Prim’s and Dijkstra’s algorithms: A comparison in implementations

Algorithm Design and Analysis Final Project

Mike McGrath

Abstract

This paper explores two well known and related algorithms; Prim’s algorithm and Dijkstra’s algorithm. The algorithms search graphs to find the minimum spanning tree and the shortest path between two points respectively. The paper looked at several different implementations of both algorithms that used different data structures to extract the minimum vertex as well as looked at several different graphs to see how the shape of the graph affects the algorithms.

Introduction

Dijkstra's and Prim's algorithms are two famous algorithms that are some of the first algorithms computer science majors learn in an undergraduate algorithm course. They solve the shortest path and minimum spanning tree (MST) problems respectively. Prim discovered his algorithm while working for Bell Labs on the problem of how to have the smallest terminal-to-terminal network, specifically a telephone network between all the state capitals[1]. Dijkstra published his algorithm two years later in a short note that described his algorithm with a rediscovery of Prim's [2]. The two problems can be more formally stated as:

> Given a connected, undirected, weighted graph G = (V, E, w) with non-negative weights, find a spanning tree of minimum weight   where the weight of a tree is defined as the sum of all its edge weights.
>
> Given a set of V vertices with a source vertex v and a destination vertex d where s,d ∈ V, and a set of E weighted edges over set V, find the shortest-path between s and d

The algorithms are similar in that they return the set of edge with the least weight. The difference is that Prim's tree must include all vertices as it spans the whole graph, whereas Dijkstra's algorithm returns the edges with the least weight between two points (or as this paper explores a source and all other points).

While the two algorithms are very similar, what is interesting is the number of ways that they can be implemented to decrease their time complexity. The brute force implementation for both gives a running time of $O(V^2)$ where V is the number of vertices. This is because both require a linear search in an array if the edges currently being explored are stored in an array. However, if the edges are stored in a different data structure the algorithms can run faster than $O(V^2)$. This is because the algorithms' time complexity is dominated by the cost of extracting the minimum vertex from the queue or array and the cost of updating the priority of each edge. For Prim's and Dijkstra's the data structure of choice is a binary heap since it reduces the time to extra the minimum vertex from a linear search through an array to a log n search through a heap. The fastest implementation is with a Fibonacci heap that can update the edges in constant time.[3]

The pseudocode for both algorithms is simple and similar[4]. The basic idea is that the initial cost to reach all the vertices except the source is initialized to infinite, the edges from the sources are

[1] Prim, R. C. (November 1957), "Shortest connection networks And some generalizations", *Bell System Technical Journal*, **36** (6): 1389–1401
[2] Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". *Numerische Mathematik*. **1**: 269–271.
[3] Madkour, Amgad & Aref, Walid & Rehman, Faizan & Rahman, Abdur & Basalamah, Saleh. (2017). A Survey of Shortest-Path Algorithms.

[4] Cormen, T. H., & Cormen, T. H. (2001). *Introduction to algorithms*. Cambridge, Mass: MIT Press.

explored with the least weighted edge added to the MST or added to the path to the source, and each new vertex is updated or added to the queue.

```
DIJKSTRA(G, w, s)
  Initialize-Single-Source(G,s)
  S = 0
  Q = G.V
   While Q != 0
        U = Extract-Min(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u]
          Relax(u,v,w)

Prim(G, w, r)
  for each u ∈ G.V
        u.key  = inf
        u.π = NIL
  r.key = 0
  Q = G.V
   while Q != 0
        u  = Extract-Min(Q)
        for each v ∈ in Q.adj[u]
          if v ∈ Q and w(u,v) < v.key
            v.π = u
            v.key = w(u,v)
```

The main difference is that Dijkstra's algorithm updates its answer immediately after extracting the minimum since it is concerned with going from vertex to vertex. Prim's algorithm has to loop all adjacent vertices from the minimum and then check if the minimum distance has an unreached node and if it is smaller than the previous weight to reach the new node. This slight difference accounts for why the two algorithms have the same time complexity of O(V^2) but different run times.

Method

For the experiment, several different implementations of Prim's and Dijkstra's algorithms were used with the pseudocode from the textbook *Introduction to Algorithms*, Wikipedia and Stackoverflow (not the best sources as we shall see). The graphs were constructed with the Networkx library that allowed for quick generation of different graphs. Furthermore, Networkx has built in functions that allows the user to specifically request that Dijkstra's or Prim's algorithm be used. These library functions served as a benchmark to see how this paper's implementation compared. It also allowed a quick way to compare results to check accuracy of the functions.

In addition to Networkx, the other major libraries used were defaultdict, heapq, and heapDict[5]. These three libraries provided the different data structures to hold the edges currently being explored. As the main chokepoint for the algorithms are the data structures and ability to get the minimum weight vertex, these libraries played a large role in how fast the algorithms would run.

Each algorithm was tested against the Networkx library's implementation to confirm that they were correctly implemented. This was done by placing the algorithms in a loop that generated a random graph and then compared the results of each algorithm. If the algorithms had different results an assertion error is thrown. The tests were run with graph sizes of 1 to 30. If any error was found, the algorithms were fixed until they matched the correct results. The main errors found were from the edge case of a graph size of one returning [] when it should be [0] or the paths having different formats such as returning as list of list when a dictionary was returned for the library. These were quickly fixed to match the libraries.

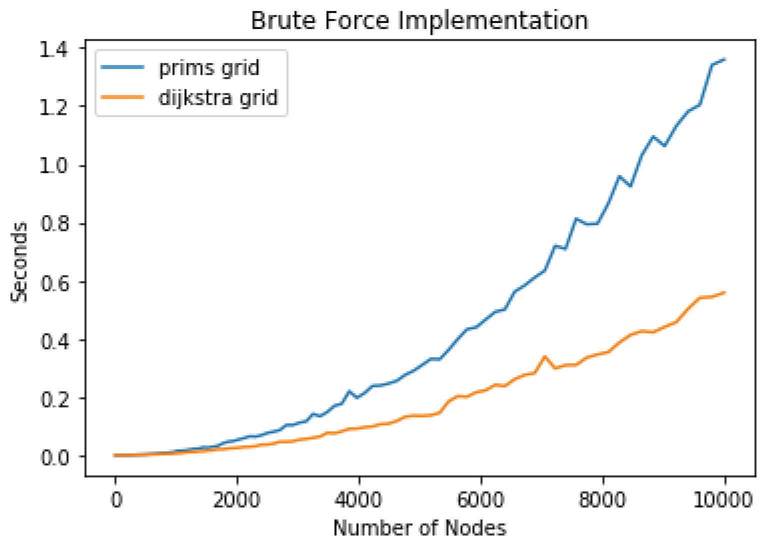
For the data, the algorithms were run in a loop that constructed a new graph with i^2 vertices where i is the iteration of the loop. The loops were run 100 times with a maximum number of vertices of 10,000. Sometimes to confirm results larger iterations were used. In each loop, the algorithms were timed to see how long it took to complete. This was done 10 times with the results then averaged to control for noise. The results were stored in an array and then graphed. Finally, the results were visually inspected to make sure no hardware interruptions caused the algorithms to slow down, i.e. the user running other programs while the algorithm was running.

For the second experiment, a similar loop and averaging was used but with an Erdős-Rényi graph with 100 vertices. This is a graph that randomly assigns edges to nodes based on a probability. The probability ranged from .01 to 1 which is approximately 50 edges to 4950 edges except with the simple implementations of the algorithms since they could not handle unconnected vertices. In that case, the probability was .5 to 1. When the probability is 1, it is a fully connected graph.

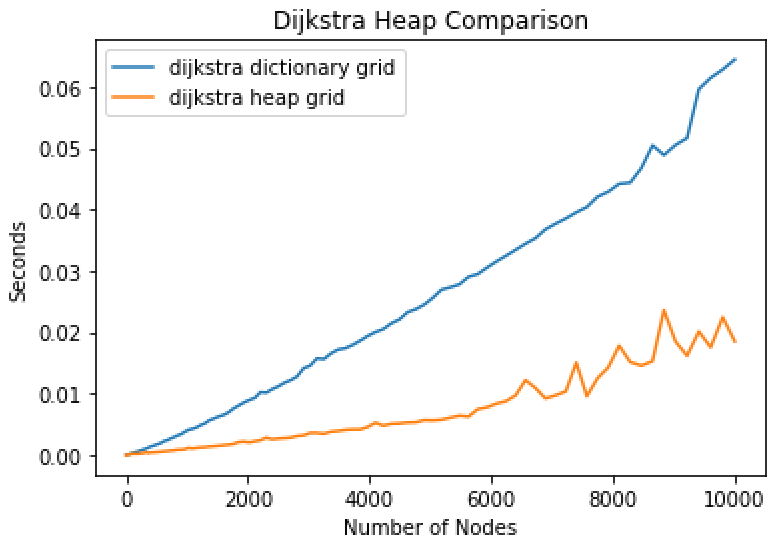
Results

The first implementations of Prim's and Dijkstra's use an array to store the edges currently being explored. Since it takes linear time to search the array for the minimum weighted edge the algorithms should take $O(V^2)$ time. This is clearly visible in the results with the below figure.

[5] https://pypi.org/project/HeapDict/

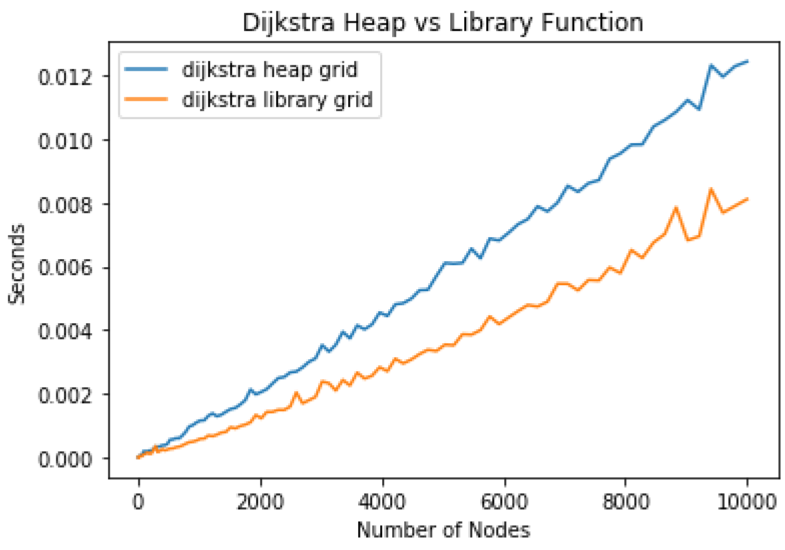


To reduce the time complexity of the two algorithms, an attempt was made to use a binary heap. Because Python's normal library for a priority queue, heapq, does not have a native way of changing the priority of value in the queue, the heapDict library was used instead. While the results did improve the time complexity from $O(V^2)$ it does not achieve the desired heap implementation of complexity of $O((E + V) \log V)$[6].

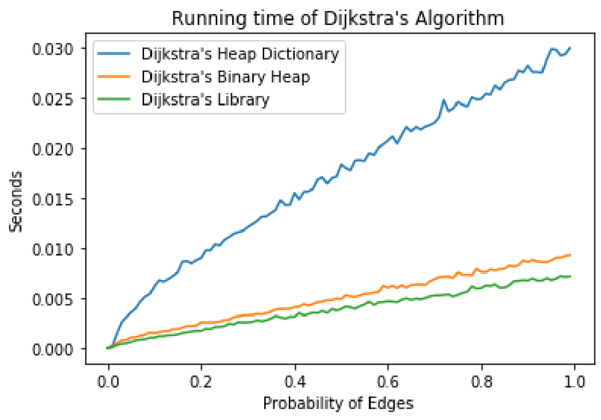


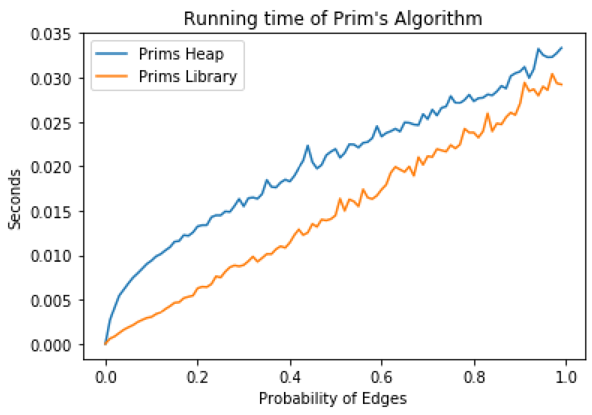
The final implementation of Dijkstra's uses a binary heap to implement the priority queue and keeps the old values in the queue without. This causes a slight slow down as it has to discard the duplicate edge values instead updating the priority, but with basic Python code it is as close to the heap implementation that the Networkx library uses that the algorithm gets.

[6]Cormen, T. H., & Cormen, T. H. (2001). *Introduction to algorithms*. Cambridge, Mass: MIT Press.

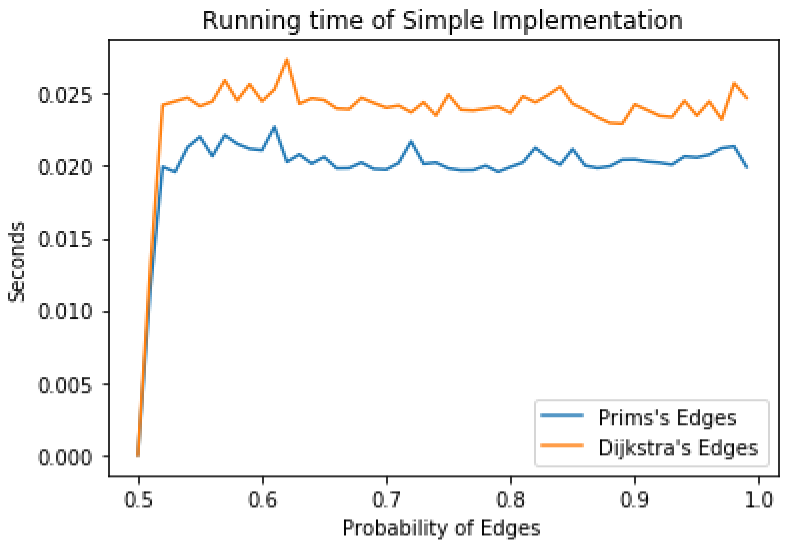


The other main experiment was to see how the number of edges effect the running time of the algorithms.





Above are the graphes for Dijkstra's heap implementations on the left and Prim's heap implementations on the right. Notice that even the library's implementation of Prim's algorithm runs slower compared to Dijkstra's. The simple implementations of the algorithm's required a change in parameters to make the algorithms work (discussed below).



Discussion

The experiment revealed how important data structures are to implementing Dijkstra's and Prim's algorithms. Since both rely on inserting and removing the least weighted edge from a data structure, it is important to have a structure that works well. With the simple implementation that uses an array, the algorithms run in $O(V^2)$. As a side note, while playing around with different implementations found on GitHub and Stackoverflow, several of the Prim's algorithms did not actually run in $O(V^2)$ time and had worst run times because of poor implementations. Often this was because the algorithms were not removing vertices and therefore checking edges repeatedly.

One of the first problems that was encountered was how to update the weights of a vertex. This is easy when the vertices are stored in an array, but when they are stored in a heap, updating the heap requires finding the current entry, changing it, and then reheaping it. Making the task harder, Python does not have a decrease-priority command for its heapq library. The initial solution was to use a heap dictionary which combines a heap and a dictionary to make updating the priority easier. This provided a solution that was faster, but it was not as fast as another method that left the old values on the heap and removes them from the inner loop with a conditional check.

For the second experiment, we see that as the number of edges grows the algorithm grows either logarithmically or not at all. With the simple implementations that are $O(V^2)$, the fact that the number of vertices are the same for each graph dominates the addition of edges; it does not matter how many edges there are since the extraction time for a new vertex is so slow. For Dijkstra's algorithm, the binary heap is very close to the library's implementation. It goes from roughly 50 to 4950 edges with barely increasing in time. The dictionary heap is noticeably slower, but this is most likely caused by the algorithm updating all edges without checking to see if the vertex has already been seen. The binary heap does this which limits the number of edges it needs to look at and could explain why it grows slower with more edges.

Another important detail to point out with the second experiment was that Erdős-Rényi graphs are not connected. For the first several iterations of the loops that collect data for experiment two, the graphs do not have all the vertices connected. In general, the graph could have a disconnect vertex until the probability reaches .5. This is why the simple simple implementations of the algorithms have different graphes. They had to start at edge probability of .5 since the start of the algorithm places all vertices into the array that is search. If a vertex is not reachable the program crashes.

Further Research

The next logical step is to implement a fibonacci heap to see how that affects the algorithms. The fibonacci heap allows for O(1) decrease-priority which save time over the current implementations that are either O(log n) or have to remove old values of priorities with a conditional. Since the Networkx library does not use a fibonacci heap, it would be possible to have results faster than the benchmark.

Besides changing the data structure, the other main variable is the type of graph. The algorithms were tested with three different types of graphs: a grid graph, a Erdős-Rényi graph, and a complete graph (the complete graph did not make the write up of the paper). Other possible graphes that could be studied would be multi-dimensional graphs, hypercube graphs, and n-lattice graphs. More advanced options would be graphs that have unique clusterings and weight distributions.

Conclusion

Implementing correct algorithms is not too hard. Implementing optimized algorithms is hard. There are lots of details to get right and often pseudocode that looks easy can be difficult to get right. For example, a quick overview of implementing fibonacci heaps finds that it is not an easy data structure to implement nor does it offer a great improvement. Overall, the in depth review of these two classic algorithms provided a key insight in basic algorithm design and analysis.