 unless C is quite small compared to n. In particular an edge weight has to be $(\log n)^{o(1)}$ bits long for this to happen. However, w is always greater than $\log n$. Can any improvements be made for this case? (A bit unlikely.)
- Can the $\log k$ factor be removed from Lemma 7? This would give direct improvements to Corollary 20 and Theorem 21.
- The fact that the operations on indices in Lemma 10 are not monotone means that we cannot apply Lemma 10 recursively. Any way around?
- Can Corollary 14 be improved by using multi-level buckets?
- A linear-time sorting algorithm for any $w = \omega(\log n) \cap (\log n)^{2+o(1)}$ would result in immediate consequences for shortest-paths problems. For instance, a randomized linear-time algorithm for all $w \geq (\log n)^{4/3}$ would not only give a linear-time algorithm for shortest paths for this range of w but would also improve Corollary 17 to $O(m + n(\log n)^{1/7})$. Deterministic linear-time sorting algorithms would also have many consequences.
- The role of multi-level buckets needs to be better understood (perhaps viewing them as tries, as suggested by Kurt Mehlhorn, would help). These are used explicitly in Lemma 3, Lemma 4 and Lemma 11. Hence, each of the applications of Lemma 4 in Section 4 has two distinct instances of multi-level bucket data structures interacting in a way that is not fully understood. Can this be simplified?

References

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.

A. Andersson, T. Hagerup, S. Nilsson and R. Raman. Sorting in linear time? In *Proc. 27th ACM STOC*, pp. 427–436, 1995.

B. V. Cherkassky and A. V. Goldberg. Heap-on-top priority queues. TR 96-042, NEC Research Institute, Princeton, NJ, 1996.

B. V. Cherkassky, A. V. Goldberg and C. Silverstein. Buckets, heaps, lists and monotone priority queues. TR 96-070, NEC Research Institute, Princeton, NJ, 1996.

E. V. Denardo and B. L. Fox. Shortest-route methods: 1. Reaching, pruning and buckets. *Oper. Res.* **27** (1979), pp. 161–186.

P. van Emde Boas, R. Kaas and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Sys. Theory*, **10** (1977), pp. 99–127.

M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization problems. *J. ACM*, **34** (1987), pp. 596–615.

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.

R. Raman. Priority queues: small, monotone and trans-dichotomous. *Proc. 4th Annual ESA*, LNCS 1136, pp. 121–137, 1996.

M. Thorup. On RAM Priority Queues. In *Proc. 7th ACM-SIAM SODA*, pp. 59–67, 1995.