Project 1: Dijkstra’s with Fib Heap

CSE 591 Foundations of Algorithms

Tim Abbott & Greg Moon

2016

Contents

[Algorithm A: 2](#_Toc443049544)

[Running time of A: 2](#_Toc443049545)

[Proving Algorithm A: 3](#_Toc443049546)

[Algorithm A Proof: 3](#_Toc443049547)

[Dijkstra’s Shortest Path Algorithm Proof: 3](#_Toc443049548)

[Fibonacci Heap Data Structure Proof: 5](#_Toc443049549)

[Why C++ 5](#_Toc443049550)

[Sample Output 6](#_Toc443049551)

[Sample 1 6](#_Toc443049552)

[Sample 2 7](#_Toc443049553)

[Sample 3 8](#_Toc443049554)

[References 9](#_Toc443049555)

# Algorithm A:

1. Insert the Graph (nodes and the mandatory edges – no optional edges) into a Fibonacci Heap.
2. Run Dijkstra Algorithm on the heap with S as the start point in order to mark the distance of all nodes from S.
3. Run Dijkstra Algorithm on the heap with T as the start point in order to mark the distance of all nodes from T.
4. Determine the length of the shortest path by looking at T’s distance from S (from step 2) OR by looking at S’s distance from T (from step 3). Let this length also represent the length of the current shortest path.
5. For Each optional edge
   1. Calculate the distance of the shortest road through that edge by summing
      1. Distance from the start node to the origin node of the optional edge.
      2. Length of the optional edge
      3. Distance from the target node to the destination node of the optional edge.
   2. If the distance is less than the shortest path of the original road (from step 4)
      1. Then add edge to the validEdges list because it generates a shorter path
   3. If the distance is less than the current shortest path (initially calculated in step 4)
      1. Then update the current shortest path to this new distance
      2. Then set pointer to this new shortest path
   4. If the shortest path pointer is null
      1. Print out "There is no optional road that generates a shorter path."
   5. Else
      1. Print out all edges in the validEdges list
      2. Generate the shortestPath list
         1. Create a list that points from the origin node to the destination node of the optimal edge.
         2. Then working with the origin node, traverse through all back (previous) pointers generated in step 2 and add these nodes to the head of shortestPath.
         3. Then working with the destination node, traverse through all back (previous) pointers generated in step 3 and add these nodes to the tail of shortestPath.
      3. Print the shortestPath

## Running time of A:

# Proving Algorithm A:

## Algorithm A Proof:

Assume that we want to find the shortest path of a graph between an arbitrary starting node S and an arbitrary target node T. Adding a single unidirectional edge E to the graph from any node X to any node Y may alter the shortest distance from S to Y (and by extension to T) but it will not alter the shortest distance from S to X. This same edge may alter the distance from T to X (and by extension S) but it does not alter the distance between T and Y. Thus the shortest paths from S to X and T to Y are fixed regardless of the length of the added edge. Then we can find the shortest length of the path from S to T through E by summing S to X, the length of E, and T to Y.

Nodes S and T can be any arbitrarily fixed nodes and X and Y represent all nodes. We need an algorithm that can find the distance from any S to all X and any T to all Y. Dijkstra's Shortest Path algorithm based on a Fibonacci Heap is guaranteed to find the shortest path between any node and all other nodes in the quickest amount of computational time. Thus we can use Dijkstra's algorithms on S and T to find the distances of S to X and T to Y.

After that we must iterate through all optional edges E to find all that produce a shorter path through the graph.

Algorithm A is the requires Dijkstra’s Algorithms made with a Fibonacci Heap to generate a shortest path tree. These proof is publicly available and is reproduced below for your convenience (from Wikipedia):

## Dijkstra’s Shortest Path Algorithm Proof:

Proof is by induction on the number of visited nodes.

Invariant hypothesis: For each visited node u, dist[u] is the shortest distance from source to u; and for each unvisited v, dist[v]is the shortest distance via visited nodes only from source to v (if such a path exists, otherwise infinity; note we do not assumedist[v] is the actual shortest distance for unvisited nodes).

The base case is when there is just one visited node, namely the initial node source, and the hypothesis is trivial.

Assume the hypothesis for *n-1* visited nodes. Now we choose an edge uv where v has the least dist[v] of any unvisited node and the edge uv is such that dist[v] = dist[u] + length[u,v]. dist[v] must be the shortest distance from source to v because if there were a shorter path, and if w was the first unvisited node on that path then by hypothesis dist[w] > dist[v] creating a contradiction. Similarly if there was a shorter path to v without using unvisited nodes then dist[v] would have been less thandist[u] + length[u,v].

After processing v it will still be true that for each unvisited node w, dist[w] is the shortest distance from source to w using visited nodes only, since if there were a shorter path which doesn't visit v we would have found it previously, and if there is a shorter path using v we update it when processing v .

Bounds of the running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of the number of edges, denoted |E|, and the number of vertices, denoted |V|, using big-O notation. How tight a bound is possible depends on the way the vertex set Q is implemented. In the following, upper bounds can be simplified because |E| = O(|V|^2)for any graph, but that simplification disregards the fact that in some problems, other upper bounds on |E| may hold.

For any implementation of the vertex set Q, the running time is in

,

where T_\mathrm{dk} and T_\mathrm{em} are the complexities of the *decrease-key* and *extract-minimum* operations in Q, respectively. The simplest implementation of the Dijkstra's algorithm stores the vertex set Q as an ordinary linked list or array, and extract-minimum is simply a linear search through all vertices in Q. In this case, the running time is O(|E| + |V|^2) = O(|V|^2).

For sparse graphs, that is, graphs with far fewer than |V|^2 edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a self-balancing binary search tree, binary heap, pairing heap, or Fibonacci heap as a priority queue to implement extracting minimum efficiently. To perform decrease-key steps in a binary heap efficiently, it is necessary to use an auxiliary data structure that maps each vertex to its position in the heap, and to keep this structure up to date as the priority queue Q changes. With a self-balancing binary search tree or binary heap, the algorithm requires

\Theta((|E| + |V|) \log |V|)

time in the worst case; for connected graphs this time bound can be simplified to \Theta( | E | \log | V | ). The Fibonacci heap improves this to

O(|E| + |V| \log|V|). [1]

## Fibonacci Heap Data Structure Proof:

The amortized performance of a Fibonacci heap depends on the degree (number of children) of any tree root being *O*(log *n*), where *n* is the size of the heap. Here we show that the size of the (sub)tree rooted at any node *x* of degree *d* in the heap must have size at least *Fd*+2, where *Fk* is the *k*th Fibonacci number. The degree bound follows from this and the fact (easily proved by induction) that  for all integers , where . (We then have , and taking the log to base  of both sides gives  as required.)

Consider any node *x* somewhere in the heap (*x* need not be the root of one of the main trees). Define **size**(*x*) to be the size of the tree rooted at *x* (the number of descendants of *x*, including *x* itself). We prove by induction on the height of *x* (the length of a longest simple path from *x* to a descendant leaf), that **size**(*x*) ≥ *Fd*+2, where *d* is the degree of *x*.

**Base case:** If *x* has height 0, then *d* = 0, and **size**(*x*) = 1 = *F*2.

**Inductive case:** Suppose *x* has positive height and degree *d*>0. Let *y*1, *y*2, ..., *yd* be the children of *x*, indexed in order of the times they were most recently made children of *x* (*y*1 being the earliest and *yd* the latest), and let *c*1, *c*2, ..., *cd* be their respective degrees. We **claim** that *ci* ≥ *i*-2 for each *i* with 2≤*i*≤*d*: Just before *yi* was made a child of *x*, *y*1,...,*yi*−1 were already children of *x*, and so *x* had degree at least *i*−1 at that time. Since trees are combined only when the degrees of their roots are equal, it must have been that *yi*also had degree at least *i*-1 at the time it became a child of *x*. From that time to the present, *yi* can only have lost at most one child (as guaranteed by the marking process), and so its current degree *ci* is at least *i*−2. This proves the **claim**.

Since the heights of all the *yi* are strictly less than that of *x*, we can apply the inductive hypothesis to them to get **size**(*yi*) ≥ *Fci*+2 ≥ *F*(*i*−2)+2 = *Fi*. The nodes *x* and *y*1 each contribute at least 1 to **size**(*x*), and so we have

\textbf{size}(x) \ge 2 + \sum_{i=2}^d \textbf{size}(y_i) \ge 2 + \sum_{i=2}^d F_i = 1 + \sum_{i=0}^d F_i.

A routine induction proves that 1 + \sum_{i=0}^d F_i = F_{d+2} for any d\ge 0, which gives the desired lower bound on **size**(*x*). [2]

# Why C++

We chose C++ for two reasons. 1) We were familiar with the language, and 2) C++ uses explicit pointers which helped us create the Fibonacci Heap.

# Sample Output

## Sample 1

|  |  |
| --- | --- |
| 4  4  0:2:20  0:1:10  1:3:20  2:3:10  1  1:2:20  0  3 | Reading 4 cities.  Reading city 0.  Reading city 1.  Reading city 2.  Reading city 3.  Done reading cities.  Reading 4 uni-directional roads.  Reading from city 0 to city 2 with length of 20  Reading from city 0 to city 1 with length of 10  Reading from city 1 to city 3 with length of 20  Reading from city 2 to city 3 with length of 10  Done reading existing uni-directional roads.  Reading 1 uni-directional roads.  Reading from city 1 to city 2 with length of 20  Done reading optional uni-directional roads.  Starting city: 0  Ending city: 3  Calculating Shortest Path...  Shortest path using existing roads:  0 -> 1 -> 3 with length 30  PATHS MADE BY OPTIONAL ROADS BELOW =====================  Paths using an optional road:  Path 1: 0 -> 1 -> 2 -> 3 with length 40  SHORTEST PATH SUMMARY BELOW ==============================  **0: No optional paths are shorter than the path using only existing roads.** |

.

## Sample 2

|  |  |
| --- | --- |
| 4  4  0:2:20  2:3:10  1:3:20  0:1:10  1  1:2:5  0  3 | Reading 4 cities.  Reading city 0.  Reading city 1.  Reading city 2.  Reading city 3.  Done reading cities.  Reading 4 uni-directional roads.  Reading from city 0 to city 2 with length of 20  Reading from city 2 to city 3 with length of 10  Reading from city 1 to city 3 with length of 20  Reading from city 0 to city 1 with length of 10  Done reading existing uni-directional roads.  Reading 1 uni-directional roads.  Reading from city 1 to city 2 with length of 5  Done reading optional uni-directional roads.  Starting city: 0  Ending city: 3  Calculating Shortest Path...  Shortest path using existing roads:  0 -> 1 -> 3 with length 30  PATHS MADE BY OPTIONAL ROADS BELOW =====================  Paths using an optional road:  **Path 1: 0 -> 1 -> 2 -> 3 with length 25**  SHORTEST PATH SUMMARY BELOW ==============================  **1: Optional path 1 is shorter than the path using only existing roads.**  **Optional path 1 is the shortest possible path and has length 25.** |

.

## Sample 3

|  |  |
| --- | --- |
| 6  4  0:1:2  4:5:2  2:3:2  3:5:2  2  1:4:1  0:2:2  0  5 | Reading 6 cities.  Reading city 0.  Reading city 1.  Reading city 2.  Reading city 3.  Reading city 4.  Reading city 5.  Done reading cities.  Reading 4 uni-directional roads.  Reading from city 0 to city 1 with length of 2  Reading from city 4 to city 5 with length of 2  Reading from city 2 to city 3 with length of 2  Reading from city 3 to city 5 with length of 2  Done reading existing uni-directional roads.  Reading 2 uni-directional roads.  Reading from city 1 to city 4 with length of 1  Reading from city 0 to city 2 with length of 2  Done reading optional uni-directional roads.  Starting city: 0  Ending city: 5  Shortest path using existing roads:  There is no path from 0 to 5 using only existing roads.  PATHS MADE BY OPTIONAL ROADS BELOW =====================  Paths using an optional road:  **Path 1: 0 -> 1 -> 4 -> 5 with length 5**  Path 2: 0 -> 2 -> 3 -> 5 with length 6  SHORTEST PATH SUMMARY BELOW ==============================  **2+: Optional paths 1 and 2 are shorter than the path using only existing roads.**  **Optional path 1 is the shortest possible path and has length 5.** |

.

# References

|  |  |
| --- | --- |
| [1] | "Dijkstra's algorithm," [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm. [Accessed 12 2 2016]. |
| [2] | "Fibonacci heap," [Online]. Available: https://en.wikipedia.org/wiki/Fibonacci\_heap. [Accessed 12 2 2016]. |