Modifying Dijkstra's Algorithm to Solve Many Instances of SSSP in Linear Time

Michael Otte

University of Colorado at Boulder
Dept. Aerospace Engineering Sciences

Technical Report
12 April 2015

Abstract. We show that for graphs with positive edge weights the single-source shortest path planning problem (SSSP) can be solved using a novel partial ordering over nodes, instead of a full ordering, without sacrificing algorithmic correctness. The partial ordering we investigate is defined with respect to carefully chosen (but easy to calculate) "approximate" level-sets of shortest-path length. We present a family of easy-to-implement "approximate" priority heaps, based on an array of linked-lists, that create this partial ordering automatically when used directly by Dijkstra's SSSP algorithm. For graphs $G = (E, V)$ with positive edge lengths, and depending on which version of the heap is used, the resulting Dijkstra variant runs in either time $O(|E| + |V| + K)$ with space $O(|E| + |V| + \frac{\ell_{max}}{\ell_{min}})$ or time $O((|E| + |V|)\log_w(\frac{\ell_{max}}{\ell_{min}} + 1))$ with space $O(|E| + \frac{\ell_{max}}{\ell_{min}} \log_w \lceil \frac{\ell_{max}}{\ell_{min}} \rceil)$, where ℓ_{min} and ℓ_{max} are the minimum (non-zero) and maximum (finite) edge lengths, respectively, and w is the word length of the computer being used (e.g., 32 or 64 in most cases), and K is a function of G such that $K = O(|E|)$ for many common types of graphs (e.g., $K = 1$ for graphs with unit edge lengths). We also describe a linear time pre-/post-processing procedure that extends these results to undirected graphs with non-negative edge weights. Thus, it possible to solve many instances of SSSP in $O(|E| + |V|)$; for these instances our method ties the fastest known runtime for SSSP, while having significantly smaller constant factor overhead than previous methods. This work can be viewed as an extension of Dial's SSSP algorithm that is able to handle floating point edge weights, yields faster runtime, and is based on new theoretical results.

1 Introduction

Finding the shortest path through a graph $G = (E, V)$ of nodes V and edges E is a classic problem. The variation of the problem known as the "single source shortest-path planning problem" (or SSSP) is concerned with finding all of the shortest-paths from a particular node $s \in V$ to all nodes $v \in V$, where each edge $\varepsilon \in E$ is associated with a length $\|\varepsilon\|$. The first algorithm that solves SSSP was presented by Dijkstra in the 1950s using an algorithm that runs in $O(|E| + |V|^2)$ time for the case of non-negative edge lengths [8]. Over the years, more sophisticated priority heap data structures have reduced the runtime to $O(|E| + |V| \log |V|)$ for an algorithm presented by [10], and for which the authors remark is the fastest time bound we can ever hope to achieve for general SSSP with non-negative edge weights ($\|\varepsilon\| \geq 0$). Undaunted, more recent work has yielded algorithms that boast even faster theoretical performance for subsets of SSSP.

[5] uses an approximate heap data structure to achieve runtime $O(|E| + C|V|)$ in space $O(|E| + C|V|)$ for the case of non-zero C-bounded integer edge weights ($0 < \|\varepsilon\| < C < \infty$). [22] presents an extension of [5] that runs in time $O(|V| + |E| + L)$, where L is the length of the longest path. [3] extend [5] to require less space, $O(|E| + |V| + C)$, as well as time $O(|E| + |V|(B + C/B))$ or time $O(|E + |V|(\varDelta + C/\varDelta))$, where $B < C + 1$ and $\varDelta$ are both user defined parameters .

[19] presents a method for C-bounded integer edge weights ($0 \leq \|\varepsilon\| < C < \infty$) that runs in $O(|E| + |V|)$, and then extends this to C-bounded floating point edge weights in [20]. While both of the latter methods represent theoretical milestones, they use Atomic Heaps [11], which has led some to criticize the $O(|E| + |V|)$ versions of [19, 20] as being impractical [2, 6, 15]. Indeed, Atomic heaps were created mainly as a theoretical tool and their presentation in [11] involved a constant factor of $2^{12^{20}}$ for the sake of readability — the authors of [11] state "*An alternative but less readable method circumvents this requirement. However, as already noted we are foregoing any pretense of practicality.*" A second (more practical) variation is also presented in [19] that runs at the slightly increased time of $O(\log(C) + \alpha(|E|, |V|)|E| + |V|)$, where $\alpha(|E|, |V|)$ is the inverse Ackermann function using $|E|$ and $|V|$.

In the current paper we present a new and remarkably simple modification to Dijkstra's algorithm that, for SSSP with positive edge weights ($\|\varepsilon\| > 0$), yields a runtime of $O(|E| + |V| + K)$, where K is a constant that depends on the instance of the problem being solved, and the $|V|$ term is dropped in the case of connected graphs. In general, $K \leq \min\{\frac{d_{max}}{\ell_{min}}, \frac{\ell_{max}}{\ell_{min}}|V|\}$, where d_{max} is the length of the longest *finite* shortest-path in the final solution, ℓ_{min} and ℓ_{max} are the minimum and maximum *finite non-zero* edge lengths in the graph, respectively. Tighter bounds likely exist for particular classes of graphs. **Note that (assuming a finite number of nodes in the graph) *infinite length edges and shortest-paths are allowed*; however they do not directly influence the runtime**. We also present a slightly less-simple version of our heap that runs in time $O((|E| + |V|) \log_w(\frac{\ell_{max}}{\ell_{min}} + 1))$, where w is the word size of the computer being used, e.g., 64. For many classes of graphs and instances of SSSP one or both of these variations runs in linear time, and in many more cases they are faster and/or requires significantly less overhead than other known methods. Finally, we also show how these result can be extended to SSSP with non-negative edge weights ($\|\varepsilon\| \geq 0$) for the special case of undirected graphs.

The contributions of this paper are threefold:

1. **Theoretical**: The presentation/analysis of a new variant of Dijkstra's algorithm that ties the fastest known linear $O(|E|+|V|)$ and $O(|E|)$ time bounds for many instances of SSSP and SSSP over connected graphs, respectively.
2. **Practical**: The description of a relatively simple data structure and corresponding Dijkstra's variant that can be implemented by anybody familiar with arrays and linked-lists.
3. **Conceptual**: The dissemination of a new insight about the SSSP that inspired these heap modifications, and which we hope will enable new and even better algorithms.

The rest of this paper is organized as follows. Section 2 provides an overview of the insight that lead to the new modifications of Dijkstra's algorithm, as well as a high-level overview of the method. In Section 3 we survey related work. In Sections 4 and 5 we define our nomenclature and formally introduce the SSSP problem, respectively. The data structures required for our modifications are presented in Section 6, and the analysis (of completeness, runtime, and runspace) of the resulting variant of Dijkstra's algorithm in Section 7. The extension to undirected graphs with non-negative edge weights appears in Section 8. We conclude with a few remarks in Section 9 and a summary in Section 10. Algorithmic details of a bit-tree required for the "less-simple" heap modification appear in the appendix.

2 Intuition

Dijkstra's algorithm works by incrementally building a "shortest-path-tree" S outward from s, one node at a time (Dijkstra's algorithm appears in Algorithm 1). Each node that is not yet part of the growing S refines a "best-guess" $D(v)$ of its actual shortest-path-length $d(v)$, with the restriction that $d(v) \leq D(v)$. Dijkstra's algorithm guarantees/requires that $d(v) = D(v)$ for the node in $V \setminus S$ with minimum $D(v)$. In modern versions of the algorithm, a min-priority-heap H is used to keep track of $D(v)$ values.

The min-priority-heap H is initialized to empty, best-guesses $D(v)$ are initialized to ∞, parent pointers $p(v)$ with respect to the shortest path tree S are initialized to NULL, and the start node s is given an actual distance of 0 from itself, lines 1-5, respectively.

Each iteration involves "processing" the node $v \in V \setminus S$ with minimum $D(v)$ lines 8-13. Such a node v is extracted from the heap on line 14 (in the first iteration we know to use s, line 6). Next, each neighbor u of v checks if $d(v) + \|(v,u)\| < D(u)$ (i.e., if the distance from s through the shortest-path-tree to v plus the distance from v to u through edge $(v,u) \in E$ is less than u's current best-guess). If so, then u updates its best-guess and parent pointer to reflect the better path via v, lines 10-13. In other words, all neighbors u of v perform the update $D(u) = \min(D(u), d(v) + \|(v,u)\|)$. The heap is adjusted to account for changing $D(u)$ on line 13.

Dijkstra's original algorithm is provably correct (see [8]), based on guarantees that the next node v processed at any step has the following properties:

Algorithm 1: `Dijkstra`(G, s)

```
Input: A graph $G = (E, V)$ of node set $V$ and edge set $E$, and a start node
       $s \in V$.
Output: Shortest path lengths $d(v)$ and parent pointers $p(v)$ with respect to
        the shortest path-tree $S$ for all $v \in V$.
1  $H = \emptyset$ ;
2  for all $v \in V$ do
3      $D(v) = \infty$ ;
4      $p(v) = \text{NULL}$ ;                          /* $S = \emptyset$ */ ;
5  $D(s) = 0$ ;
6  $v = s$ ;
7  while $v \neq \text{NULL}$ do
8      $d(v) = D(v)$ ;                                 /* $S = S \cup \{v\}$ */ ;
9      for all $u$ s.t. $(v, u) \in E$ do
10         if $d(v) + \|(v, u)\| < D(u)$ then
11             $D(u) = d(v) + \|(v, u)\|$ ;
12             $p(u) = v$ ;
13             updateValue$(H, u)$ ;
14     $v =$ extractMin$(H)$ ;
```

1. $v \in V \setminus S$.
2. Either $v = s$ or v has some neighbor u such that $u \in S$.
3. $D(v) \leq D(v')$, for all nodes $v' \in V \setminus S$.

Thus, as has often been remarked, Dijkstra's algorithm works by finding an ordering on $d(v)$ for all $v \in V$. The priority heap data structure enforces (3) and determines this ordering.

The runtime of the heap is a major contributing factor to the overall runtime of Dijkstra's algorithm. Indeed, every reduction in Dijkstra's theoretical runtime bounds has been due to the discovery of better heap implementations or heap implementations that are more amenable to the SSSP. Our paper is no exception to this trend. Before we describe the implementation details of the particular heap we use, we start by sharing the insight into the SSSP that lead us to choose it.

Consider the 4-grids depicted in Figure 1. The start node s is located at the large black node. Nodes are colored in alternating colors based on the level-set of d they belong to. Note that all edges either connect nodes within a particular level-set, or connect the nodes of adjacent level-sets. Recall that Dijkstra allows us to break ties arbitrarily. This means that any node in a particular level set may be processed *before or after* any other node in the same level set without affecting the correctness of the algorithm. The only thing required for correctness is that all nodes in the k-th level-set are processed before any of those in the $(k + 1)$-th. Thus, for this simple case, we do not need to go through the trouble of calculating a full ordering on the nodes of V — it turns out that a partial

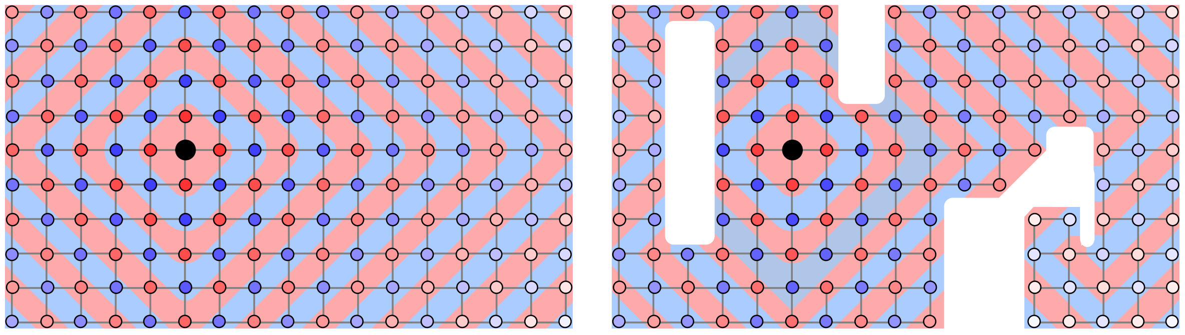

Fig. 1: A 4-connected grid graph (left) and a subgraph created by removing edges (right) are used to show the basic intuition behind our method. The start node s is located at the large black node. Nodes v of the same color *and* lightness are in the same level-set with respect to shortest-path length $d(v)$. The original Dijkstra's algorithm processes all nodes from a particular level-set (removes them from the heap) before any nodes of the next level-set are processed. Also, all nodes from a particular level-set are processed before any nodes of the *same* color but *different* lightness are *added* to the heap. We show how this idea can be generalized such that Dijkstra's algorithm is still complete when nodes are processed according to a partial ordering induced by the carefully chosen "approximate" level-sets (instead of using a full ordering on $d(v)$ as usual).

ordering based on the level-sets of d is sufficient! This is exciting because partial orderings are much faster and easier to calculate then full orderings.

The aforementioned observation was first documented by [5] and used for the case where non-negative edge weights fall into a finite number of level sets that are known *a priori*. However, this idea can be extended to cases where multiple nodes do not naturally fall into the same level-sets (the 4-grid is a special case where nodes fall into level-sets along integers because all edge lengths are 1). In fact, we show that we can get away with grouping nodes into *approximately* the same level-sets, as long as we take a few precautions. In particular, we can group together nodes u and v such that $|d(u) - d(v)| < c_{u,v}$, where $c_{u,v}$ is a constant, as long as $c_{u,v}$ is chosen such that nodes within each group will never be descendants of each other in any valid shortest-path-tree of the particular SSSP instance being solved. Taking this precaution allows us to process nodes in the top-most group in any order. Thus, we are always able to process the first node in the heap — even if it is not the one with the shortest path estimate! (This contrasts with [3] which must scan past each node in the top-most group up to Δ times).

While there are likely countless ways to choose the aforementioned constant $c_{u,v}$ (each representing another Dijkstra variant), we choose to use the length of the shortest edge, $c_{u,v} \equiv \ell_{min} = \min_{(v,u)\in E}(\|(v,u)\|)$. This is a method that is straightforward to implement and analyze, and has fast theoretical runtime on many graphs and many instances of SSSP. The use of ℓ_{min} is motivated by the observation that: while processing v it is *impossible* for any of its neighbors u to experience a best-guess update from $D_1(u)$ to $D_2(u)$ such that $D_2(u) \leq d(v) + \ell_{min}$ (because the edge between u and v is at least as long as ℓ_{min}). Thus, by defining the 0-th, 1-th, 2-th, 3-th, etc. group based on $d(v)$ that fall, respectively, into the ranges $[0, 0]$, $(0, \ell_{min}]$, $(\ell_{min}, 2\ell_{min}]$, $(2\ell_{min}, 3\ell_{min}]$, $\ldots$, we can guarantee that the members of a particular group cannot be descendants of each other in any shortest-path. The members of a particular group may be processed in any order, but all members of group k must be processed before we move onto group $k + 1$. Figure 2 depicts the intuition of this process and a formal proof is presented in Section 7.

Dijkstra's algorithm processes nodes in the order of increasing heap-keys, and so we only need to guarantee that we have placed *all* of the appropriate nodes into the k-th approximate level-set before we start processing *any* nodes from that k-th approximate level-set. This is exactly what the heap we use does. The use of ℓ_{min} is also a convenient because it can be found in a single $O(|E|)$ pass over E.

A related useful insight is that there are no (finite length) edges between nodes that end up in level sets more distant than the maximum (finite) edge length $\ell_{max} = \max_{(v,u) \in E \text{ s.t. } \|(v,u)\| < \infty}(\|(v, u)\|)$. This means that the algorithm will interact with nodes in a contiguous band of only $\frac{\ell_{max}}{\ell_{min}} + 1$ approximate level-sets at any given time (in addition to the level set at ∞), and the $d(v)$-values of the particular band of interest are non-decreasing as the algorithm runs. In Section 7 we show how this can be used to design a memory efficient version of the heap that requires space $O(|V| + \frac{\ell_{max}}{\ell_{min}})$, allowing Dijkstra's algorithm to run in space $O(|E| + |V| + \frac{\ell_{max}}{\ell_{min}})$.

3 Related Work

Dijkstra's algorithm was originally presented in [8] with a runtime of $O(|E| + |V|^2)$. Note that for some *disconnected* graphs it is possible that $|E| < |V|$; therefore, we choose to report runtimes in terms of both $|E|$ and $|V|$, even though terms involving $|V|$ are often omitted in the literature due to the fact that $|V| = O(|E|)$ for *connected* graphs. A heap by [27] yields $O(|E| \log(|V|) + |V|)$, another by [10] gives $O(|E| + |V| \log |V|)$. New heaps by [11] yield $O(|E| + |V| \frac{\log |V|}{\log \log |V|})$ but have the rather large constant factor overhead of $2^{12^{20}}$.

With regard to *expected* running times over randomizations, [12] gives expected time $O(|E| \sqrt{\log |V|} + |V|)$, and heaps by [21] yield the expected times of $O(|E| \log(\log |V|) + |V|)$ and $O(|E| + |V|(\log |V|)^{(1+\epsilon)/2})$. More recent heaps by [16] and [17] give expected times $O(|E| + |V|(\log |V| \log \log |V|)^{1/2})$ and $O(|E| + |V|(\log |V|)^{(1+\epsilon)/3})$, respectively.

The special case of SSSP in which there are z distinct positive edge lengths can be solved in time $O(|E| + |V|)$ if $z|V| \leq 2|E|$ and $O(|E| \log \frac{z|V|}{|E|} + |V|)$, otherwise, using [14]. SSSP on planar graphs can be solved in $O(|V| \sqrt{|V|})$ with a method by [9], note that $|E| = O(|V|)$ for planar graphs.

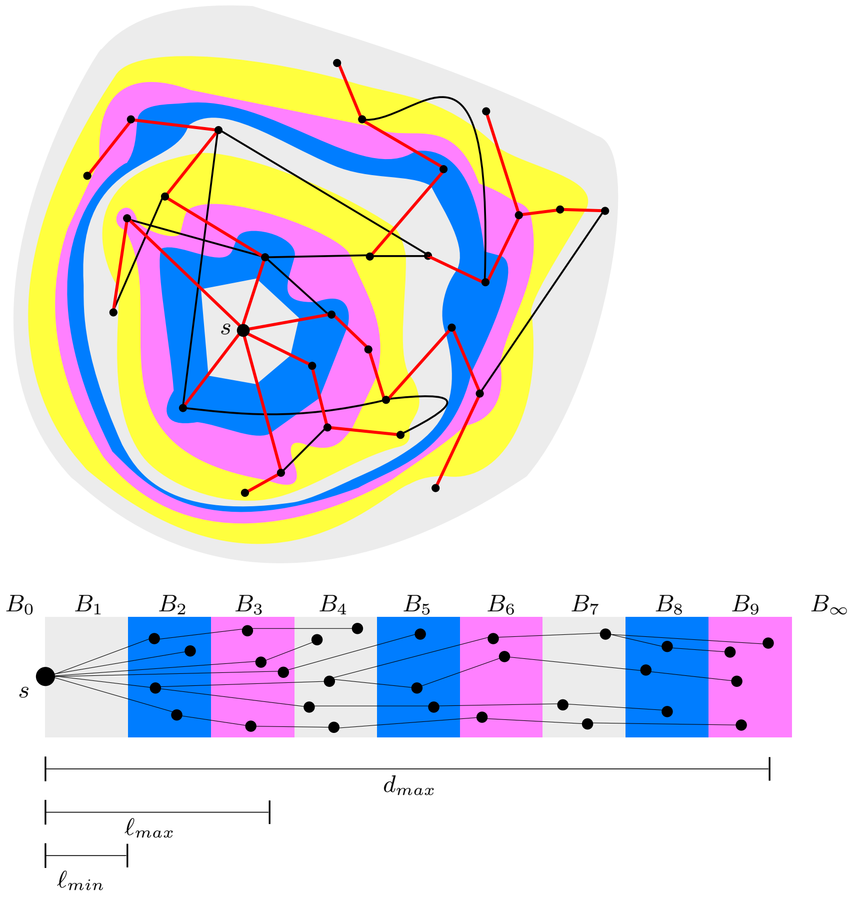

Fig. 2: Color depicts approximate level-sets of d (shortest-path lengths) from all nodes to s. Top: Edges in the shortest path tree are solid, while other edges are dashed. Bottom: A linear depiction of path length level sets, essentially the result of dangling the shortest-path tree by s, and then setting it horizontally across the page (vertical distance in bottom sub-figure is for illustrative purposes only, d is proportional to horizontal distance from s). We divide nodes into approximate level-sets (or buckets) B_k, depicted via different colors/repeats. The length of each bucket is defined by the minimum edge length ℓ_{min} and each run of buckets (i.e., set of non-repeating colors) is no longer than $\ell_{max} + \ell_{min}$, where ℓ_{max} is the maximum edge length. By construction, no edge in the shortest path tree travels between two nodes in the same bucket and no edges travel beyond a single run.

[5] uses an approximate heap data structure to achieve runtime $O(|E| + C|V|)$ in space $O(|E| + C|V|)$ for the subset of SSSP with C-bounded integer edge weights $(0 \leq \|\varepsilon\| < C < \infty)$. This is extended by [22] to floating point edge weights with a runtime of $O(|V| + |E| + L)$, where L is the length of the longest

path. [5] is also extended by [3] to require less space, $O(|E| + |V| + C)$, as well as time $O(|E| + |V|(B + C/B))$ or time $O(|E| + |V|(\Delta + C/\Delta))$, for user defined parameters $B < C + 1$ and Δ. The algorithms of [3,5,22] are arguably the most similar to our own. Indeed, the idea of using an array of linked-lists appears in [5], and [3] suggests using a looping structure to save space. The primary contribution of our work beyond [3,5,22] is a principled way to stratify nodes within the data structures such that: (1) better the time and space bounds are achieved, (2) user parameters are eliminated, (3) both integer and floating point edge weights can be handled, and (4) the first node in the top bucket may always be processed (in contrast to [3] which may scan each node Δ times). We also present an additional alternative data structure that yields better runtime when path-length level sets are sparsely populated.

For the subset of SSSP involving an upper-bound C on non-infinite edge lengths [19] remarks that $O(|E| \log(\log C) + |V|)$ runtime can be achieved by running Dijkstra's algorithm with the priority heaps presented by [13,23,24]. Subsequent heaps by [1], [4], and [17] respectively yield times of $O(|E| + |V|\sqrt{\log C})$ and $|E| + V(\log C \log\log C)^{1/3}$ (expected) and $O(|E| + |V|(\log C)^{1/4+\epsilon})$. Finally, [19] presents a method for the case of C-bounded integer edge weights that runs in $O(|E| + |V|)$, and then extends this to C bounded floating point edge weights in [20]; however, as mentioned earlier, these are mainly of theoretical significance (i.e., instead of practical) due to their use of Atomic Heaps. A variation of this algorithm achieves the slightly worse theoretical runtime bound of $O(\log(C) + \alpha(|E|, |V|)|E| + |V|)$, where $\alpha(|E|, |V|)$ is the inverse Ackermann function using $|E|$ and $|V|$.

[25,26] present a modifications to [20] that achieve better practical performance by removing the necessity of an "unvisited node structure" but do not change theoretical runtime bounds (or the constant factor). [18] perform an empirical evaluation showing that a simple binary heap outperforms state-of-the art implementations on many practical problems, despite having worse runtime bounds.

It worth mentioning that both [19,20] and [25,26] capitalize on the insight that the SSSP can often be solved using a partial orderings over nodes (instead of full orderings) as long as the groups into which nodes are divided are guaranteed not to affect each other during processing. Moreover, they also make the observation that such a valid partial ordering can be created by requiring edges between each subset to have length at least δ. The main conceptual difference between these previous works and our current paper is in the details of the partial orderings that are used. These differences cause both (A) theoretical ramifications regarding the subset of SSSP for which a particular algorithm can achieve linear runtime, and (B) practical differences affecting ease of implementation and performance (including the fact that we do not suffer from the atomic heap overhead of [19,20]). The particular partial ordering used in our current presentation is grounded in the notion of shortest-path-distance level sets, and we believe this makes our method much easier to understand and analyze then the alternative partial orderings used in previous work (e.g., [19,20] builds a dependency-tree of groups using a modified spanning-tree algorithm, and then processes nodes according to the relationships encoded in the dependency-tree).

It is also worth mentioning that our method is only applicable to SSSP with positive edge weights, as well as undirected graphs with non-negative weights if pre-/post-processing is used. Directed edges of length 0 currently break the algorithm, although we are optimistic that our method may eventually be extended to handle them.

4 Nomenclature

A graph G (either directed or underacted) is defined by its edge set E and vertex set V. We assume that both $|E|$ and $|V|$ are finite. Each edge $\varepsilon_{ij} = (v_i, v_j)$ between two vertices v_i and v_j (or from v_i to v_j if the edge is directed) is assumed to have a predefined length (or edge-length or cost) $\|\varepsilon_{ij}\| = \|(v_i, v_j)\|$ such that $0 \leq \|\varepsilon_{ij}\| \leq \infty$ iff $\varepsilon_{ij} \in E$. We follow the standard practice of defining $\|\varepsilon_{ij}\| \equiv \infty$ if $\varepsilon_{ij} \notin E$, but also discuss an alternative in which such edges are assumed not to exist in Section 9.

A path $P(v_i, v_j)$ is an ordered sequence of edges $\varepsilon_1, \ldots, \varepsilon_\ell$ such that $\varepsilon_1 = (v_i, v_1)$ and $\varepsilon_k = (v_{k-1}, v_k)$ for all $k \in \{2, \ldots, \ell - 1\}$ and $\varepsilon_\ell = (v_{\ell-1}, v_j)$ and where $\{\varepsilon_k \in P(v_i, v_j)\} \subset V$. The shortest path $P^*(v_i, v_j)$ is the shortest possible path from v_i to v_j. Formally,

$$P^*(v_i, v_j) \equiv \underset{P(v_i,v_j)}{\arg\min} \sum_{\varepsilon \in P(v_i,v_j)} \|\varepsilon\|$$

We are primarily interested in paths from nodes v to a particular "start-node" s, and define $d(v)$ to be the length of the shortest possible path from v to s.

$$d(v) \equiv \min_{P(v,s)} \sum_{\varepsilon \in P(v,s)} \|\varepsilon\|$$

Each node maintains a non-increasing "best-guess" $D(v)$ of its shortest path length, where $d(v) \leq D(v)$. We define the maximum-minimum finite path length as $d_{max} = \max_{v \in V \setminus \{v' \mid d(v') = \infty\}} d(v')$

Dijkstra's algorithm works by incrementally building a "shortest-path-tree" S outward from s. Our variation of Dijkstra's algorithm relies on knowledge of two quantities that can be obtained using a single $O(|E|)$ pass over E. Namely, ℓ_{max} is the length of longest *non-infinite* edge in E and ℓ_{min} is the length of shortest *non-zero* edge in E.

$$\ell_{max} = \max_{\varepsilon \in E \setminus \{\varepsilon' \mid \|\varepsilon'\| = \infty\}} \|\varepsilon\|$$

$$\ell_{min} = \min_{\varepsilon \in E \setminus \{\varepsilon' \mid \|\varepsilon'\| = 0\}} \|\varepsilon\| \tag{1}$$

The most basic implementations of our method assumes that E contains no zero-length edges, $E \cap \{\varepsilon' \mid \|\varepsilon'\| = 0\} = \emptyset$ and so $\ell_{min} = \min_{\varepsilon \in E} \|\varepsilon\|$. However, in Section 8 we present an extension to non-negative edge lengths for the case of undirected graphs, and for which the more general definition of ℓ_{min} from Equation 1 must be used.

We define $\beta = \lceil \frac{\ell_{max}}{\ell_{min}} \rceil$, and use β to pick the circumference of a circular array that is used in our heap data structures. A particular heap data structure is denoted H.

The heaps we present allow Dijkstra's algorithm to solve SSSP in $O(|E| + |V| + K)$ and $O((|E|+|V|) \log_w(\frac{\ell_{max}}{\ell_{min}}+1))$, respectively. w is the word size of the computer being used (currently 32 or 64 in most computers). K is a constant depending on the problem being solved; in particular, it is the number of *empty* approximate level-sets of d between the 0-th level-set and the approximate level-set containing d_{max}.

We shall often refer to the approximate level-sets, as well as the linked lists that our heap uses to store them, as "buckets" and denote the k-th bucket B_k.

5 Problem

The shortest path planning problem for positive edge weights is defined as follows:

Given $G = (V, E)$ such that $\|\varepsilon\| > 0$ for all $\varepsilon \in E$, and a particular node $s \in V$, then for all $v \in V$, find the shortest path $P^*(s, v)$.

The shortest path planning problem for undirected graphs with non-negative edge weights is defined:

Given $G = (V, E)$ such that $\|\varepsilon\| \geq 0$ for all $\varepsilon \in E$ and for all $(u, v) \in E$ there exists $(v, u) \in E$ such that $\|(u, v)\| = \|(v, u)\|$, and a particular node $s \in V$, then for all $v \in V$, find the shortest path $P^*(s, v)$.

By convention, either problem is considered solved once we have produced a data structure containing both:

1. The shortest-path lengths $d(v)$ for all $v \in V$ from s.
2. The shortest path tree that can be used to extract the shortest path from s to any v (at least for any v such that $d(v) < \infty$).

For example, the latter can be accomplished by storing the parent of each node with respect to S, allowing each shortest path to be extracted by following back pointers in the fashion of gradient descent from v to s and then reversing the result.

The reverse (i.e., sink) search that involves finding all paths to s (instead of from s) can be solved using basically the same algorithm except that the rolls played by in- and out- neighbors are swapped and the extracted path is not reversed.

6 "Approximate-Heap" Data Structure

There are three variants of the heap data structure that we present. The first is mainly used to provide an introduction of concepts and as an analytical tool. The second and third yield the time bounds of $O(|E| + |V| + K)$ and $O((|E| + |V|)\log_w(\frac{\ell_{max}}{\ell_{min}} + 1))$, respectively, and are easy and "less easy" to implement, respectively.

All three heap variations are combinations of two or three simple and widely used data structures. The general idea is to store an array of buckets, where each bucket is implemented as a doubly-linked list. The basic forms of Heap 1 and Heap 2 have previously been described in [5] and [3], respectively; however, the particular stratification that is used is unique to our work and is the source of our method's benefits.

6.1 Heap 1: an introduction and analytical tool

The first and simplest variant heap H is a standard array of L heads of doubly linked lists. We shall refer to the corresponding lists as buckets and denote them B_0 through B_{L-1}. We shall also use the convention that nodes that have never been added to the heap are defined to be an implicit bucket B_∞. Heap 1 is useful as a theoretical tool for analysis, and may have applications outside of SSSP, but is it not quite space efficient enough for direct use with Dijkstra's algorithm (Heap 2, presented in the next section fixes the latter problem).

Let us assume, for the sake of the current discussion, that we are provided with d_{max} *a priori*, where d_{max} is the length of the longest finite shortest-length path (i.e., we can ignore any infinite length paths when calculating d_{max}). The assumption of *a priori* knowledge of d_{max} will be dropped in Heap 2. The array only needs to store $L = \lceil \frac{d_{max}}{\ell_{min}} \rceil + 1$ linked list heads (any node that is ever supposed to be moved into $B_{h \geq L}$ can remain in B_∞, since (by construction of the heap) that node must eventually receive information about a path short enough to cause insertion into one of the buckets B_0 through B_{L-1}.

B_k holds nodes that currently believe they belong to the k-th approximate level set, i.e., $\forall v \in V \setminus S$ such that $(k-1)\ell_{min} < D(v) \leq k\ell_{min}$. This enforces a partial ordering of nodes $v \in V$ based on $D(v)$ with the convenient property that it is impossible for edges ε_{ji} from nodes $v_j \in B_h$ to decrease $D(v_i)$ for nodes $v_i \in B_k$ such that $k < h$ (since doing so would require $\varepsilon_{ji} < \ell_{min}$, which is impossible) — as we prove formally in Section 7.

Dijkstra's original algorithm requires an initial insertion of all nodes into the heap. The use of the implicit B_∞ means that we can ignore this step (this small modification does not affect runtime bounds and has been used by others in the past)[1]. Thus, membership in B_∞ can be determined in $O(1)$ time and does

[1]In practice, this can be achieved either by allowing nodes to store their status internally or by using an additional length $|V|$ array of pointers that are initialized to null, point to the list-node that holds each node that is currently in a linked list, and reset to ∞ when that node is extracted from the queue.

Algorithm 2: `updateValue`(H, v) for basic version (Heap 1)

```
Input: Heap $H$ (of type Heap 1) and a node $v$.
Output: Updates the position of $v$ within $H$ based on $D(v)$, adding $v$ to $H$ if
        necessary.
1 $k =$ currentBucketID$(v)$ ;
2 if $k < \infty$ and $(k-1)\ell_{min} < D(v)$ and $D(v) \leq k\ell_{min}$ then
3     return;
4 if $k < \infty$ then
5     RemoveFromList$(B_k, v)$ ;
6 $k = \lfloor D(v)/\ell_{min} \rfloor$ ;
7 AddToListFront$(B_k, v)$ ;
8 if $k < \hat{k}$ then
9     $\hat{k} = k$ ;
```

not change our results in any fundamental way. We ignore these non-critical complications for now, but discuss an alternative variation in Section 9.

In general, any priority heap must implement the two functions `extractMin`(H) and `updateValue`(H, v). `extractMin`(H) is traditionally responsible for extracting the node with the lowest key value. In our case the key value is $D(v)$ and we design the heap such that it returns node v such that v in the same approximate level-set as the node u with the lowest key-value, formally `extractMin`(H) returns v such that $v, u \in B$ and $u = \arg\min_{v' \in H} D(v')$. `updateValue`$(H, v)$ updates the location of v within H; for example, after $D(v)$ has been modified.

The `updateValue`(H, v) and `extractMin`(H) operations for Heap 1 are described in Algorithms 2 and 3, respectively. We assume that we are provided with a doubly-linked-list data structure that is able to add and remove node v to/from a list B in $O(1)$ time using `AddToListFront`(B, v) and `RemoveFromList`(B, v), and able to remove the front node of a list in time $O(1)$ using `PopList`(B) (which returns null if the list is empty). The subroutine `currentBucketID`(v) returns the index of the bucket that currently contains v and can be implemented using either the table described above or by allowing v to store the information internally; in either case the time of calling it is $O(1)$. The subroutine `IsEmpty`(B) return true if the list is empty and false if it is not; this is clearly $O(1)$. Finally, the heap also maintains the index $\hat{k}$ of the first nonempty bucket, and $\hat{k}$ is initialized to 0.

`updateValue`(H, v) is presented in Algorithm 2. It starts by checking if v is already in a bucket, and then returns if v is already in the correct bucket (lines 1-3). If v is in an incorrect bucket, then it is removed from the incorrect bucket (line 4). Next, the index of the correct bucket for v is calculated, line 6, and v is added to the appropriate linked list (line 7). Finally, we update $\hat{k}$ if v has been placed into an earlier bucket than $B_{\hat{k}}$, lines 8-9. In Section 7 we prove that $\hat{k}$ is non-decreasing, and so if this heap is used with Dijkstra's algorithm

Algorithm 3: `extractMin`(H) for basic version (Heap 1).

```
    Input: Heap $H$ (of type Heap 1).
    Output: Node $v$ such that $D(v) - \min_{u \in H} D(u) < \ell_{min}$.
1   while $\hat{k} < L$ and IsEmpty($B_{\hat{k}}$) do
2       $\hat{k} = \hat{k} + 1$ ;
3   if $\hat{k} < L$ then
4       return NULL ;
5   return PopList($B_{\hat{k}}$);
```

then $k < \hat{k}$ never evaluates to true. The final check is included here only for the purpose of presenting a correct data structure, in general.

`extractMin`(H) is presented in Algorithm 3. It starts by finding the earliest non-empty bucket (lines 1-2). If no data is contained then it returns null, lines 3-4, otherwise it removes the node from the front of the earliest non-empty bucket and then returns that node, line 5.

6.2 Heap 2: a ring array of lists

Let $\beta = \lceil \frac{\ell_{max}}{\ell_{min}} \rceil$. We observe that it is impossible for the processing of $v_i \in B_k$ to cause any neighbor v_j of v_i to be moved to any bucket between array index k and $k + h$, where $0 < h \leq \beta + 1$ (since doing so would require $\|\varepsilon_{ji} > \ell_{max}\|$, which is impossible) — this is formally proved in Section 7.

This means that only $\beta + 1$ contiguous bucket levels are ever needed at any time during the algorithm's execution. Thus, we can modify the data structure to be more space efficient by simply defining bucket $B_k \equiv B_{\mathtt{mod}(k,\beta+1)}$. Another way of thinking about this is that the array is a loop that overlaps itself every $\beta + 1$ positions.

While the runtimes for extract min and update value remain essentially unchanged (now requiring an additional modulo operation) we end up using significantly less space. The fact that we can use $B_k \equiv B_{\mathtt{mod}(k,\beta+1)}$ is also convenient given that we do not usually know d_{max} *a priori*, but $\beta = \lceil \frac{\ell_{max}}{\ell_{min}} \rceil$ can be calculated in time $O(E)$ using a single pass over E.

Readers that dislike assuming modulo and division require time $O(1)$ are referred to Section 9, where we describe how the modulo and division operations can be replaced with bit-shift and multiplication operations (and float to integer conversion in the case that edge lengths are floating point numbers).

Versions of `updateValue`(H, v) and `extractMin`(H) that are modified for use with Heap 2 appear in Algorithms 4 and 5, respectively. The main differences vs. Heap 1 are obviously related to the looping nature of the algorithm, in particular, the calculation of $\hat{k}$ (lines 4.1-4.3), and the check for the next non-empty bucket (line 5.1-5.4).

Heap 2 is complete when properly used in conjunction with Dijkstra's algorithm solving SSSP for positive edge weights (this is proved in Section 7), but may not be in other applications.

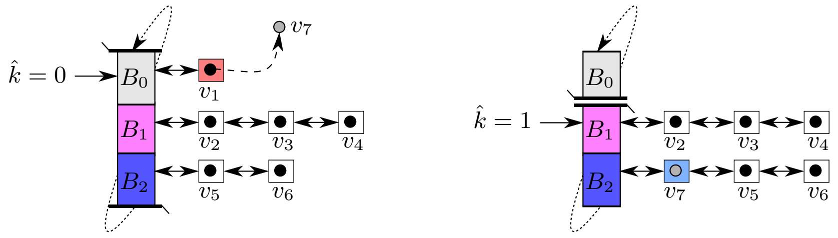

Fig. 3: Node v_1 is removed from the heap H using `extractMin`(H) (left vs. right). During the processing of v_1 its neighbor v_7 is added to the heap using `updateValue`(H, v_7) (right). Both Heap 1 and Heap 2 use an array of doubly-linked-lists, i.e., buckets B. Heap 2 defines $B_{\hat{k}} \equiv B_{\mathtt{mod}(\hat{k},\beta+1)}$ such that buckets at "index" $\hat{k}$ loop back around (loop arrow). The '∽' graphic is used to indicate the current position where this looping occurs. Note that it moves down as $\hat{k}$ decreases.

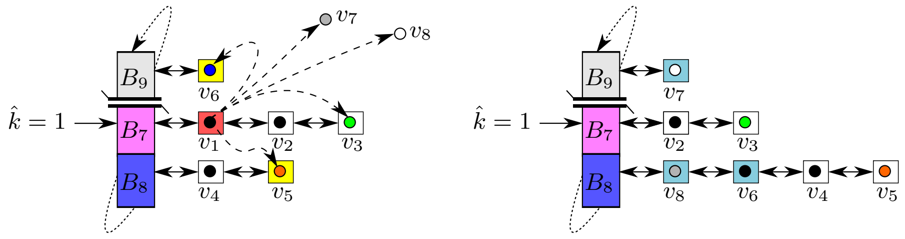

Fig. 4: Node v_1 is removed from the heap H using `extractMin`(H) (left vs. right). yellow represents reduced $D(v)$ for v, while light-blue represents movement into or within the heap. v_1 has neighbors v_3, v_5, v_6, v_7, v_8. Processing v_1 cannot reduce $D(v_3)$ when $v_1, v_3 \in B$ and so v_3 is not moved (this happens because of the particular way we have defined the approximate level-set width of buckets). $D(v_5)$ is reduced, however (as we show in Section 7), this reduction will never be enough to place v_5 into the same bucket as v_1 (thus, v_5 will remain in the next-lowest bucket $B_{8\equiv 2}$). $D(v_6)$ is reduced such that v_6 is moved to $B_{8\equiv 2}$. Node v_7 is added to H at $B_{9\equiv 0}$ and v_8 is added to H at at $B_{8\equiv 0}$.

6.3 Heap 3: skipping empty buckets using a bit-tree

We observe that `extractMin`(H) in Heaps 1 and 2 may end up checking an excessive number of empty buckets when the distribution of edge lengths is such that many approximate level-sets of d are empty (lines 3.1-3.2 and 5.2-5.3). For example, this is likely to happen when $\ell_{min} \ll \ell_{max}$. Heap 3 takes precautions to avoid unnecessary checks of empty buckets.

The basic idea is to use a bit-tree to track which buckets are empty and which are not. A bit-tree is a word-based tree of binary flags. In our case, leaf nodes are mapped to the empty vs. non-empty status of a particular bucket (and each bucket has a corresponding leaf node in the bit tree). A '1' located

Algorithm 4: `updateValue`(H, v) for ring array version (Heap 2).

```
Input: Heap $H$ (of type Heap 2) and a node $v$.
Output: Updates the position of $v$ within $H$ based on $D(v)$, adding $v$ to $H$ if
        necessary.
1 $k_{old}$ = currentBucketID($v$) ;
2 $k$ = mod($\lfloor D(v)/\ell_{min} \rfloor, \beta + 1$) ;
3 if $k_{old} == k$ then
4     return;
5 if $k_{old} < \infty$ then
6     RemoveFromList($B_{k_{old}}, v$) ;
7 AddToListFront($B_k, v$) ;
```

Algorithm 5: `extractMin`(H) for ring array version (Heap 2)

```
Input: Heap $H$ (of type Heap 2).
Output: Node $v$ such that $D(v) - \min_{u \in H} D(u) < \ell_{min}$.
1 $\hat{k}_f$ = mod($\hat{k} - 1, \beta + 1$) ;
2 while $\hat{k} < \hat{k}_f$ and IsEmpty($B_{\hat{k}}$) do
3     $\hat{k}$ = mod($\hat{k} + 1, \beta + 1$) ;
4 if IsEmpty($B_{\hat{k}}$) then
5     return NULL ;
6 return PopList($B_{\hat{k}}$) ;
```

in i-th position of the word stored at bit-tree-node τ indicates that one of the descendants (buckets in our case) of τ's i-th child is non-empty. The specific details of bit-tree implementation are presented in the appendix; however, it is important to know that the bit-tree provides the following accessing functions:

– `markPositionNonEmpty`(k) informs the bit-tree that B_k is non-empty in time $O(\log_w(\beta + 1))$.
– `markPositionEmpty`(k) informs the bit-tree that B_k is empty in time $O(\log_w(\beta + 1))$.
– `nextNonEmptyPosition`(k) returns the index h of the next non-empty bucket after (and including) B_k. If B_k is nonempty then the runtime is $O(1)$, and otherwise the runtime is $O(\log_w(\beta + 1))$.

See Section 7 for the derivation of the runtimes associated with our application of the bit-tree, recall that w is the word length of the computer being used.

The versions of `updateValue`(H, v) and `extractMin`(H) that are modified for use with the new data structure appear in Algorithms 6 and 7, respectively. Note that `nextNonEmptyPosition`(k) ignores the ability to loop around the array, which is why it is called a second time on line 7.3.

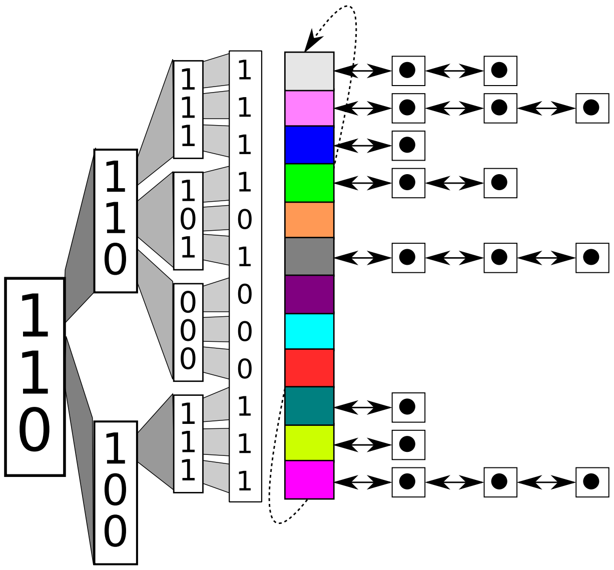

Fig. 5: A bit-tree is used to facilitate faster `extractMin`(H) operations in cases where K is large (i.e., when there are many empty buckets). Each node in the bit tree is associated with an integer of w bits, where the i-th significant bit represents whether or not any of that node's i-th child's descendants (self inclusive) are associated with non-empty buckets. In this figure $w = 3$. This allows us to find the next non-empty bucket in $O(\log_w(\frac{\ell_{max}}{\ell_{min}} + 1))$ time. The bit-tree is described in more detail in the appendix.

Algorithm 6: `updateValue`(H, v) for ring array with bit-tree version (Heap 3).

```
Input: Heap $H$ (of type Heap 3) and a node $v$.
Output: Updates the position of $v$ within $H$ based on $D(v)$, adding $v$ to $H$ if
        necessary.
1 $k_{old}$ = currentBucketID($v$) ;
2 $k$ = mod($\lfloor D(v)/\ell_{min} \rfloor, \beta + 1$) ;
3 if $k_{old} == k$ then
4     return;
5 if $k_{old} < \infty$ then
6     RemoveFromList($B_{k_{old}}, v$) ;
7 AddToListFront($B_k, v$) ;
```

7 Analysis

7.1 Correctness of Heap 1

We now show that Dijkstra's algorithm is correct (terminates in finite time and produces a shortest path from all nodes $v \in V$) when using our heaps. In this

Algorithm 7: `extractMin`(H) for ring array with bit-tree version (Heap 3)

```
Input: Heap $H$ (of type Heap 3).
Output: Node $v$ such that $D(v) - \min_{u \in H} D(u) < \ell_{min}$.
1 $\hat{k}$ = nextNonEmptyPosition($\hat{k}$) ;
2 if $\hat{k} == \infty$ then
3     $\hat{k}$ = nextNonEmptyPosition(0) ;
4     if $\hat{k} == \infty$ then
5         return NULL ;
6 $v$ = PopList($B_{\hat{k}}$);
7 if IsEmpty($B_{\hat{k}}$) then
8     markPositionEmpty($\hat{k}$);
9 return $v$ ;
```

subsection we consider Heap 1, and then extend the results to Heap 2 in the next subsection (the extension to Heap 3 is implied by the extension to Heap 2). The only significant difference between our version of Dijkstra's algorithm and the original is the heap data structure that is used. Therefore, a sufficient proof for the correctness of our modified Dijkstra's is to demonstrate that a node v is removed from the top of our heap only if $d(v) = D(v)$, i.e., the shortest path from v to s has already been computed.

In order to achieve this, we require a few intermediate results regarding the way that nodes in different buckets interact.

Lemma 1. *If v_i and v_j are currently in the same bucket B_k, then $D(v_i)$ cannot be decreased via any edge ε_{ji} from node v_j to node v_i.*

Proof. (by contradiction). If using ε_{ji} decreases $D(v_i)$ then clearly $D(v_i) > D(v_j) + \|\varepsilon_{ji}\|$, which can be rearranged $D(v_i) - D(v_j) > \|\varepsilon_{ji}\|$. By construction $\|\varepsilon_{ji}\| \geq \ell_{max}$, and so it follows that $D(v_i) - D(v_j) > \ell_{max}$. However, v_i and v_j are in the same bucket, and thus $D(v_i) - D(v_j) < \ell_{max}$ by construction, which is a contradiction. □

Lemma 2. *If v_i is in bucket B_k and v_j is in bucket B_h, where $k < h$, then $D(v_i)$ cannot be decreased via edge ε_{ji} from node v_j to node v_i.*

Proof. By construction $D(v) < D(u)$ for all u, v such that $v \in B_k$ and $u \in B_h$ where $h > k$. Also by construction all edge lengths, including $\|\varepsilon_{ji}\|$, are non-negative. Therefore, $D(v_i) < D(v_j) + \|\varepsilon_{ji}\|$. □

Recall that "Processing" v_i is the act of removing v_i from the heap and updating its neighbors $v \in \{v_j \mid (v_i, v_j) \in E\}$ with respect to any path-length decreases that can be achieved via edges from v_i. Processing v_i reduces the path length of its neighbor v_j if and only if $D(v_i) + \|\varepsilon_{ij}\| < D(v_j)$. A reduced path length at v_j may decrease the bucket in which v_j resides or moves v_j into an initial bucket if it is not already in a bucket. The reasoning in the following Lemma is very similar to that in Lemma 1.

Lemma 3. *Processing $v_i \in B_k$ cannot move v_j into B_k.*

Proof. (by contradiction). In order for v_j to move into B_k then its path length (after processing v_i and moving v_j into B_k) is $D(v_j) = D(v_i) + \|\varepsilon_{ij}\|$. Rearranging gives $D(v_j) - D(v_i) = \|\varepsilon_{ij}\|$. By construction $\|\varepsilon_{ij}\| \geq \ell_{min}$, and substituting gives $D(v_j) - D(v_i) \geq \ell_{min}$. However, v_i and v_j are in the same bucket B_k, and thus $D(v_j) - D(v_i) < \ell_{max}$ by construction, which is a contradiction. □

It is important to note that v_j may already be in B_k before v_i is processed, Lemma 3 just guarantees that its existence in B_k is not caused by processing v_i.

Lemma 4. *Processing $v_i \in B_k$ cannot move v_j into B_h, where $h < k$.*

Proof. By construction edges are non-negative, including ε_{ij}. Moving v_j into a bucket B_h such that $h < k$ and $v_i \in B_k$ (immediately prior to the processing of v_i) would require $\varepsilon_{ij} < 0$. □

Note that Lemmas 3 and 4 together guarantee that we are able to process buckets in the order of increasing bucket index.

We are now ready to prove our main result.

Theorem 1. *$v \in B_k$ is removed from Heap 1 only if $d(v) = D(v)$.*

Proof. (By induction).

Base case: B_0 contains s and nothing else (since all other nodes are at least as far from s as ℓ_{min}). s has the correct shortest path of $d(s) = D(s) = 0$ by definition, and is processed first by construction.

Inductive step on index k: Assuming nodes v' from buckets B_0 through B_{k-1} have correctly calculated $d(v') = D(v')$, and B_{k-1} is now empty, we must prove that all nodes v in B_k are guaranteed to have $d(v) = D(v)$.

More formally, we must demonstrate that $d(v') = D(v')$ for all $v' \in \bigcup\{u \mid u \in B_h \wedge 0 \leq h < k\}$ guarantees $d(v) = D(v)$ for all $v \in B_k$.

Lemmas 1 and 2 guarantee that when any $v \in B_k$ is processed there are no nodes in the heap (including B_k) that are *currently* able to reduce $D(v)$.

Lemmas 3 and 4 guarantee that all future heap additions and reshuffling involving any node $u' \in \bigcup\{u \mid u \in B_{\hat{h}} \wedge \hat{h} \geq k\}$ cannot move u' into bucket B_k or a lower bucket B_h, where $h < k$. This is necessary for the inductive step to work, but more importantly guarantees that, for all $v \in B_k$, no combination of *future* heap operations will ever yield a node u' that could have reduced $D(v)$.

Finally, because all nodes $v' \in \bigcup\{v \in B_h \mid 0 \leq h < k\}$ were processed correctly (via inductive assumption), we can conclude that all edges from any $v \in B_k$ to $v' \in \bigcup\{v \in B_h \mid 0 \leq h < k\}$ must have been evaluated and, by construction, the best of them has been used to calculate $D(v)$. Thus, $d(v) = D(v)$ for all $v \in B_k$ even before any member of B_k is processed. □

The correctness of Dijkstra's Algorithm using our data structure is a corollary that follows from Theorem 1 and the original completeness proof of Dijkstra's algorithm.

Corollary 1. *Dijkstra's algorithm using Heap 1 is correct.*

7.2 Correctness of Heap 2

When using Dijkstra's algorithm with Heap 1, during the processing of node $v \in B_k$ it is impossible that any of v's neighbors will require reshuffling into B_h, where $h > k + \beta$, this is formalized in the following theorem, and implies that we are able to use the ring-based array of Heap 2 (and by extension Heap 3).

Theorem 2. *At most $\beta + 1$ levels are needed simultaneously when using Heap 1 with Dijkstra's Algorithm.*

Proof. (by contradiction). If we do need more than $\beta + 1$ levels, then at some point during the algorithm's execution it must be the case that we are processing a node v_i with a neighbor v_j such that: v_i is in bucket B_k (prior to processing), and v_j is in some B_h, where $h \geq \beta + 1$ (after processing). Thus $D(v_i) \in \big((k-1)\ell_{min}, k\ell_{min}\big]$, and $D(v_j) \in \big((k+\beta)\ell_{min}, \infty\big)$, and so $\|\varepsilon_{ij}\| = D(v_j) - D(v_i) > \beta\ell_{min}$. However, $\beta\ell_{min} \geq \ell_{max} \geq \|\varepsilon_{ij}\|$ also by construction, which is a contradiction. □

Corollary 2. *Dijkstra's algorithm using Heap 2 and Heap 3 is correct.*

7.3 Runtime and runspace of Heap 1 and Heap 2

We assume a RAM computational model that is similar to the architectures used by most modern digital computers. We assume that addition, subtraction, multiplication, division, modulo, comparison, type conversion, and pointer operations are all $O(1)$. See Section 9 for an alternative version that replaces division and modulo with integer bit shift (and float to integer casting if edge weights are floating point numbers).

Finding the values ℓ_{min} and ℓ_{max} can be achieved in time $O(|E|)$ by scanning all edges.

Each call to `updateValue`(H, v) requires $O(1)$ time. In particular, we access a bucket (or skip this step if v is not currently in the heap), calculate a new bucket index and then add a node to a doubly linked list. The former requires a floating-point division and conversion (with truncating) of a floating-point number to an integer. In Heap 2 we also require a modulo operation.

There are two cases for `extractMin`(v). In the first, there is at least one element in the current 'top' bucket; this requires an array look-up to find the bucket's list head and also the removing of a node from the head of a doubly-linked list. In the second, we must move down the bucket array until we find a populated bucket. In the worst case, most buckets are empty and over the entire run of the algorithm look-up operations require a cumulative time of

$K = O(\frac{d_{max}}{\ell_{min}})$ to move down the array (since there are that many array positions in Heap 1, and the index of the bucket used for processing is non-decreasing). Alternatively, we observe that Theorem 2 guarantees that there will never be any more than β empty buckets *in a row*, at least until the heap is empty. Thus we also know that $K = O(\beta|V|) = O(\frac{\ell_{max}}{\ell_{min}}|V|)$. Together, these observations provide the following combined bound on K of:

$$K = O\left(\min\left\{\frac{d_{max}}{\ell_{min}}, \frac{\ell_{max}}{\ell_{min}}|V|\right\}\right).$$

`updateValue`(H, v) is called at most $|E|$ times, while `extractMin`(v) is called at most $|V|$ times. Combining results (and remembering to account for the traversal over empty buckets), we find that the running time for Dijkstra's algorithm using either Heap 1 or Heap 2 is $O(|E| + |V| + K)$.

The total space required by Heap 1 is an array of size $\lceil\frac{d_{max}}{\ell_{min}}\rceil$ plus a collection of doubly linked lists that cumulatively require no more than $|V|$ storage containers, for a heap space requirement of $O(|V| + \frac{d_{max}}{\ell_{min}})$, and thus a total of $O(|E| + |V| + \frac{d_{max}}{\ell_{min}})$ for the modified version of Dijkstra's using Heap 1.

Heap 2 reduces this by using a ring array that only requires $\lceil\frac{\ell_{max}}{\ell_{min}}\rceil + 1$ positions. Thus the total space required for Heap 2 is $O(|V| + \frac{\ell_{max}}{\ell_{min}})$, and so the modified version of Dijkstra's using Heap 2 requires space $O(|E| + |V| + \frac{\ell_{max}}{\ell_{min}})$.

We note that the number of list containers stored at any particular time is no greater than the number of nodes in $\beta + 1$ contiguous level-set buckets, which could be used to calculated tighter bounds in some special cases.

7.4 Runtime and Runspace of Heap 3

Heap 3 uses a bit-tree which additionally assumes the word operations of 'shift by x bits' and 'return location of the first non-zero bit' each run in $O(1)$. It provides the subroutines `markPositionNonEmpty`(k), `markPositionEmpty`(k), and `nextNonEmptyPosition`(k). The first two run in time $O(\log_w(\beta + 1))$ (In the worst case, they each involve moving from a leaf of the bit-tree to its root, flipping one bit at each level, and the depth of the tree is $\log_w(\beta + 1)$). There are two cases for the third; if B_k is nonempty then the runtime is $O(1)$, and otherwise the runtime is $O(\log_w(\beta+1))$ (the worst case involves moving from a leaf to the root and then back down to another leaf). The first two are each called once in `updateValue`(H, v), while the latter is called at most twice in `extractMin`(v). All together this means that Dijkstra's algorithm using Heap 3 runs in time $O((|E| + |V|)\log_w(\frac{\ell_{max}}{\ell_{min}} + 1))$.

Heap 3 also requires additional space vs. Heaps 1 and 2. In particular, $O(\frac{\ell_{max}}{\ell_{min}} \log_w\lceil\frac{\ell_{max}}{\ell_{min}}\rceil)$ words are used to store the word-length tree. For a total of $O(|V| + \frac{\ell_{max}}{\ell_{min}} \log_w\lceil\frac{\ell_{max}}{\ell_{min}}\rceil)$ for the Heap 3 alone and $O(|E| + \frac{\ell_{max}}{\ell_{min}} \log_w\lceil\frac{\ell_{max}}{\ell_{min}}\rceil)$ for Dijkstra using Heap 3.

8 Undirected graphs with non-negative edge weights

In the case of undirected graphs, Dijkstra's algorithm using any of the aforementioned heaps can be extended to handle non-negative edges using straightforward pre-processing and post-processing routines that run in time $O(|E| + |V|)$. The key insight is that, for undirected graphs, if there exists a zero-length path between two nodes v and u, then v and u have the same shortest-path lengths vs. all nodes in V. Thus, we may treat such v and u as a single node for the purposes of SSSP, allowing us to solve instead a dual of the original problem that contains no zero-length edges.

In pre-processing (Algorithm 11 presented in the Appendix) we essentially combine each $v \in V$ with all nodes that it can reach in 0 distance, i.e., all u such that $\|P^*(v, u)\| = 0$, into a "meta-node" $\hat{v}$. The resulting set of meta-nodes is $\hat{V}$, and the resulting set of edges between such meta-nodes is $\hat{E}$. This can be done in both time and space $O(|E| + |V|)$ by using one level of abstraction that also adds $O(1)$ operations, vs. the basic version of the algorithm, each time we touch a meta-node or an edge between meta-nodes (in particular, one or two pointer operations). Note that $|\hat{V}| \leq |V|$ and $|\hat{E}| \leq |E|$.

Instead of solving the original problem (V, E), we run Dijkstra's Algorithm using either Heap 2 or Heap 3 on the dual problem involving $(\hat{V}, \hat{E})$ while simultaneously *ignoring* any edges of the form $(\hat{v}, \hat{v})$. The latter removes zero-length edges from consideration without affecting the dual's solution.

Post-processing (Algorithm 12 presented in the Appendix) involves "unpacking" each $\hat{v} \in \hat{V}$ by transferring $d(\hat{v})$ to $d(v)$ for all v that were combined into $\hat{v}$ during pre-processing. All sub-paths that moves through a sequence of nodes $v_1, v_2, \ldots, v_i$, such that $v_1, v_2, \ldots v_i$ are all part of the same $\hat{v} \in \hat{V}$, have length 0. This fact allows us to recover shortest-path parent pointers with respect to the original problem in a single $O(|E| + |V|)$ pass over the original graph and the dual's shortest-path tree. For each of the dual's meta-nodes $\hat{v} \in \hat{V}$ we perform breadth first search *restricted to zero-length edges* (and thus over only the sub-graph containing the nodes from which $\hat{v}$ was created). Even though one (restricted) breadth-first search is performed per each $\hat{v} \in \hat{V}$, we only touch each original edge $\varepsilon \in E$ once during post-processing, since each $v \in V$ is part of exactly one $\hat{v} \in \hat{V}$, and so post-processing also takes time and space $O(|E| + |V|)$.

9 Remarks

9.1 Regarding infinite edge lengths

There are two slightly different notions of "infinite" edge weight that the methods presented above have conflated for notational convenience and the sake of using standard practice. In particular, 'no edge exists between two nodes' has been combined with 'an edge of infinite length exists between two nodes' due to the definition of $\|\varepsilon\| \equiv \infty$ for $\varepsilon \notin E$.

In cases where this distinction is important, we can regain the distinction by storing two separate "external" buckets B_∞ and $B_\emptyset$ instead of just one, and using instead $\|\varepsilon\| \equiv \emptyset$ for $\varepsilon \notin E$. We implicitly assume that un-inserted nodes reside in $B_\emptyset$. If an infinite-length edge connects v and u, where $v \in B_\emptyset$ and u is processed, then we place v into B_∞ (which should now be implemented as a linked list instead of an array). Finally, after all nodes have been processed from the normal heap H, we process the nodes in B_∞, while moving any newly connected nodes from $B_\emptyset$ to B_∞ (the latter is allowed because all remaining best-path lengths are either infinite or undefined).

This modification handles infinite length edges as a separate special case that is guaranteed to run in $O(|E| + |V|)$ time. Therefore, much of the analysis is unaffected. The only technical detail worth mentioning (indeed, a convention we have been using throughout our discussion) is that we must carefully define ℓ_{max} as the longest *finite*-length edge, and d_{max} as the longest finite-length shortest-path.

9.2 Regarding division and modulo operations

On computational architectures where division takes $\Omega(1)$ time, we can store the reciprocal of ℓ_{min} instead of ℓ_{min}, and then perform multiplication instead of division. The reciprocal only needs to be calculated once; and so, assuming we have some way of computing the reciprocal in time $O(|E|+|V|)$, this modification does not affect our runtime analysis.

On (binary) computational architectures where the modulo operation takes $\Omega(1)$ we can replace the modulo operation with bit-shifting operations by replacing β with $\bar{\beta} = 2^n - 1$, where 2^n is the *smallest* power of 2 such that $2^n - 1 \geq \lceil \frac{\ell_{max}}{\ell_{min}} \rceil$. This makes the circular array in Heaps 2 and 3 slightly longer than necessary, so that it can be a power of two, but causes all modulo operations to have the form $\texttt{mod}(x, 2^n)$, where n is constant. For integers on standard binary architectures $\texttt{mod}(x, 2^n)$ can be implemented using the up $<<$ and down $>>$ shifting operations as follows: $\texttt{mod}(x, 2^n) \equiv (x - ((x >> n) << n))$, and which uses $O(1)$ time, assuming that bit shifts and subtraction are $O(1)$.

In cases where we want to remove the modulo operation and there are floating point edge weights, we need to convert $x = \lfloor D(v)/\ell_{min} \rfloor$ to an integer from a float; thus, this strategy still requires that such a casting operation runs in time $O(1)$. With the above assumptions, these modifications do not affect our asymptotic runtime and space bounds because we are simply using an array of length $\bar{\beta} < 2\lceil \frac{\ell_{max}}{\ell_{min}} \rceil$ instead of $\beta = \lceil \frac{\ell_{max}}{\ell_{min}} \rceil$.

9.3 Regarding runtime

For the subset of SSSP instances involving strictly positive edge weights, as well as those involving undirected graphs with non-negative edges Section 8, the methods we present tie the previous best known bounds on runtime whenever $K \leq |V| + |E|$. This happens whenever $\frac{d_{max}}{\ell_{min}} \leq |V| + |E|$ and also whenever $\frac{\ell_{max}}{\ell_{min}}|V| \leq |E|$ (and other cases may also exist).

The constant factors that affect practical performance are easy to calculate (or bound) and give direct insight into when the algorithm should be expected to perform well or poorly. For example, the method will perform better as the relative difference between ℓ_{max} and ℓ_{min} decreases.

As previously remarked, K is the total number of approximate level sets of d that turn out to be empty. For many types of graphs K is guaranteed to be small. The simplest case is when all edge weights are the same, and in which case the algorithm runs in linear time. This happens for problems in which path length is defined by the number of edges in the path (e.g., social degrees-of-separation). It also happens on 4-grids and other uniform lattices, which are frequently used for path-planning.

In some practical cases of interest (e.g., trajectory library based robotic motion planning) we also have the power to design the graph that we intend to plan over. In these cases we can manually enforce that edge lengths are "close-enough" to each other that the algorithm runs at a desired speed (for example, within a few orders of magnitude of each other).

On the other hand, there are many types of graphs that are *not* well suited (at least theoretically) to our method. For example, using Heap 3 with Graphs that are created by sampling points in space uniformly at random (e.g., randomly sampled disc-graphs) will tend to see long-term (but relatively slow) performance degradation vs. an increasing number of nodes, since the ratio $\frac{\ell_{max}}{\ell_{min}}$ is expected to increase without bound as $|V|$ approaches infinity in that case. On the other hand, Heap 1 and 2 will suffer due to decreasing ℓ_{min} but also benefit from the fact that the chances a particular level set is empty will approach 0, in the limit, as $|V|$ approaches infinity. A particularly bad case for any version of our heap happens when a graph simultaneously has relatively few edges but edge lengths exist at different orders of magnitude.

9.4 Potential for Parallelization

The idea of parallel processing nodes based on any partial ordering that guarantees best-path-length independence between subsets of nodes is, in general, amenable to Parallelization on multi-core architectures. in order to ensure such a parallel implementation is correct, synchronization between processors is (only) necessary after each bucket is emptied. For our data structure this will obviously work best in cases where each bucket contains many nodes and/or ℓ_{max} and ℓ_{min} are similar.

10 Summary

We have presented a family of new heap data-structures that enable Dijkstra's Algorithm to solve the SSSP with positive edge-weights in either time $O(|E| + |V| + K)$ and space $O(|E|+|V|+\frac{\ell_{max}}{\ell_{min}})$ or time $O((|E| + |V|) \log_w(\frac{\ell_{max}}{\ell_{min}} + 1))$ and space $O(|E| + \frac{\ell_{max}}{\ell_{min}} \log_w \lceil \frac{\ell_{max}}{\ell_{min}} \rceil)$, where ℓ_{min} and ℓ_{max} are problem dependent constants and w is an architecture dependent constant. We derive bounds $K = O(\min \left\{ \frac{d_{max}}{\ell_{min}}, \frac{\ell_{max}}{\ell_{min}}|V| \right\})$. Thus, our method is able to solve many instances of SSSP in linear time, e.g., whenever $\frac{d_{max}}{\ell_{min}} \leq |V| + |E|$ and also whenever $\frac{\ell_{max}}{\ell_{min}}|V| \leq |E|$.

The method works because, as we prove in Section 7, it is possible to solve the positive weight SSSP using only a partial ordering over nodes instead of a full ordering, as long as we choose that partial ordering carefully. In particular, by using approximate level-sets based on the shortest-path-lengths of nodes. Even though shortest-path-lengths are initially unknown for all nodes but the start node, we can correctly calculate the partial ordering on-the-fly if the widths of each approximate level set are bounded by the graph's shortest edge length. Nodes within an approximate level set can be processed in any order by a standard implementation of Dijkstra's algorithm and still yield the correct solution. The method can be extended to non-negative edges for the special case of undirected graphs using pre-/post-processing to solve a dual of the original problem that ignores zero-length edges.

The heaps that we present store the approximate level sets in an array of buckets, where each bucket is a doubly linked list. We show that only a small and calculable subset of contiguous buckets are only ever used at one time, and so we can use a looping array that saves considerable space. In particularly challenging cases many empty buckets exist (which may slow down heap extraction operations), therefore, the "less-simple" version uses a bit-tree to quickly determine the next non-empty bucket.

The heaps that we present are easy to implement, and the resulting modification of Dijkstra's algorithm efficient vs. time and space, and easy to analyze. We hope that the concept of level-set based partial orderings will increase the overall understanding of the SSSP and perhaps even be useful in other domains.

Acknowledgments

This work was supported by the Control Science Center of Excellence at the Air Force Research Laboratory (AFRL), the National Science Foundation (NSF) grant IIP-1161029, and the Center for Unmanned Aircraft Systems. This work was completed while the author was "in residence" at AFRL. We would also like to acknowledge Erik Komendera, who independently discovered but never published a data structure similar to Heap 1 while working on an unrelated problem, and thank him for his discussions involving it. The author would also like to thank Dmitry Yershov and Wheeler Ruml for pointing our relevant related work.

References

1. Ravindra K Ahuja, Kurt Mehlhorn, James Orlin, and Robert E Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM (JACM)*, 37(2):213–223, 1990.
2. Yasuhito Asano and Hiroshi Imai. Practical efficiency of the linear time algorithm for the single source shortest path problem. *Journal of the Operations Research*, 43:431–447, 2000.

3. Boris V Cherkassky, Andrew V Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming*, 73(2):129–174, 1996.
4. Boris V Cherkassky, Andrew V Goldberg, and Craig Silverstein. Buckets, heaps, lists, and monotone priority queues. *SIAM Journal on Computing*, 28(4):1326–1346, 1999.
5. Robert B Dial. Algorithm 360: Shortest-path forest with topological ordering [h]. *Communications of the ACM*, 12(11):632–633, 1969.
6. Cornelius Diekmann. An unusual way of solving the sssp problem. 2010.
7. Henry Gordon Dietz. The Aggregate Magic Algorithms. Technical report, University of Kentucky.
8. Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
9. Greg N Federickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.
10. Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.
11. Michael L Fredman and Dan E Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 719–725. IEEE, 1990.
12. Michael L Fredman and Dan E Willard. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993.
13. Kurt Mehlhorn and Stefan Näher. Bounded ordered dictionaries in o (log log n) time and o (n) space. *Information Processing Letters*, 35(4):183–189, 1990.
14. James B Orlin, Kamesh Madduri, K Subramani, and Matthew Williamson. A faster algorithm for the single source shortest path problem with few distinct positive lengths. *Journal of Discrete Algorithms*, 8(2):189–198, 2010.
15. Nick Prühs. Implementation of thorup's linear time algorithm for undirected single-source shortest paths with positive integer weights. *B. S. thesis, University of Kiel, Kiel, Germany*, 2009.
16. Rajeev Raman. Priority queues: Small, monotone and trans-dichotomous. In *AlgorithmsESA'96*, pages 121–137. Springer, 1996.
17. Rajeev Raman. Recent results on the single-source shortest paths problem. *ACM SIGACT News*, 28(2):81–87, 1997.
18. K Subramani and Kamesh Madduri. Two-level heaps: a new priority queue structure with applications to the single source shortest path problem. *Computing*, 90(3-4):113–130, 2010.
19. Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394, 1999.
20. Mikkel Thorup. Floats, integers, and single source shortest paths. *Journal of Algorithms*, 35(2):189–201, 2000.
21. Mikkel Thorupf. On ram priority queues. 81:59, 1996.
22. John N Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE TRANSACTIONS ON AUTOMATIC CONTROL*, 40(9), 1995.
23. Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information processing letters*, 6(3):80–82, 1977.
24. Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127, 1976.

25. Yusi Wei and Shojiro Tanaka. An improved thorup shortest paths algorithm with a modified component tree. In *Natural Computation (ICNC), 2013 Ninth International Conference on*, pages 1160–1165. IEEE, 2013.
26. Yusi Wei and Shojiro Tanaka. Improvement of thorup shortest path algorithm by reducing the depth of a component tree. *Journal of Advances in Computer Networks*, 2(2), 2014.
27. John William Joseph Williams. Heapsort. *Communications of the ACM*, 7:347–348, 1964.

Bit-Tree Implementation

Each tree-node τ in the bit-tree portion of the looping word-length based tree data structure T has the following fields (we shall denote the "field of" relationship using a '.' (dot):

- A flag indicating if the node is a leaf node or not L, with value true or false respectively.
- A word $\mathcal{B}$ of bits of word-length w, where w determined by the computer hardware being used. We shall denote the i-th bit in $\mathcal{B}$ as $\mathcal{B}[i]$. This is only used in leaf nodes.
- An array C of w pointers to the w children of the current tree-node (these are to other tree nodes if this node is internal, and to a particular bucket if the node is a leaf). We shall index into the array using '$[\cdot]$'. This is also only used in leaf nodes.
- A pointer to the parent of the current tree-node p.
- An integer i_p that represents this node's location in its parent's C.
- Leaf nodes also store $\hat{k}$, the index of the bucket they are associated with.

Additionally, we store two arrays $\mathcal{A}$ and $\mathcal{P}$ that each have one element for every leaf node in the bit tree. We enforce that $\mathcal{A}[k] == 1$ if B_k is nonempty, and $\mathcal{A}[k] == 0$ otherwise. $\mathcal{P}[k]$ contains a pointer to the leaf node that is associated with B_k

The tree is designed such that there is one path from the root τ_0 of the tree to each of the $\beta + 1$ buckets B in the bucket array. Let $\tau_{d,k}$ denote the tree-node at depth d along the path from the root to bucket B_k. If a particular bucket B_k is nonempty, then we enforce the condition that $\tau_{d,k}.\mathcal{B}[i] = 1$, where i is the position of the pointer to $\tau_{d+1,k}$ that is stored in $\tau_{d+1,k}.C$.

This has the advantage of letting us quickly determine when none of the descendant buckets of τ have any nodes in them (i.e., because $\tau_{d,k}.\mathcal{B} = 0$ in that case). In the event that descendant buckets of τ are nonempty, then it also allows us to quickly determine the earliest tree-child of τ that also has nonempty descendant buckets, i.e. $\diamond(\tau_{d,k}.\mathcal{B})$, using the 'first non-zero bit' primitive which we shall denote '$\diamond$'. Note that we assume bit indexing that starts at 1 (if architecture indexing starts at 0, then we need only add one to the result, which requires $O(1)$ time).

Many common architectures (e.g., X86, ARM, SPARC, etc.) implement a 'count trailing zeros' primitive, $\texttt{ctz}(x)$, that runs in $O(1)$ time, and which

Algorithm 8: `nextNonEmptyPosition`(T, k)

```
Input: Bit-tree $T = (\mathcal{A}, \mathcal{P}, \bigcup\{\tau\})$ of bit-word-nodes $\tau$ and integer $k$.
Output: The next non-empty position of $T$ after (and including) $k$.
1  if $\mathcal{A}[k] == 1$ then
2      return $k$ ;
3  $\tau = \mathcal{P}[k]$ ;
4  while $\tau \neq \tau_0$ do
5      $i_{next} = \diamond((\tau.p.\mathcal{B} >> \tau.i_p) << \tau.i_p)$
6      if $i_{next} > 0$ then
7          break ;
8      $\tau = \tau.p$ ;
9  if $\tau == \tau_0$ then
10     return $\infty$ ;
11 $\tau = \tau.p.C[i_{next}]$ ;
12 while $\tau.L \neq 1$ do
13     $\tau = \tau.C[\diamond(\tau.\mathcal{B})]$ ;
14 return $\tau.\hat{k}$ ;
```

can be used to implement $\diamond(x)$ as follows: $\diamond(x) = w - \texttt{ctz}(x)$. For architectures that have neither 'first non-zero bit' nor 'count trailing zeros' primitives the function $\diamond(x)$ can be implemented in software with runtime $O(\log_2 w)$, see [7] for details. In the latter case that $\diamond(x)$ is implemented in software (and not hardware), the overall runtime of Dijkstra's algorithm using Heap 3 becomes $O((|E| + |V|)\log_2(w)\log_w(\frac{\ell_{max}}{\ell_{min}} + 1))$. Although this represents a slightly larger architecture dependent constant, it does not change the overall validity of our qualitative claim of 'linear time performance on many graphs with small overhead'.

We now describe in more detail the subroutines `nextNonEmptyPosition`(k), `markPositionEmpty`(k), and `markPositionNonEmpty`(k) that are used to interact with the bit-tree that is used in the bit-tree version of our data structure. The i-th bit of the integer k is denoted $k[i]$

`nextNonEmptyPosition`(k) appears in Algorithm 8. A quick check is used to see if the bucket at index k is empty, and if not then k is returned directly, line 1. If B_k is empty, then we walk up the tree using parent pointers until we find a parent that has a later child with non-empty descendants or we reach the bit-tree-root, lines 7. Note that '$>>$' and '$<<$' are the decreasing and increasing bit-shift operators, respectively. If we have reached the root, then we we know that there were no nonempty buckets after B_k, and so we return ∞, lines 9-10. Otherwise, we now move down through the branches of the tree toward the first non-empty bucket, lines 12-13, and return it on line 14.

`markPositionEmpty`(k) is presented in Figure 9. A quick check is used to test if the position k is already marked as empty, and if so then the algorithm returns immediately (lines 1-2). The position is marked as empty on line 4 (where '$\otimes$' is

Algorithm 9: `markPositionEmpty`(T, k)

```
Input: Bit-tree $T = (\mathcal{A}, \mathcal{P}, \bigcup\{\tau\})$ of bit-word-nodes $\tau$ and integer $k$.
Output: Marks position $k$ of $T$ as empty.
1 if $\mathcal{A}[k] == 0$ then
2     return;
3 $\mathcal{A}[k] = 0$ ;
4 $\tau = \mathcal{P}[k]$ ;
5 while $\tau \neq \tau_0$ do
6     $\tau.p.\mathcal{B} = \tau.p.\mathcal{B} \otimes (1 << \tau.i_p)$ ;
7     if $\tau.p.\mathcal{B} > 0$ then
8         return;
9     $\tau = \tau.p$ ;
```

Algorithm 10: `markPositionNonEmpty`(T, k)

```
Input: Bit-tree $T = (\mathcal{A}, \mathcal{P}, \bigcup\{\tau\})$ of bit-word-nodes $\tau$ and integer $k$.
Output: Marks position $k$ of $T$ as nonempty.
1 if $\mathcal{A}[k] == 1$ then
2     return;
3 $\mathcal{A}[k] = 1$ ;
4 $\tau = \mathcal{P}[k]$ ;
5 while $\tau \neq \tau_0$ do
6     if $\tau.p.\mathcal{B} > 0$ then
7         $\tau.p.\mathcal{B} = \tau.p.\mathcal{B}\&(1 << \tau.i_p)$ ;
8         return;
9     $\tau.p.\mathcal{B} = \tau.p.\mathcal{B}\&(1 << \tau.i_p)$ ;
10    $\tau = \tau.p$ ;
```

the exclusive or operation) and then we walk up the bit tree to the root of the largest sub-tree that is now empty due to the removal of B_k, while recording the latter, lines 4-9. Note that we can stop as soon as we find a parent with other non-empty descendants (lines 7-8).

`markPositionNonEmpty`(k) is presented in Figure 10 and is similar to `markPositionEmpty`(k), except that we record information marking B_k as full instead of empty. We can stop walking up the bit-tree as soon as we find a parent with other non-empty descendants (after marking that the path we have taken up the tree is now non-empty as well), lines 6-8.

Pre-/post-processing undirected graphs with zero-length edges

This section presents subroutines that can be used to pre-/post-process an undirected graphs with zero-length edges such that a dual problem (without zero-length edges) can be solved instead of the original problem (with zero-length edges). These subroutines appear in Algorithms 11 and 12, respectively.

Algorithm 11: `preProcess`(V, E) for the extension of our method to undirected graphs with non-negative weights.

```
Input: Node set $V$ and edge set $E$ of the graph $G = (V, E)$ with non-negative
       edge weights.
Output: Node set $\hat{V}$ of a dual problem in which zero-length edges can be
        ignored. The dual's edge set $\hat{E}$ is implicitly defined by $V$, $E$, and $\hat{V}$.
1  $\hat{V}$ ;                              /* array of $|V|$ of linked list heads */ ;
2  $n = 0$ ;
3  for $v \in V$ do
4      $n = n + 1$ ;
5      $\hat{V}[n] = 0$ ;
6      $v.i = 0$ ;                          /* position of $\hat{V}$ containing $v$ */ ;
7  $n = 0$ ;                                /* counts non-empty $\hat{V}$ elements */ ;
8  $Q_{FIFO}$ ;                             /* FIFO queue */ ;
9  for $v \in V$ do
10     if $v.i == 0$ then
11         $n = n + 1$ ;
12         AddToListFront($\hat{V}[n], v$) ;
13         $v.i = n$ ;
14         $u = v$ ;
15         while $u \neq NULL$ do
16             for $(u, w) \in E$ do
17                 if $\|(v, w)\| == 0$ and $w.i == 0$ then
18                     AddToListFront($\hat{V}[n], w$) ;
19                     $w.i = n$ ;
20                     PushBack($Q_{FIFO}, w$) ;
21             $u =$ PopFront($Q_{FIFO}$) ;
22 $\hat{V} = \hat{V}[1 \text{ to } n]$;
23 return $\hat{V}$;
```

Each $v \in V$ of the original problem is combined, along with all nodes u that it can reach in 0 distance (all u such that $\|P^*(v, u)\| = 0$) into a "meta-node" $\hat{v}$ in the dual. Each meta-node contains a particular subset of nodes from the original problem that are zero-distance from each other (and contains all such nodes). Each node from the original problem is associated with exactly one meta-node in the dual. We assume that each node v maintains an internal integer field $v.i$ that stores the (index of) the particular "meta-node" $\hat{v}$ in the dual problem that v is associated with.

With some abuse of notation $v \in \hat{v}$ denotes that pre-processing has placed v into the meta-node $\hat{v}$ and meta-nodes are stored in an array of linked lists, also denoted $\hat{V}$, such that each linked list $\hat{V}[n]$ stores all of the (original) nodes in a particular meta-node $\hat{v}$.

Algorithm 12: `postProcess`$(V, E, s, \hat{V})$ for extension of our method to undirected graphs with non-negative weights.

Input: Node set V and edge set E of the graph $G = (V, E)$ with non-negative edge weights and start node s of the original problem; the dual problem's node set $\hat{V}$ as calculated in the pre-processing step, and assuming that the SSSP has been solved for the dual problem.

Output: The solution to the original SSSP problem (the shortest path tree S and values are stored implicitly in V and E).

```
1  for all $v \in V$ do
2      $d(v) = d(\hat{V}[v.i])$ ;                    /* unpack path-lengths */ ;
3      $p(v) = \text{NULL}$ ;
4  unpackParents$(s, E)$ ;
5  for $n = 1, \ldots, |\hat{V}|$ do
6      if $s.i == n$ then
7          continue ;
8      $\hat{v} = \hat{V}[n]$ ;
9      $v =$ memberWithBestParent$(\hat{v}, E)$ ;
10     unpackParents$(v, E)$ ;
```

Pre-processing is accomplished using `preProcess`(V, E) in Algorithm 11. The subroutine works by walking the graph (V, E). If a node v is found that is not yet associated with a meta-node in the dual, then a new meta-node $\hat{v} = \hat{V}[n]$ is created (lines 10-13) and an internal breadth-first search over zero-length edges (lines 14-21) is performed to find all other nodes that belong in $\hat{v}$. It is important to note that each edge in E is touched at most twice during the entire execution of `preProcess`(V, E), once each in the (u, v) and (v, u) directions. The latter property is guaranteed due to the fact that each node $v \in V$ can belong to at most one meta-node $\hat{v} \in \hat{V}$, and by construction all nodes associated with a particular meta-node $\hat{v}$ are found (using the restricted breadth-first-search) as soon as any node associated with $\hat{v}$ is discovered, and each direction of each edge (v, u) appears in a single restricted breadth-first search.

Post-processing is accomplished using `postProcess`$(V, E, s, \hat{V})$ in Algorithm 12. It involves "unpacking" each $\hat{v} \in \hat{V}$ by transferring $d(\hat{v})$ to $d(v)$ for all $v \in \hat{v}$ (line 2). All sub-paths that moves through a sequence of nodes $v_1, v_2, \ldots, v_i$, such that $v_1, v_2, \ldots v_i \in \hat{v}$, have length 0. This fact allows us to recover shortest-path parent pointers with respect to the original problem in a single $O(|E| + |V|)$ pass over the original graph and the dual's shortest-path tree. The details of the latter are delegated to the subroutines `unpackParents`(v, E) and `memberWithBestParent`$(\hat{v}, E)$, each of which cumulatively touches each edges in E at most twice.

`unpackParents`(v, E) is called once per meta-node $\hat{v} \in \hat{V}$ and uses a breadth-first-search restricted to zero-length edges to find parent pointers among nodes within the same meta-node $\hat{v}$. The restriction to zero-length edges means that

Algorithm 13: `unpackParents`(v, E)

```
Input: A node $v$ and edge set $E$ of the original problem, assuming path-lengths
       have already been unpacked but not parent pointers.
Output: Unpacks the parent pointers for all original nodes in the dual's
        meta-node $\hat{v}$ such that $v$ was part of $\hat{v}$.
1 $Q_{FIFO}$ ;                    /* empty FIFO queue */ ;
2 $u = v$ ;
3 while $u \neq$ NULL do
4     for $(u, w) \in E$ s.t $\|(u, w)\| = 0$ do
5         if $p(w) ==$ NULL then
6             $p(w) = u$ ;
7             PushBack($Q_{FIFO}, w$) ;
8     $u =$ PopFront($Q_{FIFO}$) ;
```

Algorithm 14: $v =$ `memberWithBestParent`$(\hat{v}, E)$

```
Input: A node $v$ and edge set $E$ of the original problem, assuming path-lengths
       have already been unpacked but not parent pointers.
Output: The original node $v$ in the dual's meta-node $\hat{v}$ such that $v$ was part of
        $\hat{v}$ and $v$ has a parent $u$ within the parent meta-node $\hat{u} = p(\hat{v})$. Also
        unpacks the parent pointer for $v$.
1 for $v \in \hat{v}$ do
2     for $(v, u) \in E$ do
3         if $u.i == p(\hat{v})$ then
4             $p(v) = u$ ;
5             return $v$;
6 return NULL;
```

each search is is constrained to consider only those nodes associated with a particular meta-node.

`memberWithBestParent`$(\hat{v}, E)$ is used to find a particular node $v \in \hat{v}$ that may use a parent outside of $\hat{v}$ in a valid shortest-path-tree. In other words, it finds v such that $v \in \hat{v}$ and $(u, v) \in E$ and $(\hat{u}, \hat{v}) \in \hat{E}$ and $\hat{u} = p(\hat{v})$. This routine also “unpacks” the appropriate parent pointer for v (line 4).