

# A Comparison of Dijkstra’s Algorithm Using Fibonacci Heaps, Binary Heaps, and Self-Balancing Binary Trees

Rhyd Lewis

School of Mathematics,  
Cardiff University, Cardiff, Wales.  
*LewisR9@cf.ac.uk, <http://www.rhydlewis.eu>*

March 22, 2023

## Abstract

This paper describes the shortest path problem in weighted graphs and examines the differences in efficiency that occur when using Dijkstra’s algorithm with a Fibonacci heap, binary heap, and self-balancing binary tree. Using C++ implementations of these algorithm variants, we find that the fastest method is not always the one that has the lowest asymptotic complexity. Reasons for this are discussed and backed with empirical evidence.

## 1 Introduction

Dijkstra’s algorithm is an efficient, exact method for finding shortest paths between vertices in edge- and arc-weighted graphs. It is particularly useful in transportation problems when we want to determine the shortest (or fastest) route between two geographic locations on a road network [7, 10, 14]. It is also applicable in areas such as telecommunication, social network analysis, arbitrage, and currency exchange [15, 18].

In this paper, we examine the changes in computational complexity and computing times that occur when using either a self-balancing binary tree, binary heap, or Fibonacci heap within Dijkstra’s algorithm. As part of this work, we give an efficient C++ implementation of the algorithm and of Fibonacci heaps. Many programming languages, including C++, contain versions of self-balancing binary trees and binary heaps as part of their libraries; however, implementations of Fibonacci heaps are less common. Existing C++ implementations of Fibonacci heaps are also buggy, inefficient, and/or difficult to use. This is not the case for the custom implementation used here, which has been fully tested and evaluated.

The next section formally defines the shortest path problem and surveys several algorithms for solving it. Section 3 gives a detailed description of Dijkstra’s algorithm, while Section 4 shows how the efficiency of this method can be improved through the use of priority queues. In Section 5 we describe implementations of four variants of Dijkstra’s algorithm, which are then compared and evaluated in Section 6. Conclusions are drawn in Section 7.

## 2 Problem Definition and Existing Algorithms

Let  $G = (V, A)$  be an arc-weighted, directed graph in which  $V$  is a set of  $n$  vertices, and  $A$  is a set of  $m$  arcs (directed edges). In addition, let  $\Gamma(u)$  denote the set of vertices that are *neighbours* of a vertex  $u$ . That is,  $\Gamma(u) = \{v : (u, v) \in A\}$ . Finally, we also define a nonnegative *weight* (or length)  $w(u, v)$  for each arc  $(u, v) \in A$ . The weight (or length) of a path is defined by the sum of the weights of its arcs.

According to Cormen et al. [9], three problems involving shortest paths on arc-weighted graphs can be distinguished:

**The single-source single-target shortest path problem.** This involves finding the shortest path between a particular *source* vertex  $s$  and *target* vertex  $t$ . In other words, we want to identify the  $s$ - $t$ -path in  $G$  whose length (weight) is minimal among all possible  $s$ - $t$ -paths.

**The single-source shortest path problem.** This involves determining the shortest path from a source  $s$  to all other reachable vertices in  $G$ . In this sense, we are seeking a “shortest path tree rooted at  $s$ ”. An example of such a tree is shown in Figure 1.

**The all-pairs shortest path problem.** This involves finding the shortest path between every pair of vertices in  $G$ . That is, we are seeking the shortest  $u$ - $v$ -paths for all  $u, v \in V$ .

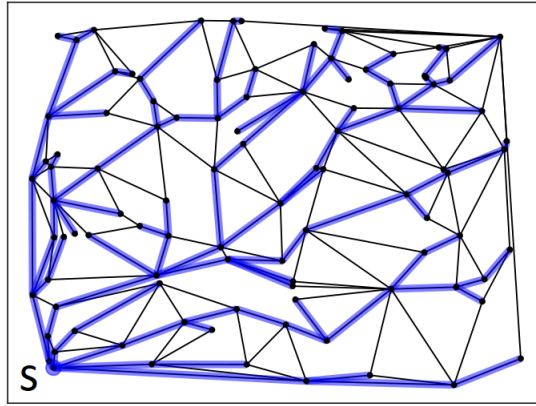


Figure 1: An example graph with  $n = 100$  vertices (the black circles). In this case, arcs are drawn with straight lines, and arc weights correspond to the lengths of these lines. This particular graph is also symmetric in that  $(u, v) \in A$  if and only if  $(v, u) \in A$ , with  $w(u, v) = w(v, u)$ . The highlighted arcs show a shortest path tree rooted at the vertex  $s$ . Because this graph is a strongly connected component, the shortest path tree is also a spanning tree.

In this work, we will use Dijkstra’s algorithm to solve the second problem in the above list. It can, however, also be used for the other two variants. To do this with the single-source single-target shortest path problem, we simply need to halt Dijkstra’s algorithm as soon as the target vertex becomes “distinguished” (see Section 3). For the all-pairs shortest path problem, meanwhile, it is sufficient to execute Dijkstra’s algorithm  $n$  times, using each vertex  $u \in V$  as the source in turn.

Several other algorithms also exist for the above three problems. In cases where graphs feature negative arc weights, a more suitable alternative is the  $O(nm)$ -time Bellman-Ford algorithm [9]. Although this algorithm has a higher growth rate than that of Dijkstra’s, it has the added advantage of being able to detect if a particular instance of the shortest path problem is “ill-defined”, in that it contains negative cycles. Bellman-Ford can also be augmented with additional data structures to form Moore’s algorithm [16] which, though still featuring a complexity of  $O(nm)$ , usually features faster run times than Bellman-Ford.

For the single-source single-target shortest path problem, other specialised algorithms exist, though none of these is known to run asymptotically faster than Dijkstra’s algorithm. One well-known alternative is the A\* algorithm of Hart et al. [11, 12]. This is a heuristic-based variant of Dijkstra’s algorithm and usually gives much faster run times in applications involving transportation networks. Algorithms for variants of the single-source single-target shortest path problem have also been proposed by Yen [19] and Bhandari [8]. Yen’s algorithm is used to find the  $k$  shortest  $s$ - $t$ -paths, where  $k$  is a user-defined parameter. The methods of Bhandari, meanwhile, are used to produce a pair of shortest  $s$ - $t$ -paths that are either edge-disjoint and/or vertex-disjoint.

Finally, alternative algorithms are also available for the all-pairs shortest path problem. One well-known approach is the  $O(n^3)$  Floyd-Warshall algorithm, which is also able to handle graphs containing negative arc weights. Another option in the presence of negative weights is the algorithm of Johnson [13]. This operates by transforming the input graph into a second graph that has no negative weights but which maintains the same shortest-paths structure as the original. This can be achieved through a single application of the Bellman-Ford algorithm. After this,  $n$  applications of Dijkstra’s algorithm can then be performed.

### 3 Dijkstra’s Algorithm

In this section, we give a more detailed description of Dijkstra’s algorithm. We also describe its underlying data structures and derive its complexity.

To produce a shortest-path tree rooted at  $s$ , Dijkstra’s algorithm operates by maintaining a set  $D$  of so-called “distinguished vertices”. Initially, only the source vertex  $s$  is considered distinguished. During execution, further vertices are then added to  $D$ , one at a time, until all reachable vertices have been inserted. Two other data structures are also maintained. First, a “label”  $L(u)$  is stored for each vertex  $u \in V$  in the graph. During execution,  $L(u)$  stores the length of the shortest  $s$ - $u$ -path that uses distinguished vertices only. Consequently, on termination of the algorithm,  $L(u)$  gives the length of the shortest  $s$ - $u$ -path in the graph. If a vertex  $u$  has a label  $L(u) = \infty$ , then no  $s$ - $u$ -path is possible. Finally, a “predecessor”  $P(u)$  is also stored for each vertex  $u \in V$ . During execution,  $P(u)$  stores the vertex that occurs before  $u$  in the shortest  $s$ - $u$  path (of length  $L(u)$ ) that uses distinguished vertices only. If a vertex  $u$  has no predecessor, then  $P(u) = \text{NULL}$ . On termination, these predecessor values can be used to construct the shortest paths from  $s$  to all reachable vertices.

---

**Algorithm 1: Dijkstra’s Algorithm (Basic Form)**

---

- input :** An arc-weighted graph  $G = (V, A)$  and source vertex  $s \in V$   
**output:** A populated label array  $L$  and predecessor array  $P$
- 1 Set  $L(u) = \infty$  and  $P(u) = \text{NULL}$  for all  $u \in V$ . Also let  $D = \emptyset$ , and set  $L(s) = 0$ .
  - 2 Choose a vertex  $u \in V$  such that: (a) its value for  $L(u)$  is minimal; (b)  $L(u)$  is less than  $\infty$ ; and (c)  $u$  is not in  $D$ . If no such vertex exists, then end; otherwise insert  $u$  into  $D$  and go to Step 3.
  - 3 For all neighbours  $v \in \Gamma(u)$  that are not in  $D$ , if  $L(u) + w(u, v) < L(v)$  then set  $L(v) = L(u) + w(u, v)$  and set  $P(v) = u$ . Now return to Step 2.
- 

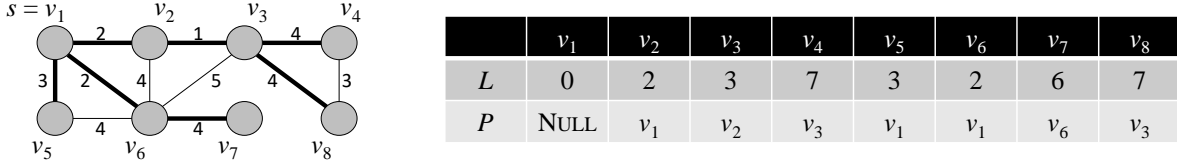


Figure 2: Example output from Algorithm 1 using the indicated eight-vertex graph and source vertex  $s = v_1$ . In this case, the graph is undirected. Consequently, an arc  $(u, v)$  exists if and only if the arc  $(v, u)$  exists. In all cases,  $w(u, v) = w(v, u)$ , as shown. The shortest path tree rooted at  $s$  (defined by  $P$ ) is shown by the bold lines in the graph.

---

**Algorithm 2: GET-PATH**

---

- input :** The arc weighted graph  $G$ , predecessors  $P$ , source vertex  $s$ , and an arbitrary vertex  $u$   
**output:** A vertex sequence  $\pi$  corresponding to the shortest  $s$ - $u$ -path in  $G$
- 1 Let  $\pi = ()$ , and let  $v = u$
  - 2 **if**  $P(u) \neq \text{NULL}$  **then**
  - 3     **while**  $v \neq s$  **do**
  - 4         Append  $v$  to  $\pi$  and set  $v = P(v)$
  - 5     Append  $s$  to  $\pi$  and then reverse  $\pi$
- 

In its most basic form, Dijkstra’s algorithm can now be described by just three steps. These are given in Algorithm 1. In these steps, note that one vertex is inserted into  $D$  at each iteration. This gives  $O(n)$  iterations of the algorithm in total. Furthermore, within each iteration, we need to identify the vertex  $u \notin D$  with the minimum label (an  $O(n)$  operation) and then examine (and possibly update) the labels of all vertices  $v \in \Gamma(u)$ . This leads to an overall complexity of  $O(m + n^2)$ . Since the upper bound of  $m = O(n^2)$ , this complexity can be simplified to  $O(n^2)$ . Output from an example run of this algorithm is shown in Figure 2.

On completion of Dijkstra’s algorithm, the sequence of vertices that occurs in each shortest path starting at  $s$  is stored in  $P$ . The shortest  $s$ - $u$ -path (for all  $u \in V$ ) can be constructed using the GET-PATH procedure shown in Algorithm 2. As shown, this operates by starting at  $u$ , and taking each preceding vertex until the source  $s$  is encountered. The  $s$ - $u$ -path is then the reverse of this sequence. For example, the shortest  $s$ - $v_8$ -path from Figure 2 is written  $\pi = (s, v_2, v_3, v_8)$ . Note that, because all arc weights are assumed to be nonnegative, the paths returned by Dijkstra’s algorithm will always be “simple”. That is, they will never contain the same vertex more than once.

As mentioned earlier, Dijkstra’s algorithm is also exact, meaning that it is guaranteed to determine the shortest  $s$ - $u$ -path for all reachable vertices  $u \in V$ . Short proofs of this correctness can be found in several well-known textbooks such as [9] and [17].

## 4 Using Priority Queues

Although the complexity of Dijkstra’s algorithm is  $O(n^2)$ , for sparse graphs its run times can be significantly improved by making use of a priority queue. During execution, this priority queue is used to hold the labels of all vertices that have been considered by the algorithm but that are not yet marked as distinguished. It should also allow us to quickly identify the undistinguished vertex that has the minimum label value. This “improved” version of Dijkstra’s algorithm is expressed in Algorithm 3.

As shown, the DIJKSTRA procedure in Algorithm 3 uses four data structures,  $D$ ,  $L$ ,  $P$  and  $Q$ . The first three of these contain  $n$  elements and should allow direct access (e.g., by using arrays).  $D$  is used to mark the distinguished vertices, while  $L$  and  $P$  hold the labels and predecessors of each vertex as before. In this pseudocode, the priority queue is denoted by  $Q$ . At each iteration,  $Q$  is used to identify the element  $(L(u), u)$ , representing the undistinguished

---

**Algorithm 3: DIJKSTRA**

---

**input :** An arc weighted graph  $G = (V, A)$ , and source vertex  $s \in V$   
**output:** A populated label array  $L$  and predecessor array  $P$

- 1 For all  $u \in V$ , set  $L(u) = \infty$ , set  $D(u) = \text{false}$ , and set  $P(u) = \text{NULL}$
- 2 Set  $L(s) = 0$  and insert the ordered pair  $(L(s), s)$  into  $Q$
- 3 **while**  $Q$  is not empty **do**
- 4     Let  $(L(u), u)$  be the element in  $Q$  with the minimum value for  $L(u)$
- 5     Remove the element  $(L(u), u)$  from  $Q$
- 6     Set  $D(u) = \text{true}$
- 7     **foreach**  $v \in \Gamma(u)$  such that  $D(v) = \text{false}$  **do**
- 8         **if**  $L(u) + w(u, v) < L(v)$  **then**
- 9             **if**  $L(v) < \infty$  **then**
- 10                 Decrease the key of  $(L(v), v)$  to  $L(u) + w(u, v)$ . That is, replace the element  $(L(v), v)$  in  $Q$  with the element  $(L(u) + w(u, v), v)$
- 11             **else**
- 12                 Insert the element  $(L(u) + w(u, v), v)$  into  $Q$
- 13             Set  $L(v) = L(u) + w(u, v)$  and set  $P(v) = u$

---

	Identify-Minimum	Remove-Minimum	Insert	Decrease-Key
Self-balancing binary tree	$O(1)$	$O(1)^\dagger$	$O(\lg n)$	$O(\lg n)$
Binary heap	$O(1)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
Fibonacci heap	$O(1)$	$O(\lg n)^\dagger$	$O(1)$	$O(1)^\dagger$

Table 1: Complexities of several operators using self-balancing binary trees, binary heaps, and Fibonacci heaps. Here,  $n$  represents the number of elements in the data structure. All entries are worst-case run times except for those marked by  $^\dagger$ , which are amortised run times. Note that the Decrease-Key operation is not available in the binary heap implementation provided by `std::priority_queue` in C++ [1].

vertex  $u$  with the minimal label value. In the remaining instructions, this element is removed from  $Q$ ,  $u$  is marked as distinguished and, if necessary, adjustments are made to the labels of undistinguished neighbours of  $u$  and the corresponding entries in  $Q$ .

The running time of DIJKSTRA now depends on the data structure used for the priority queue  $Q$ . Our first option here is to use a self-balancing binary tree. These are a class of binary trees that automatically keep their height logarithmic to the number of elements they contain. In C++, implementations of a self-balancing binary tree are provided by the `std::set` container, usually using a red-black tree [2].

A second option for  $Q$  is to use a binary heap. Binary heaps are data structures that take the form of *complete* binary trees. Because of this restriction, unlike self-balancing binary trees they can be implemented using an array. This allows them to be stored in contiguous memory and also means that the parent and children of any node can be determined using arithmetic on array indices, as opposed to pointers. In C++, binary heaps are provided by the `std::priority_queue` container [1].

Our final option for  $Q$  is to use a Fibonacci heap. This data structure operates by maintaining several heap-ordered trees. It also features better amortized running times than the previous two options for some of the operations used by Dijkstra’s algorithm. Note, however, that C++ does not contain a Fibonacci heap in its standard library. Instead, a custom class is required.

Table 1 considers these three alternatives for  $Q$  and uses big O notation to summarise the complexities of operations relevant to Dijkstra’s algorithm. Further information on how these data structures work “under the hood” can be found in [9]. For self-balancing binary trees, observe that the removal of the minimum element  $(L(u), u)$  (Line 5 of DIJKSTRA) takes constant amortised time. At Line 10 of the algorithm, Decrease-Key operations are then carried out by finding and removing the element  $(L(v), v)$  in  $Q$  and inserting the new element  $(L(u) + w(u, v), v)$ . This process has a complexity  $O(\lg n)$ . Similarly, the Insert operation on Line 12 also has a complexity of  $O(\lg n)$ . Using a self-balancing binary tree for  $Q$ , therefore, leads to an overall complexity for DIJKSTRA of  $O(m \lg n)$ .

As shown in Table 1, the operations with binary heaps show similar complexities to self-balancing binary trees, though the removal of the minimum element is now  $O(\lg n)$  as opposed to constant amortised time. Note, however, that the `std::priority_queue` container in C++ does not feature the functionality for performing Decrease-Key operations or for removing arbitrary elements in  $Q$ . The use of this data structure, therefore, requires modifications to the DIJKSTRA procedure. Specifically, at Line 10, instead of replacing the element  $(L(v), v)$  in  $Q$  with  $(L(u) + w(u, v), v)$ , the latter

Variant	Complexity	Comments
Basic form (Section 3)	$O(n^2)$	Minimum label $L(u)$ found using linear search. Optimal bound for dense graphs.
Self-balancing binary tree	$O(m \lg n)$	More efficient than the basic form with sparse graphs.
Binary heap	$O(m \lg n)$	Same complexity as previous. Note that the C++ implementation of binary heaps ( <code>std::priority_queue</code> ) does not allow the removal of arbitrary elements, so the size of the heap is $O(m)$ , leading to a complexity of $O(m \lg m)$ .
Fibonacci heap	$O(m + n \lg n)$	Lower complexity than the previous variants.

Table 2: Complexities of the different variants of Dijkstra’s algorithm considered in this paper. Recall that  $n$  gives the number of vertices in the graph, and  $m$  is the number of arcs.

element is now simply inserted into  $Q$  alongside the former. This means that, unlike previously, a vertex  $v$  can occur in several elements of  $Q$ . Because of this, an additional check is now required between Lines 5 and 6 of DIJKSTRA to determine if the selected vertex  $u$  is already distinguished (that is, if  $D(u) = \text{true}$ ). If this is the case, then the remaining steps in the while-loop should not be considered, and the process should return to Line 3. Note that this modification increases the overall complexity of DIJKSTRA to  $O(m \lg m)$ . On the other hand, binary heaps are usually seen to be faster than self-balancing binary trees because they use contiguous memory, require fewer memory allocations, and therefore tend to feature lower constant factors in their operators. The effects of this trade-off are considered in Section 6.

Finally, Table 1 also shows the complexities of these operations using Fibonacci heaps. For this data structure, Insert and Decrease-Key operations both take place in constant amortised time. In particular, unlike self-balancing binary trees, the Decrease-Key operation does not involve a removal followed by an insertion; instead, it operates by identifying the correct element  $(L(v), v)$  in the heap, and then lowering the first value of the element to  $L(u) + w(u, v)$ , modifying its position in the heap as applicable. The use of a Fibonacci heap for  $Q$ , therefore, leads to an overall complexity for DIJKSTRA of  $O(m + n \lg n)$ . This is better than both of the previous options. Despite this, however, Fibonacci heaps are often considered to be slow in practice due to their larger memory consumption and the high constant factors contained in their operators. Indeed, it is noted by Cormen et al. [9] that:

“the constant factors and programming complexity of Fibonacci heaps makes them less desirable than ordinary binary (of  $k$ -ary) heaps for most applications. Thus Fibonacci heaps are predominantly of theoretical interest.”

This claim will also be investigated further in the next section. A summary of the complexities of these algorithm variants is provided in Table 2.

## 5 An Implementation

In this section, we consider four C++ implementations of Dijkstra’s algorithm. The first three use a self-balancing binary tree (`std::set`), a binary heap (`std::priority_queue`), and a Fibonacci heap, as described in the previous section. The fourth implements the basic form of Dijkstra’s algorithm seen in Section 3.

A complete listing of our code is shown in Appendix A and can be downloaded at [3]. The bespoke `FibonacciHeap` class is defined on Lines 79 to 296. Lines 299 to 448 then give our four implementations of Dijkstra’s algorithm: `dijkstraFibonacci(...)`, `dijkstraTree(...)`, `dijkstraHeap(...)`, and `dijkstraBasic(...)`. The `main()` function on Lines 464 onwards applies these methods to a small toy graph. The output is shown at the end of Appendix A.

The following features in this code should be noted.

- Here, graphs are defined using the custom `Graph` class. Graph objects are directed and are stored using the adjacency list representation for weighted graphs [6]. It is assumed that the vertices are labelled from 0 to  $n - 1$  (where  $n$  is the number of vertices). The algorithm will also work perfectly well on undirected graphs by ensuring that if an edge  $\{u, v\}$  is present in the graph, then both of the arcs  $(u, v)$  and  $(v, u)$  are present in the adjacency list.
- All arc weights in the graph are assumed to be nonnegative integers. The code uses the inbuilt C++ constant `INT_MAX` to represent infinity values. Exceeding this value will result in overflow and incorrect behaviour.
- Each version of Dijkstra’s algorithm returns two arrays (vectors): the label vector  $L$  and the predecessor vector  $P$ .

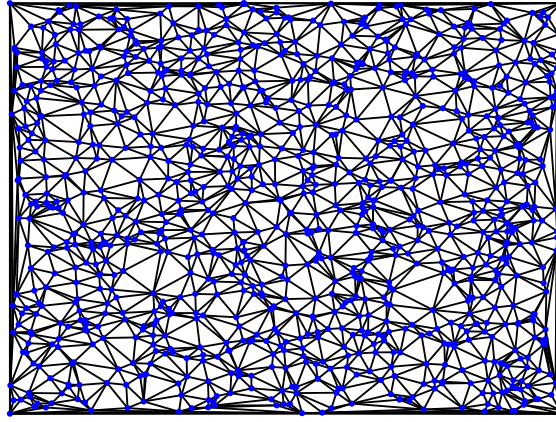


Figure 3: Example of the dense planar graphs used in our trials. This particular instance has  $n = 1000$  vertices and  $m = 5984$  arcs.

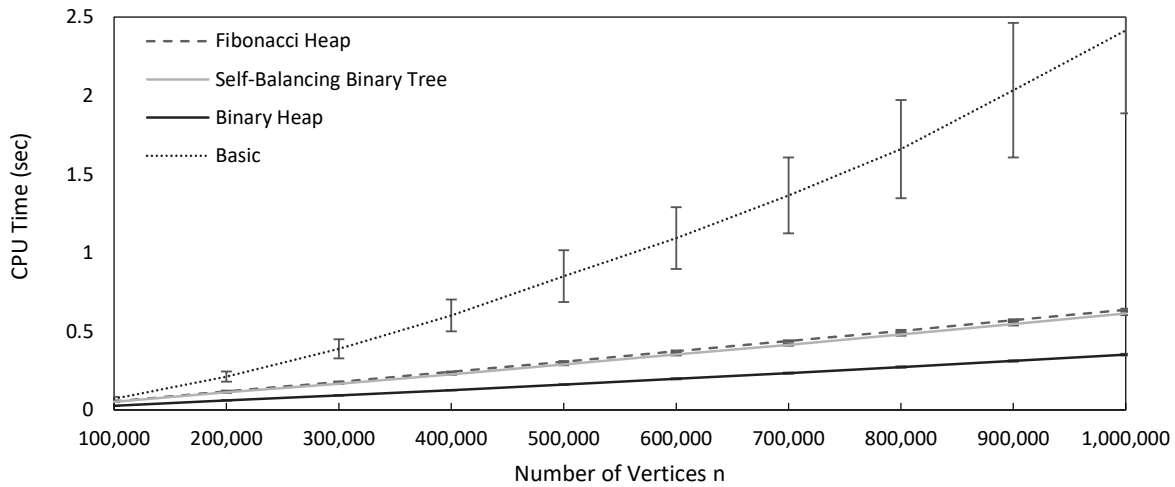


Figure 4: Execution times of the four algorithm variants with dense planar graphs. Each point on the graph is the mean taken from 100 runs. Error bars show one standard deviation on either side of the mean.

- Lines 450 to 462 of the code also give a function for extracting a path from  $P$ . This corresponds to Algorithm 2 above.

In the following, all trials were performed on a 64-bit Windows 10 machine with a 3.3 GHz Pro Intel Core i5-4590 CPU and 8 GB of RAM. In our case, the code was compiled using Microsoft Visual Studio 2019 under release mode.

## 6 Empirical Evaluation

To assess the performance of the four variants of Dijkstra’s algorithm, timed tests were carried out on two graph topologies: dense planar graphs and random graphs. Planar graphs are a type of graph that can be drawn on a plane so that no arcs intersect. Here, they were formed by randomly placing  $n$  vertices into a  $(10,000 \times 10,000)$ -unit square before generating a random Delaunay triangulation to give a graph with approximately (but not exceeding)  $6n - 12$  arcs. The weight of each arc was then set to the Euclidean distance between its two endpoints, giving  $w(u, v) = w(v, u)$  for all  $(u, v) \in A$ . An example is shown in Figure 3. These planar graphs can be considered similar to road networks which, as noted, are an important application area of shortest path algorithms. Our random graphs, meanwhile, were generated by creating  $n$  vertices and then, for each ordered pair of vertices  $u, v$ , adding the arc  $(u, v)$  with a probability  $p$ . This generation process leads to graphs with approximately  $pn(n - 1)$  arcs. Here, each arc was assigned a weight between 1 and 10,000, selected at random.

Execution times of the four algorithm variants with our dense planar graphs are summarised in Figure 4 using a range of values for  $n$ . Across this range, we see that the use of a self-balancing binary tree gives faster run times than Fibonacci heaps, though these differences are very marginal. On the other hand, the use of a binary heap gives a noticeable improvement over these two variants, with run times dropping by approximately half. That said, each

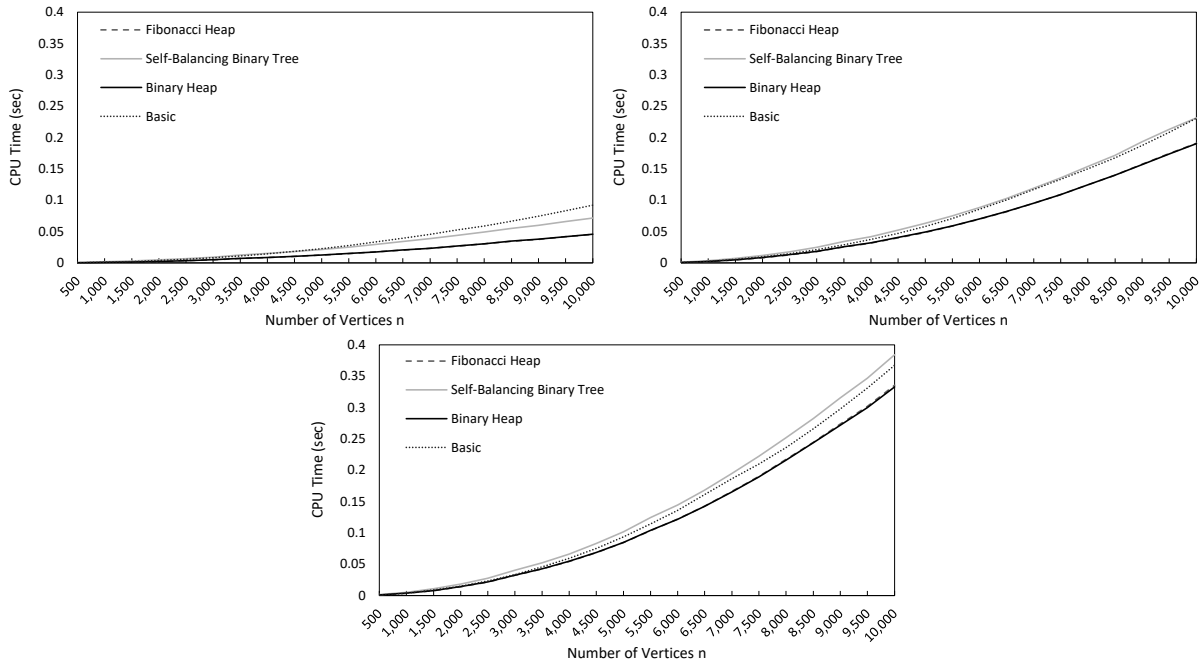


Figure 5: Execution times of the four algorithm variants with random graphs using  $p = 0.1, 0.5$  and  $0.9$  respectively. Each point on the graph is the mean taken from 100 runs. Error bars are not shown here because all standard deviations were seen to be less than 0.006 seconds.

of these three algorithms can compute shortest path trees of up to a million vertices in well under a second with these graphs. As noted, the maximum number of arcs in a directed planar graph is  $6n - 12$  meaning that, in these cases,  $m = O(n)$ . Consequently, we can consider these three variants of Dijkstra’s algorithm to have a complexity of  $O(n \lg n)$  here. Given that binary heaps involve fewer computational overheads than self-balancing binary trees and Fibonacci heaps, this helps to explain the superior performance of the binary heap variant in these cases. Finally, observe that the basic form of Dijkstra’s algorithm shows both higher means and variances in run times compared to the other three options. This is to be expected due to the sparsity of these graphs. The gap between the basic version and the other three also widens with increases in  $n$ , highlighting the quadratic growth rate of the former.

The charts in Figure 5 show the results of the same experiments using random graphs with  $p = 0.1, 0.5$  and  $0.9$  respectively. Note that, in these cases, the number of arcs is much higher than the number of vertices; consequently, the value of  $m$  is now the dominant factor in the variants using priority queues. Since  $m \approx pn(n - 1)$ , increases to  $p$  and/or  $n$  therefore result in longer run times.

For these graphs, we see that the relative performance of the Fibonacci heap variant improves, with its results being almost indistinguishable from those of the binary heap. The reasons for this are that, with these denser graphs, the number of neighbours per vertex is larger. This brings a higher number of Decrease-Key and Insert operations during execution, which are more efficient with Fibonacci heaps. Despite this, however, the additional overheads required by Fibonacci heaps seem to prevent the algorithm from improving on the binary heap’s run times. In cases where  $n$  and/or  $p$  are high, the basic version of Dijkstra’s algorithm also sometimes outperforms the variant using self-balancing binary trees. This is particularly the case for graphs with  $p = 0.9$  where the number of arcs  $m$  in these cases is close to  $n^2$ .

Finally, note that the largest instances considered here involve  $n = 10,000$  vertices, density  $p = 0.9$ , and therefore approximately 90 million arcs. In our runs, such graphs were seen to occupy around 900 MB of memory, but the run times of the three variants using priority queues were still well below half a second in all cases. The times taken to load these graphs into RAM are not included in the above timings, however.

## 7 Conclusions

This paper has described the shortest path problem and shown how the performance of Dijkstra’s algorithm can be affected by the choice of data structure used for its priority queue. Using a C++ implementation tested over a large range of problem instances, we have seen that the best-performing algorithm does not always have the lowest complexity. Indeed, on the whole, the best performance has been seen when using Dijkstra’s algorithm with a binary heap, even though its complexity of  $O(m \lg m)$  is higher than the other variants. For dense graphs, however, the variant using a Fibonacci heap features very similar run times to the binary heap version. Interestingly, this binary-heap variant is also

the chosen method of implementation in several open-source libraries including NetworkX [5] and GraphHopper [4].

As noted, our current implementation operates by loading the entire graph into RAM before execution. It also assumes that vertices are labelled with indices from 0 to  $n - 1$ . In cases where these conditions are not possible, our code will need to be modified to make use of associative arrays instead of vectors. In C++ these are provided by the `std::map` and `std::unordered_map` containers.

## References

- [1] C++ priority queue documentation. [https://en.cppreference.com/w/cpp/container/priority\\_queue](https://en.cppreference.com/w/cpp/container/priority_queue). Accessed 2023-03-15.
- [2] C++ set documentation. <https://en.cppreference.com/w/cpp/container/set>. Accessed 2023-03-15.
- [3] C++ source code and results datasets. <https://doi.org/10.5281/zenodo.7741249>. Accessed 2023-03-15.
- [4] Graphhopper implementation of Dijkstra's algorithm (in java). <https://github.com/graphhopper/graphhopper/blob/master/core/src/main/java/com/graphhopper/routing/Dijkstra.java>. Accessed 2023-03-15.
- [5] Networkx implementation of Dijkstra's algorithm (in python). [https://networkx.org/documentation/stable/\\_modules/networkx/algorithms/shortest\\_paths/weighted.html#single\\_source\\_dijkstra](https://networkx.org/documentation/stable/_modules/networkx/algorithms/shortest_paths/weighted.html#single_source_dijkstra). Accessed 2023-03-15.
- [6] Weighted graph representation. <https://www.tutorialspoint.com/weighted-graph-representation-in-data-structure>. Accessed 2023-03-15.
- [7] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. Werneck. *Route Planning in Transportation Networks*, pages 19–80. Springer International Publishing, Cham, 2016.
- [8] R. Bhandari. *Survivable Networks: Algorithms for Diverse Routing*. Kluwer Academic Publishers, 1999.
- [9] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1st edition, 2000.
- [10] L. Fu, D. Sun, and L. Rilett. Heuristic shortest path algorithms for transportation applications: State of the art. *Computers and Operations Research*, 33(11):3324–3343, 2006.
- [11] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.
- [12] P. Hart, N. Nilsson, and B. Raphael. Correction to 'A formal basis for the heuristic determination of minimum cost paths'. *ACM SIGART Bulletin*, 37:28–29, 1972.
- [13] D. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.
- [14] R. Lewis. Algorithms for finding shortest paths in networks with vertex transfer penalties. *Algorithms*, 13(11), 2020.
- [15] R. Lewis. Who is the centre of the movie universe? Using python and networkx to analyse the social network of movie stars. *arXiv*, 2002.11103, 2020. <https://arxiv.org/pdf/2002.11103.pdf>.
- [16] F. Moore. The shortest path through a maze. Technical report, Bell Telephone System, 1959. vol. 3523.
- [17] K. Rosen. *Discrete Mathematics and its Applications*. Mcgraw Hill, 8th edition, 2018.
- [18] R. Sedgewick. *Algorithms in Java, Part 5: Graph Algorithms*. AddisonWesley Professional, 3rd edition, 2003.
- [19] J. Yen. Finding the  $k$  shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

## A Code Listing and Example Run

The following code, together with this paper's experimental data can be downloaded at [3]

```
1 #include <iostream>
2 #include <climits>
3 #include <algorithm>
4 #include <vector>
5 #include <tuple>
6 #include <set>
7 #include <queue>
8 #include <time.h>
9
10 using namespace std;
11
12 const int infity = INT_MAX;
13
14 //Code for printing a vector
15 template<typename T>
16 ostream& operator<<(ostream& s, vector<T> t) {
17     s << "[";
18     for (size_t i = 0; i < t.size(); i++) {
19         s << t[i] << (i == t.size() - 1 ? "" : ",");
20     }
21     return s << "]" ";
22 }
23
24 //Struct used for each element of the adjacency list.
25 struct Neighbour {
26     int vertex;
```



```

27     int weight;
28 };
29 //Graph class (uses adjacency list)
30 class Graph {
31 public:
32     int n; //Num. vertices
33     int m; //Num. arcs
34     vector<vector<Neighbour> > adj;
35     Graph(int n) {
36         this->n = n;
37         this->m = 0;
38         this->adj.resize(n, vector<Neighbour>());
39     }
40     ~Graph() {
41         this->n = 0;
42         this->m = 0;
43         this->adj.clear();
44     }
45     void addArc(int u, int v, int w) {
46         this->adj[u].push_back(Neighbour{ v, w });
47         this->m++;
48     }
49 };
50
51 //Struct and comparison operators used with std::set std::priority_queue)
52 struct QueueItem {
53     int label;
54     int vertex;
55 };
56 struct minQueueItem {
57     bool operator() (const QueueItem& lhs, const QueueItem& rhs) const {
58         return tie(lhs.label, lhs.vertex) < tie(rhs.label, rhs.vertex);
59     }
60 };
61 struct maxQueueItem {
62     bool operator() (const QueueItem& lhs, const QueueItem& rhs) const {
63         return tie(lhs.label, lhs.vertex) > tie(rhs.label, rhs.vertex);
64     }
65 };
66
67 //Struct used for each Fibonacci heap node
68 struct FibonacciNode {
69     int degree;
70     FibonacciNode* parent;
71     FibonacciNode* child;
72     FibonacciNode* left;
73     FibonacciNode* right;
74     bool mark;
75     int key;
76     int nodeIndex;
77 };
78 //Fibonacci heap class
79 class FibonacciHeap {
80 private:
81     FibonacciNode* minNode;
82     int numNodes;
83     vector<FibonacciNode*> degTable;
84     vector<FibonacciNode*> nodePtrs;
85 public:
86     FibonacciHeap(int n) {
87         //Constructor function
88         this->numNodes = 0;
89         this->minNode = NULL;
90         this->degTable = {};
91         this->nodePtrs.resize(n);
92     }
93     ~FibonacciHeap() {
94         //Destructor function
95         this->numNodes = 0;
96         this->minNode = NULL;
97         this->degTable.clear();
98         this->nodePtrs.clear();
99     }
100     int size() {
101         //Number of nodes in the heap
102         return this->numNodes;

```

```

103     }
104     bool empty() {
105         //Is the heap empty?
106         if (this->numNodes > 0) return false;
107         else return true;
108     }
109     void insert(int u, int key) {
110         //Insert the vertex u with the specified key (value for L(u)) into the Fibonacci
111         //heap. O(1) operation
112         this->nodePtrs[u] = new FibonacciNode;
113         this->nodePtrs[u]->nodeIndex = u;
114         FibonacciNode* node = this->nodePtrs[u];
115         node->key = key;
116         node->degree = 0;
117         node->parent = NULL;
118         node->child = NULL;
119         node->left = node;
120         node->right = node;
121         node->mark = false;
122         FibonacciNode* minN = this->minNode;
123         if (minN != NULL) {
124             FibonacciNode* minLeft = minN->left;
125             minN->left = node;
126             node->right = minN;
127             node->left = minLeft;
128             minLeft->right = node;
129         }
130         if (minN == NULL || minN->key > node->key) {
131             this->minNode = node;
132         }
133         this->numNodes++;
134     }
135     FibonacciNode* extractMin() {
136         //Extract the node with the minimum key from the heap. O(log n) operation, where n
137         //is the number of nodes in the heap
138         FibonacciNode* minN = this->minNode;
139         if (minN != NULL) {
140             int deg = minN->degree;
141             FibonacciNode* currChild = minN->child;
142             FibonacciNode* remChild;
143             for (int i = 0; i < deg; i++) {
144                 remChild = currChild;
145                 currChild = currChild->right;
146                 _existingToRoot(remChild);
147             }
148             _removeNodeFromRoot(minN);
149             this->numNodes--;
150             if (this->numNodes == 0) {
151                 this->minNode = NULL;
152             }
153             else {
154                 this->minNode = minN->right;
155                 FibonacciNode* minNLeft = minN->left;
156                 this->minNode->left = minNLeft;
157                 minNLeft->right = this->minNode;
158                 _consolidate();
159             }
160         }
161         return minN;
162     }
163     void decreaseKey(int u, int newKey) {
164         //Decrease the key of the node in the Fibonacci heap that has index u. O(1)
165         //operation
166         FibonacciNode* node = this->nodePtrs[u];
167         if (newKey > node->key) return;
168         node->key = newKey;
169         if (node->parent != NULL) {
170             if (node->key < node->parent->key) {
171                 FibonacciNode* parentNode = node->parent;
172                 _cut(node);
173                 _cascadingCut(parentNode);
174             }
175         }
176         if (node->key < this->minNode->key) {
177             this->minNode = node;
178         }

```

```

176 }
177 private:
178 //The following are private functions used by the public methods above
179 void _existingToRoot(FibonacciNode* newNode) {
180     FibonacciNode* minN = this->minNode;
181     newNode->parent = NULL;
182     newNode->mark = false;
183     if (minN != NULL) {
184         FibonacciNode* minLeft = minN->left;
185         minN->left = newNode;
186         newNode->right = minN;
187         newNode->left = minLeft;
188         minLeft->right = newNode;
189         if (minN->key > newNode->key) {
190             this->minNode = newNode;
191         }
192     }
193     else {
194         this->minNode = newNode;
195         newNode->right = newNode;
196         newNode->left = newNode;
197     }
198 }
199 void _removeNodeFromRoot(FibonacciNode* node) {
200     if (node->right != node) {
201         node->right->left = node->left;
202         node->left->right = node->right;
203     }
204     if (node->parent != NULL) {
205         if (node->parent->degree == 1) {
206             node->parent->child = NULL;
207         }
208         else {
209             node->parent->child = node->right;
210         }
211         node->parent->degree--;
212     }
213 }
214 void _cut(FibonacciNode* node) {
215     _removeNodeFromRoot(node);
216     _existingToRoot(node);
217 }
218 void _addChild(FibonacciNode* parentNode, FibonacciNode* newChildNode) {
219     if (parentNode->degree == 0) {
220         parentNode->child = newChildNode;
221         newChildNode->right = newChildNode;
222         newChildNode->left = newChildNode;
223         newChildNode->parent = parentNode;
224     }
225     else {
226         FibonacciNode* child1 = parentNode->child;
227         FibonacciNode* child1Left = child1->left;
228         child1->left = newChildNode;
229         newChildNode->right = child1;
230         newChildNode->left = child1Left;
231         child1Left->right = newChildNode;
232     }
233     newChildNode->parent = parentNode;
234     parentNode->degree++;
235 }
236 void _cascadingCut(FibonacciNode* node) {
237     FibonacciNode* parentNode = node->parent;
238     if (parentNode != NULL) {
239         if (node->mark == false) {
240             node->mark = true;
241         }
242         else {
243             _cut(node);
244             _cascadingCut(parentNode);
245         }
246     }
247 }
248 void _link(FibonacciNode* highNode, FibonacciNode* lowNode) {
249     _removeNodeFromRoot(highNode);
250     _addChild(lowNode, highNode);
251     highNode->mark = false;

```

```

252 }
253 void _consolidate() {
254     int deg, rootCnt = 0;
255     if (this->numNodes > 1) {
256         this->degTable.clear();
257         FibonacciNode* currNode = this->minNode;
258         FibonacciNode* currDeg, * currConsolNode;
259         FibonacciNode* temp = this->minNode, * itNode = this->minNode;
260         do {
261             rootCnt++;
262             itNode = itNode->right;
263         } while (itNode != temp);
264         for (int cnt = 0; cnt < rootCnt; cnt++) {
265             currConsolNode = currNode;
266             currNode = currNode->right;
267             deg = currConsolNode->degree;
268             while (true) {
269                 while (deg >= int(this->degTable.size())) {
270                     this->degTable.push_back(NULL);
271                 }
272                 if (this->degTable[deg] == NULL) {
273                     this->degTable[deg] = currConsolNode;
274                     break;
275                 }
276                 else {
277                     currDeg = this->degTable[deg];
278                     if (currConsolNode->key > currDeg->key) {
279                         swap(currConsolNode, currDeg);
280                     }
281                     if (currDeg == currConsolNode) break;
282                     _link(currDeg, currConsolNode);
283                     this->degTable[deg] = NULL;
284                     deg++;
285                 }
286             }
287         }
288         this->minNode = NULL;
289         for (size_t i = 0; i < this->degTable.size(); i++) {
290             if (this->degTable[i] != NULL) {
291                 _existingToRoot(this->degTable[i]);
292             }
293         }
294     }
295 };
296 //End of FibonacciHeap class
297
298 tuple<vector<int>, vector<int>> dijkstraFibonacci(Graph& G, int s) {
299     //Dijkstra's algorithm using a Fibonacci heap object
300     int u, v, w;
301     FibonacciHeap Q(G.n);
302     vector<int> L(G.n), P(G.n);
303     vector<bool> D(G.n);
304     for (int u = 0; u < G.n; u++) {
305         D[u] = false;
306         L[u] = infity;
307         P[u] = -1;
308     }
309     L[s] = 0;
310     Q.insert(s, 0);
311     while (!Q.empty()) {
312         u = Q.extractMin()->nodeIndex;
313         D[u] = true;
314         for (auto& neighbour : G.adj[u]) {
315             v = neighbour.vertex;
316             w = neighbour.weight;
317             if (D[v] == false) {
318                 if (L[u] + w < L[v]) {
319                     if (L[v] == infity) {
320                         Q.insert(v, L[u] + w);
321                     }
322                     else {
323                         Q.decreaseKey(v, L[u] + w);
324                     }
325                 }
326                 L[v] = L[u] + w;
327                 P[v] = u;

```

```

328     }
329     }
330 }
331 }
332 return make_tuple(L, P);
333 }
334
335 tuple<vector<int>, vector<int>> dijkstraTree(Graph& G, int s) {
336 //Dijkstra's algorithm using a self-balancing binary tree (C++ set)
337 int u, v, w;
338 set<QueueItem, minQueueItem> Q;
339 vector<int> L(G.n), P(G.n);
340 vector<bool> D(G.n);
341 for (u = 0; u < G.n; u++) {
342     D[u] = false;
343     L[u] = inf;
344     P[u] = -1;
345 }
346 L[s] = 0;
347 Q.emplace(QueueItem{ 0, s });
348 while (!Q.empty()) {
349     u = (*Q.begin()).vertex;
350     Q.erase(*Q.begin());
351     D[u] = true;
352     for (auto& neighbour : G.adj[u]) {
353         v = neighbour.vertex;
354         w = neighbour.weight;
355         if (D[v] == false) {
356             if (L[u] + w < L[v]) {
357                 if (L[v] == inf) {
358                     Q.emplace(QueueItem{ L[u] + w, v });
359                 }
360                 else {
361                     Q.erase({ L[v], v });
362                     Q.emplace(QueueItem{ L[u] + w, v });
363                 }
364                 L[v] = L[u] + w;
365                 P[v] = u;
366             }
367         }
368     }
369 }
370 return make_tuple(L, P);
371 }
372
373 tuple<vector<int>, vector<int>> dijkstraHeap(Graph& G, int s) {
374 //Dijkstra's algorithm using a binary heap (C++ priority_queue)
375 int u, v, w;
376 priority_queue<QueueItem, vector<QueueItem>, maxQueueItem> Q;
377 vector<int> L(G.n), P(G.n);
378 vector<bool> D(G.n);
379 for (u = 0; u < G.n; u++) {
380     D[u] = false;
381     L[u] = inf;
382     P[u] = -1;
383 }
384 L[s] = 0;
385 Q.emplace(QueueItem{ 0, s });
386 while (!Q.empty()) {
387     u = Q.top().vertex;
388     Q.pop();
389     if (D[u] != true) {
390         D[u] = true;
391         for (auto& neighbour : G.adj[u]) {
392             v = neighbour.vertex;
393             w = neighbour.weight;
394             if (D[v] == false) {
395                 if (L[u] + w < L[v]) {
396                     Q.emplace(QueueItem{ L[u] + w, v });
397                     L[v] = L[u] + w;
398                     P[v] = u;
399                 }
400             }
401         }
402     }
403 }

```

```

404     return make_tuple(L, P);
405 }
406
407 tuple<vector<int>, vector<int>> dijkstraBasic(Graph& G, int s) {
408     //Basic Dijkstra's algorithm (O(n^2) complexity)
409     int u, v, w, minL;
410     size_t i, uPos;
411     vector<int> L(G.n), P(G.n), Candidates;
412     vector<bool> D(G.n);
413     for (u = 0; u < G.n; u++) {
414         D[u] = false;
415         L[u] = infity;
416         P[u] = -1;
417     }
418     L[s] = 0;
419     Candidates.push_back(s);
420     while (!Candidates.empty()) {
421         uPos = 0;
422         minL = L[Candidates[0]];
423         for (i = 1; i < Candidates.size(); i++) {
424             if (L[Candidates[i]] < minL) {
425                 minL = L[Candidates[i]];
426                 uPos = i;
427             }
428         }
429         u = Candidates[uPos];
430         swap(Candidates[uPos], Candidates.back());
431         Candidates.pop_back();
432         D[u] = true;
433         for (auto& neighbour : G.adj[u]) {
434             v = neighbour.vertex;
435             w = neighbour.weight;
436             if (D[v] == false) {
437                 if (L[u] + w < L[v]) {
438                     if (L[v] == infity) {
439                         Candidates.push_back(v);
440                     }
441                     L[v] = L[u] + w;
442                     P[v] = u;
443                 }
444             }
445         }
446     }
447     return make_tuple(L, P);
448 }
449
450 vector<int> getPath(int u, int v, vector<int>& P) {
451     //Get the u-v-path specified by the predecessor vector P
452     vector<int> path;
453     int x = v;
454     if (P[x] == -1) return path;
455     while (x != u) {
456         path.push_back(x);
457         x = P[x];
458     }
459     path.push_back(u);
460     reverse(path.begin(), path.end());
461     return path;
462 }
463
464 int main() {
465     //Construct a small example graph. (A directed cycle on 5 vertices here. All arcs have
466     //weight 10)
467     Graph G(5);
468     G.addArc(0, 1, 10);
469     G.addArc(1, 2, 10);
470     G.addArc(2, 3, 10);
471     G.addArc(3, 4, 10);
472     G.addArc(4, 0, 10);
473
474     //Set the source vertex and declare some variables
475     int s = 0;
476     vector<int> L, P;
477
478     //Execute Dijkstra's algorithm using a Fibonacci heap
479     clock_t start = clock();

```

```

479 tie(L, P) = dijkstraFibonacci(G, s);
480 double duration1 = ((double)clock() - start) / CLOCKS_PER_SEC;
481
482 //Execute Dijkstra's algorithm using a self-balancing binary tree
483 start = clock();
484 tie(L, P) = dijkstraTree(G, s);
485 double duration2 = ((double)clock() - start) / CLOCKS_PER_SEC;
486
487 //Execute Dijkstra's algorithm using a binary heap
488 start = clock();
489 tie(L, P) = dijkstraHeap(G, s);
490 double duration3 = ((double)clock() - start) / CLOCKS_PER_SEC;
491
492 //Execute basic version of Dijkstra's algorithm
493 start = clock();
494 tie(L, P) = dijkstraBasic(G, s);
495 double duration4 = ((double)clock() - start) / CLOCKS_PER_SEC;
496
497 //Output some information
498 cout << "Input graph has " << G.n << " vertices and " << G.m << " arcs\n";
499 cout << "Dijkstra with Fibonacci heap took " << duration1 << " sec.\n";
500 cout << "Dijkstra with self-balancing tree took " << duration2 << " sec.\n";
501 cout << "Dijkstra with binary heap took " << duration3 << " sec.\n";
502 cout << "Dijkstra (basic form) took " << duration4 << " sec.\n";
503 cout << "Shortest paths from source to each vertex are as follows:\n";
504 for (int u = 0; u < G.n; u++) {
505     cout << "v-" << s << " to v-" << u << ",\t";
506     if (L[u] == infity) {
507         cout << "len = infinity. No path exists\n";
508     }
509     else {
510         cout << "len = " << L[u] << "\t\tpath = " << getPath(s, u, P) << "\n";
511     }
512 }
513 }

```

Running this code produces the following output

```

1 Input graph has 5 vertices and 5 arcs
2 Dijkstra with Fibonacci heap took 1.9e-05 sec.
3 Dijkstra with self-balancing tree took 1.2e-05 sec.
4 Dijkstra with binary heap took 1.1e-05 sec.
5 Dijkstra (basic form) took 1.5e-05 sec.
6 Shortest paths from source to each vertex are as follows:
7 v-0 to v-0, len = 0 path = []
8 v-0 to v-1, len = 10 path = [0,1]
9 v-0 to v-2, len = 20 path = [0,1,2]
10 v-0 to v-3, len = 30 path = [0,1,2,3]
11 v-0 to v-4, len = 40 path = [0,1,2,3,4]

```