A survey of the connection between SSSP, sorting, and priority queues on the word RAM

Abstract

We survey and discuss the connection between three of the most fundamental problems in theoretical computer science: sorting, priority queues, and shortest paths. We will begin by illustrating the developments of priority queue-based shortest path algorithms from Dijkstra's original formulation to Thorup's results [8]. We will then move on to analyzing the connection between the aforementioned class of methods and the problem of sorting; in order to do so, we will present Thorup's results [10; 12] on the equivalence of sorting and priority queues. Finally, we will tie the discussion back to single-source shortest paths by showing the results obtained by Thorup [9], who studied methods that bypass the sorting bottleneck by relaxing the order in which the graph nodes are visited.

Keywords: SSSP, Sorting, Priority Queues, Thorup's algorithm

1. Introduction

The Single-Source Shortest Paths (SSSP) problem is one of the most fundamental issues studied in theoretical computer science. The problem is to compute, given a graph and a single source node within the graph, the path of least distance from that source to every other node. In this paper, we mostly examine the SSSP problem on graphs with non-negative integer weights, which has connections to both integer priority queues and integer sorting algorithms.

This problem has great practical importance, and there are several well-known algorithms that have been developed to approach it. Dijkstra's algorithm, which is probably the most widely used algorithm, uses a greedy optimization strategy that is best implemented using a priority queue. Priority queues are data structures which at minimum allow for insertions and deletions of elements as well as tracking of the minimum stored element. While they are important on their own, their connection to the SSSP problem through Dijkstra's algorithm has motivated the discovery of fast integer priority queues discussed in Section 3.

Priority queues are also deeply linked to sorting: given a priority queue, clearly it is possible to sort a set of elements by inserting them all and then repeatedly removing the priority queue's minimum. Interestingly, this connection also goes in the opposite direction as well, which was shown by Thorup in 2007 [12]; this is discussed in Section 4. As a result, one can use developments in integer sorting to create faster implementations of Dijkstra's algorithm. However, we see that the link between sorting and priority queues introduces "bottleneck" for solving the SSSP problem using Dijkstra's algorithm, since in order to solve integer SSSP using Dijkstra's we would also have to solve integer sorting.

For this reason, there have also been attempts to solve the integer SSSP problem without using Dijkstra's in order to avoid this "bottleneck." In particular, we will discuss in Section 5 a linear-time algorithm presented by Thorup [9] which solves SSSP on undirected graphs, demonstrating that it is possible to solve SSSP quickly without using Dijkstra's algorithm and priority queues.

2. Integer SSSP and the RAM Model

In this paper, we discuss the specific type of the SSSP problem where all edge weights are non-negative integers. Explicitly, given a graph $G = (V, E)$ where each edge has an associated non-negative integer weight and a single source node $s \in V$, we compute the shortest path from s to all other vertices in V. Throughout, we will use $n = |V|$ and $m = |E|$. Since the edge weights are all non-negative, we focus on *monotonic* priority queues, which are priority queues where the minimum value in the priority queue never decreases. Additionally, since we are considering the integer version of SSSP, we focus mainly on integer priority queues and integer sorting algorithms.

Since we are working with integers, it is useful for us to define the RAM model as an alternative to the so-called comparison model. In the comparison model, determining the order of elements can only be done via binary comparisons, and it is well known that under this model, sorting any set of n comparable elements can be done in no better than $O(n \log n)$ operations. In the RAM model, elements, in this case integers, are stored in binary in machine words of length w, and we are able to add, subtract, and shift[1] machine words in constant time. However, a machine word could hold multiple integers, allowing a single operation to encode a larger number of comparisons, leading to sorting algorithms running in $o(n \log n)$.

The advantage of the comparison model is that it allows sorting on a broad class of elements, while the RAM model's advantages apply only to non-negative integers. However, Thorup argues in his paper that non-negative integers may be used to represent a variety of other domains while still preserving intact ordering. An example of this is negative integers. Suppose that we are storing integers with the highest degree bit meaning a negative number. Then, it is sufficient for us to flip this signed bit and consider the result as an unsigned integer, and the resulting ordering of numbers will be the same as the ordering of the (possibly negative) integers before the bit switch.

On the other hand, the flexibility given by integer operations is truly impactful when it comes to the performance of the algorithms of interest for this paper. As previously mentioned, the key insight is that a single operation may help manage what otherwise would necessitate a number of comparisons. For example, Albers and Hagerup [1] introduce the concept of "packed sorting" by demonstrating how two machine words, each containing k elements in sorted order, can be merged into two sorted machine words in $O(\log k)$ time, compared to the $O(k)$ that the comparison model would yield. Even for larger keys (that still fit in a machine word), Hagerup [1] gives a method that can sort in $O(n \log \log n)$ time. Many of the results discussed leverage the power of the RAM operations, so for the interested reader, Hagerup [5] further defines the model and discusses it in additional depth.

1. In particular, multiplication is not a constant-time operation (with respect to the word length) in general. However, it is often possible to avoid the use of multiplication with little to no loss in performance.

3. Developments in Monotonic Priority Queues

3.1 From Priority Queues to SSSP

As mentioned previously, Dijkstra's algorithm is the most well-known method for computing single-source shortest paths. This algorithm maintains two sets of graph nodes: a set V of visited nodes and a set $V \setminus S$ of unvisited nodes. $V \setminus S$ is maintained as a priority queue, where each node v is stored with a key that represents the currently known distance $D(v)$ from the source to it. $D(v)$ is defined as the distance from s to v using only nodes in S, as opposed to $d(v)$, the actual shortest distance from s to v. Initially, $D(v)$ is set to zero for the source node and ∞ for every other node. Because the weights are non-negative, for the closest unvisited node, $D(v) = d(v)$, so the algorithm computes all $d(v)$ by progressively visiting the closest unvisited node by taking the minimum element of the priority queue containing $V \setminus S$, adding it to S, and updating each distance $D(v)$ from the source for all vertices adjacent to this minimum.

The typical implementation of this algorithm has a runtime of $O(m \cdot T(\texttt{decrease-key}) + n \cdot T(\texttt{extract-min}))$, assuming the initial construction of the priority queue is dominated by the cost of these operations. So the main approach to designing priority queues has been to make `decrease-key` constant-time and then to minimize the runtime of `extract-min`. For sparse graphs, however, it is reasonable to simply try to make both operations as fast as possible. In his 1996 paper [8], Thorup presents a basic monotonic priority queue which gives a $O(m \log \log m)$ solution to the SSSP problem which we will discuss here.

3.2 Thorup's $O(\log \log k)$ Priority Queue

Thorup defines a "basic" priority queue as a priority queue supporting three operations: `find-min`, `insert`, and `delete-min`. For Dijkstra's algorithm, instead of performing a `decrease-key` operation, we can simply insert a new node with the updated key without deleting the old node from the queue. Although the number of keys in the queue is still $O(m)$, the number of `extract-min` operations is now $\Omega(m)$. We also view `extract-min` as `find-min` followed by `delete-min`. Thorup constructs a basic priority queue that provides `find-min` in constant time, and `insert` and `delete-min` in $O(\log \log k)$ time where k is the number of keys in the priority queue, which gives both `decrease-key` and `extract-min` in $O(\log \log m)$ time, yielding an overall runtime of $O(m \log \log m)$ for SSSP.

To create this basic priority queue, Thorup leverages two previous results: the ability to quickly merge two RAM words containing sorted keys proved by Albers and Hagerup [5], and the existence of a basic priority queue that supports a small number of keys with small values where all operations take constant time shown by Raman [7]. Specifically, Albers and Hagerup state that two RAM words each containing k sorted keys can be merged into two words in $O(\log k)$ time, and Raman provides a constant-time basic priority queue for k keys of length at most $O(\frac{w}{k})$ where w is the word-length.

Thorup expands on these results to build another constant-time basic priority queue, this time supporting k keys with size at most $O(\frac{w}{\log k \log \log k})$. Thorup then uses this priority queue along with the recursive range-reduction technique used by van Emde Boas priority queues to obtain a basic priority queue for general integers with constant-time `find-min` and $O(\log \log k)$ time `insert` and `delete-min`.

3.2.1 Building a Constant-Time Integer Priority Queue

To give intuition for the constant-time basic priority queue, Thorup first constructs a basic priority queue offering amortized constant-time operations. In order to hold k keys each of length at most $O(\frac{w}{\log k \log\log k})$, he uses one of Raman's constant-time basic priority queues which can hold $O(\log k \log\log k)$ keys of length $O(\frac{w}{\log k \log\log k})$ (call this data structure R) and a series of "buffers" $B_0, B_1, B_2, \ldots B_{\log k}$ where each buffer B_i contains keys sorted and packed into at most 2^i words. Additionally, there is another one of Raman's constant-time basic priority queues M that holds the minimum key of R and the minimum key of each B_i as well.

We see that `find-min` is the same as calling `find-min` on M, which takes constant time. Additionally, `delete-min` is the same as calling `delete-min` on M, which takes constant time, and then either deleting from R or some B_i, which can be done in constant time. Thus, the only operation left to consider is `insert`.

When we call `insert`, we always try to insert into R. If we insert while R is already full, we extract all the keys in R in order, and pack them into one sorted word to be placed into B_0, which may itself overflow. In general, we place a sorted list of (at most) 2^i words into B_i by merging these with the current contents of B_i, which can be done quickly (in $O(2^i \log(\log k \log\log k)))$ as shown by Albers and Hagerup [6]. If the result from the merge is less than 2^i words, then we keep it in B_i, and otherwise, we place it in B_{i+1}.

We see that extracting all the keys from R and packing them into a word to be placed in B_0 takes constant time per key. Additionally, for an insert resulting in merges up to buffer B_i, we see that we must have at least 2^{i-1} words of keys, and thus, for each merge, we see that it takes $O(\frac{\log(\log k \log\log k)}{\log k \log\log k}) = O(\frac{1}{\log k})$ time per key. Since there are at most $\log k$ buffers, it follows that it takes constant time per key to perform all the merges, and thus `insert` takes amortized constant time.

3.2.2 Building a General Integer Priority Queue

Thorup provides modifications to de-amortize this priority queue, which are interesting but will not be discussed in this paper. He then uses the resulting constant-time basic priority queue on k keys of size at most $O(\frac{w}{\log k \log\log k})$ and recursively builds a general basic priority queue with $O(\log\log k)$ `insert` and `delete-min`. Essentially, for a priority queue with b-bit keys, we split each key into the $b/2$ most significant bits, which we denote $high(x)$. and the $b/2$ least significant bits, which we denote $low(x)$. We create a priority queue H of $b/2$-bit keys keyed by $high(x)$ and then create a priority queue $L[h]$ keyed by $low(x)$ where $h = high(x)$. Similar to van Emde Boas priority queues, we store the minimum value of each priority queue non-recursively, so each `insert` or `delete-min` requires only 1 non-constant-time recursive call to one of the $b/2$-bit priority queues.

This allows us to form the recurrence $T(b) = O(1) + T(b/2)$ where $T(b)$ is the time for `insert` or `delete-min` for a priority queue with b-bit keys. Since Thorup constructed a $\frac{w}{\log k \log\log k}$-bit priority queue that can hold k keys with constant-time operations, it follows that from this recurrence that `insert` and `delete-min` take $O(\log\log k)$ time, assuming each key must be able to be contained in one word.

One of the drawbacks to this priority queue is its space complexity. Since we recursively define priority queues, this deterministic algorithm takes $O(k2^{\epsilon w})$ space. However, Thorup also provides a randomized modification that takes $O(k)$ space while still giving expected $O(\log \log k)$ time per `insert` and `delete-min`. Later, in 2003 Thorup modified this priority queue to also support constant-time `decrease-key`, using exponential trees and Fibonacci heaps, giving an $O(m + n \log \log n)$ solution to the SSSP problem.

4. From Sorting to Priority Queues

As mentioned earlier, given a priority queue that can perform operations in $O(S(n))$ time, we can sort in $O(nS(n))$ time by inserting the items into the priority queue and then repeatedly extracting the minimum item. Interestingly, Thorup also showed the reverse: given the ability to sort n keys in $O(nS(n))$ time, we can construct a priority queue supporting `insert` and `delete` in $O(S(n))$ time, and `find-min` in constant time, assuming our integers can fit within a single machine word in the RAM model [12].

There are several key insights involved in the construction. First, there is laziness—because in a priority queue we only ever query the minimum element, we don't have to keep the whole thing sorted. Instead, we store the keys in buckets (*base sets*) that are divided into progressively smaller chunks (by *level splitters*). The buckets are sorted relative to each other, but not internally. Of course, we have to somehow take advantage of the (assumed) ability to sort quickly. A second level of laziness is that instead of moving keys around one at a time, store them in buffers until there are enough for merging them into the buckets to be efficient. This is where the (assumed) ability to sort quickly is used: before merging the contents of a buffer into the buckets, sort the buffer. Finally, similar to how the priority queues from the previous sections use small, fast priority queues to handle the most important parts of their structure, we use two *atomic heaps*, one to store the bucket containing the smallest keys, and one to store the splitters.

4.1 Atomic Heaps

Although atomic heaps, due to Fredman and Willard [2], illustrate many of the techniques relied on throughout, we will stay at the same level of detail and not describe their construction. Instead, we will simply use the result that given m preprocessing time and space, we can construct a heap with size $O(\log^2 m)$ that allows for constant-time `find-min`, `insert`, and `delete`.

There are a couple of notes that should be made about atomic heaps. First, they are definitely *not* practical: aside from their complex implementation and large constant factors hidden in their asymptotic runtime, they are only defined for $n > 2^{12^{20}}$. Secondly, their original implementation uses multiplication, which is not an AC^0 operation—that is, multiplication cannot be implemented by a circuit of constant depth. However, it was later shown by Thorup that atomic heaps can be implemented using only AC^0 operations on (at the time) new processors, so this issue falls back under the issue complex implementation. Finally, avoiding using atomic heaps is sometimes possible with no loss in performance and often possible with only a small loss in performance. It seems that their convenience—in situations where a few small, fast priority queues would be helpful—causes them to be invoked often, before being replaced later.[11]

4.2 Construction of the Priority Queue

Returning now to the problem of using sorting to construct a priority queue, let Φ be the first multiple of 12 greater than $\log n$. The queue will have disjoint *base sets* $A_0, A_1, \ldots, A_k$, each containing between $\Phi/4$ and Φ keys (so that there are $\Theta(n/\log n)$ base sets in total). The base sets are ordered relative to each other, but are not required to be internally sorted. Consecutive A_i, A_{i+1} are separated by s_{i+1}, a *base splitter* (which is not a key), so that $\max A_{i-1} < s_i \leq \min A_i$.

For each base set, we have a doubly linked list containing the keys of the set. In each node of the list, a pointer to the base set itself will also be stored. Also, we store a counter $m_i = |A_i|$.

The list of base sets will be stored (in sorted order) as a doubly linked list, along with the associated splitters.

The first base set A_0 (the "head") will also be stored in an *atomic heap.* Because $|A_0| = O(\log n)$, these keys will fit in an atomic heap that can be constructed in $O(n)$ time/space.

Similar to in a skiplist, $l = \log_4(n/\Phi) = \theta(\log n)$ base splitters are promoted to become *level splitters* $t_0, t_1, \ldots, t_{l+1}$. It will be enforced that for each j, there are more than $\frac{1}{2}4^j\Phi$ base keys below t_j, and less than $\frac{7}{4}4^j\Phi$ keys below t_j.

The level splitters are also stored in an atomic heap, which will be called the *pre-searcher.*

Finally, for each $j = 1, \ldots, l$, we associate with the splitter t_j a *buffer* B_j containing at most 4^j keys. The buffers may overlap: we require only that the keys in B_j are in $[t_j, t_{j+2})$. Each buffer B_j is stored as a doubly linked list, and is associated with a counter q_j for the number of elements.

Initially, the buffers will be empty. This means that rebuilding the structure from a sorted list can be done in linear time, which implies that the queue could be easily modified to be of variable size (amortizing by table doubling).

4.3 Operations

`find-min` is immediately taken care of—we can simply query the atomic heap containing A_0 in constant time.

For `insert` and `delete`, let's ignore for now the issue of maintaining the invariants outlined above. That means that for `delete`, where we assume we are given a pointer to the node to delete, we can simply remove it from the doubly linked list it is contained in (base set or buffer), and from the atomic heap corresponding to A_0, if applicable. We also decrement any associated counters—all constant time operations.

For insert(x), we first check if x is smaller than the first level splitter, t_1. If it is, then in constant time we compare against the base splitters smaller than t_1 and add x to the proper base set. Otherwise, we use the pre-searcher to find the appropriate level splitter, add x to the corresponding buffer, and increment the counter. Again, this all is constant-time.

4.4 Maintaining Size Invariants

Now, we will discuss how to move keys around to maintain the size invariants on buffers and base sets.

When a buffer B_j reaches 4^j keys (we check this using the associated counter), we *flush* by first sorting B_j and then moving the keys from the buffer into the base sets. It's worth emphasizing that this is the only place we use the assumed sorting algorithm—everything else can be handled using fast known data structures. Once the buffer is sorted, merging it with the sorted list of base splitters can be done efficiently in $O(4^j)$ time: the largest key in B_j is less than t_{j+2}, which has at most $\frac{7}{4}4^{j+2}\Phi$ base keys below it, which implies at most $\frac{7}{4}4^{j+2}\Phi/(\Phi/4)$ base splitters below it. The runtime of this is dominated by the sorting, which is $O(S(n))$ amortized (per key) by assumption.

Now, the above merging of the buffer may cause base sets to overflow. Also, deleting keys may cause base sets to underflow. We can resolve both of these issues in a similar fashion to B-trees: split the base set or merge it with one its neighbors. That is, if a base set reaches the upper bound of Φ keys, we split it around its median, which becomes a new base splitter. If a base set is depleted to its lower bound, $\Phi/4$ keys, we merge it with one of its neighbors, and then split if necessary. Joins and splits each run in $O(\Phi)$ time (including any necessary updates to the atomic heap on A_0), which can be amortized over the $\Omega(\Phi)$ updates which must happen before more than a constant number of joins or splits can occur (the details here are nothing new and so are omitted).

There are couple of details here:

- First, a minor point—in addition to single insert or deletes, flushing a buffer may induce splits or joins, and so the cost of these operations should be amortized in the same way as the cost of the merge.
- Second, and more importantly, we do not want to destroy base splitters that have been promoted to be level splitters. However, each base set has at least one neighbor that it can be safely joined with, except for the very last base set, which we will simply allow to become arbitrarily small.

4.5 Maintaining the Level Splitters

The final invariant to maintain is that of the level splitters.

First, for simplicity, we will always flush B_j before modifying t_j, so as to trivially ensure that $\min B_j \geq t_j$. Conversely, it will be convenient to modify t_j when we have flushed B_j. Finding an appropriate base splitter to promote can be done by scanning through the base sets, adding up the counters until reaching the desired level. This takes $O(4^j)$ time, by essentially the same argument for the merging step of flushing.

Now, note that with the chosen constants, we can't wait for a buffer to be flushed before updating the level splitter. Thorup chooses to increment buffer counters more often than items are added to the buffer, in such a way as to ensure that the level splitters are updated often enough, and there are good reasons for doing this. However, one could argue that allowing a larger range for the level splitters is simpler and sufficient for the amortized bounds claimed here.

4.6 Runtimes

From the analysis above, it should be clear that the desired runtimes have been achieved (amortized). When inserting an item, we can charge it for the cost of flushing it will incur, as well as some constant cost for merging, splitting, and eventual deletion, giving $O(S(n) + O(1)) = O(S(n))$ amortized cost per update, as desired.

We will not discuss this here, but Thorup proceeds to show how to de-amortize this construction, achieving the same guarantees but for the worst-case operation of the queue. He also manages to avoid using an atomic heap for the pre-searcher without changing the running time, but the atomic heap for the head A_0 seems to be necessary to get the claimed runtimes, so we will not present this either.

As a remark, there are a couple of reasons why the reduction from sorting to priority queues is interesting in a larger sense. Of course, it gives a way to take advantage of progress on sorting to make progress on priority queues. Also, more pessimistically, it may provide a way to lower bound the time required to sort, by finding a lower bound on priority queue operations.

Practically, for algorithms such as Dijkstra's and Prim's, constant-time decrease-key is highly desirable. Here, given $O(S(n))$ insert and delete, we can implement decrease-key in $O(S(n))$. However, it is not known if a general sorting to priority queue reduction can achieve constant-time decrease-key.

Thorup (again) did manage to demonstrate a priority queue achieving constant-time insert and decrease-key and delete in $O(\log \log n)$ time [12], which is good as could be derived from the current best known deterministic sorting algorithm, running in $O(n \log \log n)$ time. It is currently unknown if a priority queue to match the known randomized $O(n\sqrt{\log \log n})$ sorting algorithm can be made.

In any case, we will return now to the other direction, the obvious reduction from priority queues to sorting, which motivates a different approach to SSSP.

5. Avoiding the Sorting Bottleneck for SSSP

In the previous sections of this paper, we discussed the connection between sorting, priority queues and the SSSP problem. This suggests an important question: whether or not it is possible to achieve a faster SSSP runtime by using something other than Dijkstra's algorithm and priority queues. The invariant that makes Dijkstra's algorithm correct is that for the vertex in $V \setminus S$ with minimum key $D(v)$, we are guaranteed that $D(v) = d(v)$ and so visiting v is certainly correct. Thorup, on the other hand, attempts to find weaker conditions that imply $D(v) = d(v)$, which would allow vertices to be visited in some acceptable ordering even without needing to keep track of the minimum distance. This results in a linear-time algorithm that solves the integer SSSP problem on undirected graphs [9]. This section of the paper will describe Thorup's algorithm, sketch a proof of its correctness, and analyze its runtime.

5.1 Relaxing Visiting Rules via Bucketing

As anticipated above, Thorup's algorithm is fundamentally different from Dijkstra's because it does not attempt to establish a strict visiting order for vertices. Rather, just like the priority queue in Section 4, it uses buckets which allows for some freedom in picking the next vertex to visit. This algorithm places all vertices into buckets such that given the appropriate bucket, we can choose to visit any vertex in this bucket. The challenge is, of course, devising a bucketing scheme that allows the above process to be performed correctly and efficiently.

Intuitively, the problem of efficient bucketing can be approached as follows: suppose we have two groups of vertices, which are only connected to each other by edges with very large weights. It follows that if the two groups have a similar distance from the source, the shortest paths to any vertices in these two groups must not use the connecting edges with very large weight, so the shortest path to any vertex in one of the two groups will not go through the other. Thus, we can visit the two groups in either order.

We can formalize this intuition: suppose that the vertex set V is partitioned into subsets V_i such that all edges between vertices in different subsets have weight at least δ. Additionally, suppose we can find some i and some $v \in V_i \setminus S$ where $D(v) = \min(D(V_i \setminus S)) \leq \delta + min(D(V \setminus S))$, then $D(v) = d(v)$.

To prove this statement, consider the shortest path from s to v. Let u be the first vertex in the path not in S. We have that $D(v) \geq d(v) \geq d(u) = D(u)$. If $u \in V_i$, since $D(v) = \min(D(V_i \setminus S))$, we know that $D(u) \geq D(v)$, so $D(v) = d(v)$. Otherwise, any path from u to v must cross at least one edge of length δ, so $D(v) \geq d(v) \geq d(u) + \delta = D(u) + \delta$. We see that

$$D(u) + \delta \geq \min(D(V \setminus S)) + \delta \geq \min(D(V_i \setminus S)) = D(v),$$

so it follows that $d(v) = D(v)$.

The second case of this proof demonstrates how this algorithm can do better than Dijkstra's. Clearly, the path from s to u is shorter than the path from s to v, so Dijkstra's algorithm would necessarily visit u before v. However, $d(v) = D(v)$, so it is also valid to visit v before u. This gives us some intuition towards Thorup's algorithm: in this case, the group of vertices containing u is different from the group of vertices containing v, but the two groups are far enough apart, so we can visit them in arbitrary order.

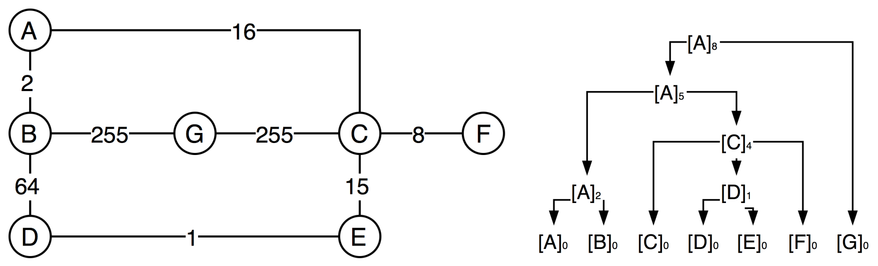
5.2 Efficient Bucketing: The Component Tree

Thorup implements his bucketing structure via a so-called *component tree*. In order to understand the component tree, we must first define a *component hierarchy*, which is a hierarchy of vertex partitions that, like the example above, are separated by edges of at least a certain weight. We define the hierarchy as follows: let G_i be the subgraph of G that contains all vertices in G and all the edges in G of length smaller than 2^i. We find all connected components of G_i and define $[v]_i$ as the connected component of G_i that contains v. We say that $[v]_i$ is the *parent* of $[w]_{i-1}$ if and only if $[w]_i = [v]_i$. We now define the component tree T as follows: each of the nodes of T is a connected component in one of the subgraphs G_i. The root of T is the entire graph G, which we can also express as $[v]_r$ for any $v \in V$, where r is the most significant bit of the longest edge in G. The leaves of T are connected components of G_0, which are simply the vertices of G, and each interior node is connected to its parent and its children as per the definition given above. We additionally disregard any node with only one child, and connect the child to its lowest ancestor with two or more children.

The result is a tree where each node has at least two children, and each node's connected component is a strict subset of its parent. As mentioned above, each leaf is a graph vertex, so there are n leaves in total, and thus the component tree will have at most $2n - 1$ nodes.

This description of the graph is quite useful for Thorup's goal, since it establishes a recursive bucketing structure that follows the intuition from before. Figure 1 gives an example of a graph and its corresponding component tree:

Figure 1: A graph and its component tree



Creating the component tree is easily done by maintaining a union-find data structure and processing the graph's edges in sorted order according to their most significant bits. Whenever we process an edge whose most significant bit is larger than all those seen before, we know that we are moving to a higher level of the hierarchy, and so we can add all the elements in our union-find to the component tree. This construction can be achieved with a running time of $O(m\alpha(m+n))$ thanks to Gabow's data structure [4] and bucket sorting on the edges' most significant bits. However, Thorup explains that it is possible to improve on this bound. When building the component tree, we are only interested in connectivity via short edges, so we can restrict our analysis to the edges in a minimum most significant bit spanning tree of the graph instead of its entirety. Creating such a MST can be achieved using Fredman and Willard's atomic heaps [2] in linear time, and once we have a tree instead of a full graph, we can use Gabow and Tarjan's [4] tabulation-based union-find structure to build the component tree in linear time.

5.3 Visiting the Component Tree Nodes

Now that we have constructed the component tree, we need to establish a visiting scheme that correctly traverses this structure. As we visit a node, we can imagine having access to all its children as elements in an array of buckets. For each node $[v]_i$, we have an array of buckets that its children may be placed in; we place child $[w]_{i-1}$ in the bucket whose index is equal to $\lfloor \frac{\min(D([w]_{i-1} \setminus S)}{2^i} \rfloor$. Notice how this, once again, follows the intuition that we were talking about in Section 5.1: we can pick arbitrarily among two buckets as long as they are at similar distances to the source and are separated by long enough edges. Then, visiting a node in the tree is the same as iterating through its array of buckets and visiting any nodes encountered in any of the buckets. When we find a leaf of the component tree, we finally visit the corresponding vertex, and relax all its adjacent edges, which may require us to update the visiting order of any of the unvisited nodes in the tree.

5.4 A Data Structure for the Unvisited

This highlights the need for another data structure, designed to manage the minimum distances to all the unvisited nodes. Specifically, we need a data structure that, given any root the subtree of an unvisited node $[v]_i$, can update and report $\lfloor\frac{\min(D([v]_i \setminus S)}{2^i}\rfloor$, and upon visiting any unvisited node $[v]_i$, can delete it and compute $\lfloor\frac{\min(D([w]_{i-1} \setminus S)}{2^i}\rfloor$ for any child $[w]_{i-1}$. Luckily, Gabow's split-min data structure [3] was built exactly to fulfill this role, achieving a running time of $O(m\alpha(m+n))$ throughout the execution of the algorithm. Thorup, however, proposes an alternative solution based on atomic heaps, which can achieve a linear running time.

5.5 Linear-time SSSP

Consider now the algorithm as a whole: first, Thorup proposes to build a minimum most significant bit spanning tree for the graph, and then use it to construct a component tree. Using atomic heaps, this takes linear time; otherwise, it requires $O(m\alpha(m+n))$ operations. Thorup then initializes the aforementioned "unvisited data structure" from Section 5.4, and uses the scheme defined above to visit the entire component tree, starting at the root. Thorup proves that the the number of necessary buckets is also $O(m+n)$, showing that SSSP is solvable in linear time in the RAM model. Before arguing about the algorithm's correctness, there is an important point to make about the running time: while atomic heaps offer extremely powerful theoretical results, it is worth noting that they are defined only for $n > 2^{12^{20}}$, which makes them practically useless. On the other hand, the union-find data structure that is proposed as an alternative offers good (even if super-linear) theoretical bounds and better practical performance.

We may now move on to arguing why the algorithm proposed by Thorup correctly computes the shortest path from the source of the graph to all vertices. While the full proof of the algorithm is beyond the scope of this paper, we want to sketch an argument that can prepare the interested reader for following its structure.

First, we consider any parent node $[v]_i$ in the component tree. Given one of its children $[w]_{i-1}$, we say that $[v]_{i-1}$ is a min-child of $[v]_i$ if $\lfloor\frac{\min(D([w]_{i-1} \setminus S)}{2^i}\rfloor = \lfloor\frac{\min(D([v]_i \setminus S)}{2^i}\rfloor$. Then, given any other component tree node $[v]_i$ that contains any unvisited nodes, we say that $[v]_i$ is minimal if it is a min-child of its parent and its parent is also minimal. Thorup shows that given any minimal node $[v]_i$, we have that $\min(D([v]_i \setminus S) = \min(d([v]_i \setminus S)$. Applying this condition to a leaf implies that if only minimal leaves are visited, the algorithm will correctly find a shortest path. Notice the analogy with Dijkstra's algorithm: Dijkstra defines a global minimum, while Thorup constructs this notion of minimality that can obtain the same result.

Going further, Thorup proves that as we visit vertices and relax edges, while we decrease the distance to some of the unvisited nodes, we do not alter the minimality of the nodes currently being processed. This means that all the nodes processed are always minimal, which ensures the algorithm's correctness.

5.6 Other Developments

Besides Thorup's algorithm, there have been other attempts to approach SSSP using an approach that deviates from Dijkstra's algorithm. Hagerup, for example, wrote a paper that attempts to extend Thorup's result to directed graphs. The result obtained, however, does not achieve a linear running time; rather, it runs in $O(n + m \log (w))$ time, which achieves the same $O(m \log \log m)$ runtime as Thorup's priority queue from 1996 for directed integer SSSP, but can be extended to SSSP's close relative APSP (All-Pairs Shortest Paths) to provide a $O(nm + n^2 \log \log (n))$ time solution.

6. Conclusion

We surveyed the work of Mikkel Thorup, who manages to tie together the three fundamental problems that were of our interest: SSSP, sorting, and priority queues. Thorup proves results for the integer SSSP problem by taking advantage of the RAM model, achieving remarkable results in the field. Specifically, we discuss above Thorup's advances in building effecient priority queues, his result that shows the equivalence between sorting in the RAM model and integer SSSP, and finally his approach to SSSP that does not rely on priority queues at all. The connections established between priority queues, sorting, and SSSP suggest some possible results. For instance, a $O(n\sqrt{\log \log n})$ randomized sorting algorithm was given by Han [13], which implies the existence of a priority queue with $O(\sqrt{\log \log n})$ operations, and thus a $O(m\sqrt{\log \log m})$ algorithm for SSSP. However, just as Thorup presented a deterministic priority queue with $O(\log \log n)$ operations, but constant time `decrease-key`, it can be conjectured that there is a randomized $O(\sqrt{\log \log n})$ priority queue with constant-time `decrease-key`, offering a $O(m + n\sqrt{\log \log n})$ solution to SSSP, which remains as an open problem.

Additionally, the connection between sorting, priority queues, and SSSP indicates that finding a fast algorithm for integer sorting would also provide a fast algorithm for SSSP. However, it would be enlightening to see if a connection could be drawn on the last missing side of our triangle: that is, from SSSP to sorting. Results like the linear-time solution to undirected SSSP presented in Section 5 suggest that perhaps SSSP is not equivalent to sorting, yet the question of equivalence between SSSP and sorting is still open. We hope that our survey provides insight on how and why these three problems are related, and how approaching one can help us devise strategies to approach the others.

Acknowledgments

We would like to acknowledge the 6.854 staff for all the helpful lectures and insights from this semester.

References

[1] Albers, S., and Hagerup, T. Improved parallel integer sorting without concurrent writing. *Information and Computation* (1992).

[2] Fredman, M., and Willard, D. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences* (1994).

[3] Gabow, H. A scaling algorithm for weighted matching on general graphs. *26th Annual Symposium on Foundations of Computer Science* (1985).

[4] Gabow, H., and Tarjan, R. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences* (1985).

[5] Hagerup, T. Sorting and searching on the word RAM. *Proceedings of the 15th Symposium on the Theoretical Aspects of Computer Science* (1998).

[6] Hagerup, T. Improved shortest paths on the word RAM. *Lecture notes in computer science* (2000).

[7] Raman, R. Priority queues: small, monotone and trans-dichotomous. *Lecture notes in computer science* (1996).

[8] Thorup, M. On RAM priority queues. *SIAM journal of computing* (1996).

[9] Thorup, M. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM 46-3* (1999).

[10] Thorup, M. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Journal of Computer and System Sciences* (2003).

[11] Thorup, M. On AC^0 implementations of fusion trees and atomic heaps. *Symposium on Discrete Algorithms* (2003).

[12] Thorup, M. Equivalence between priority queues and sorting. *Journal of the ACM 54-6*, 28 (2007).

[13] Thorup, M., and Han, Y. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. *Proceedings of the 43rd Symposium on Foundations of Computer Science* (2002).