**Finding the Shortest Path from a Vertex to All Other Vertices**

The bfs found the shortest path from the start vertex to all other vertices, assuming that the length or weight of each edge was the same

Dijkstra’s algorithm finds the shortest path in a weighted directed graph

We need 2 sets and 2 arrays

* Set S will contain the vertices for which we have computed the shortest distance
  + Initialize S by placing the starting vertex s into it
* Set V-S will contain the vertices we still need to process
  + Initialize V-S by placing the remaining vertices into it

---------------------------------------------------------------------------------------------------------------------------

* d[v] will contain the shortest distance from s to v
  + Initially, for each v in V-S, set d[v] to the weight of the edge w(s, v) for each vertex v adjacent to s and to fır the other vertices
* p[v] will contain the predecessor of v in the path from s to v
  + Initialize p[v] to s for each v in V-S

For the vertices in S, values in d and p arrays are fixed.

Dijkstra’s Algorithm

Chart

Description automatically generated

For start vertex (0), d[0] = 0 and p[0] = -1

S : start vertex

Diagram

Description automatically generated

Diagram

Description automatically generated

A picture containing text, device

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Chart

Description automatically generated

Chart

Description automatically generated

A picture containing chart

Description automatically generated

p[2] is better to be -1 first but it is going to modified so you can keep it as 0.

At each iteration, we find the smallest distance in d in V-S (adjacent to according vertex) and fix it.

Chart

Description automatically generated

Chart

Description automatically generated with medium confidence

Chart, diagram

Description automatically generated with medium confidence

Chart, diagram

Description automatically generated

Chart, diagram

Description automatically generated

Chart, diagram

Description automatically generated

Diagram

Description automatically generated

Chart

Description automatically generated with medium confidence

Diagram

Description automatically generated

Chart, diagram

Description automatically generated with medium confidence

Chart

Description automatically generated

Chart

Description automatically generated

Chart

Description automatically generated

Chart, diagram

Description automatically generated with medium confidence

Chart

Description automatically generated

Chart

Description automatically generated

Chart

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated with medium confidence

Diagram

Description automatically generated

Algorithm

Text, letter

Description automatically generated

w(s, v) 🡪 weight of edge from s to v

First 6 lines are initialization of the ---> S, V-S, d, p

At the end, S and V-S are not useful but d and p are useful.

n: number of vertices, m: number of edges

If we use TreeSet which uses red-black tree for S and V-S and regular array for d and p:

* LINE 1 🡪 O(nlogn)
  + Initializing S with 1 element is constant time
  + Initializing V-S with the remaining (n-1) takes logn for each and there are n-1 elements
  + …
  + If we have used HashSet, we have amortized O(n) time because adding an element is constant
* LINE 2 🡪 (n)
  + This line iterates through 0 to n-1 which doesn’t include s
* LINE 3 🡪 (n) (line itself is constant but runs n times)
* LINE 4 🡪 (n) (adjacency matrix is used)
  + Running time depends on how the graph is implemented (these 2 are line itself):
    - Adjacency List : (n) -----> line 4 takes O(n2) time
    - Adjacency Matrix : (1) --> line 4 takes (n) time
* LINES 5 & 6 🡪 (n)
* LINE 7 🡪 (n)
  + Iterates n-1 times
* LINE 8 🡪 (n2) (line itself is n, other n comes from while loop)
  + At first we have n-1 elements in V-S set, then n-2 🡪 n-3 🡪 n-4 🡪 … 🡪 1
  + We should use an iterator
  + Iterator in TreeSet requires a inorder traversal 🡪 takes linear time with size of V-S
  + Checking for smallest and updating the smallest takes constant time
* LINE 9 🡪 O(nlogn)
  + Removing from and adding to TreeSet is logarithmic
* LINE 10 🡪 (n2) + O(mlogn) (adjacency matrix is used)
  + Finding vertices adjacent to u depends on graph implementation
  + Also we have to check contains operation in TreeSet 🡪 takes logarithmic time
  + We find the adjacent vertices first, then check whether it is in V-S or not 🡪 Checking whether it is V-S has to be performed 2m times for each edges.
  + So this operation whether it is in V-S or not will take O(mlogn) time additional to n2
  + For adjacency list:
    - We obtain the adjacent edges in constant time for each edge and total of m edges are traversed in the while loop 🡪 (m) + O(mlogn)
* LINE 11 & 12 & 13 🡪 (2m) = (m)
  + d[u] can be obtained in constant time
  + since we found the edge, we can get the weight in constant time
  + checking whether it is less than d[v] can be done in constant time
  + sets can be done in constant time
  + These 3 lines are constant time but they are performed 2m times for all vertices and all the vertices adjacent to it (a to b, b to a 🡪 each edge is processed twice)

OVERALL RUNNING TIME FOR ADJACENCY MATRIX ------> (mlogn + n2)

OVERALL RUNNING TIME FOR ADJACENCY LIST ------> (mlogn + n2)

What if we used HashSet instead of TreeSet?

LINE 1 🡪 Inserting will be constant time and line becomes O(n)

LINE 9 🡪 Becomes constant time because removing is constant

LINE 10 🡪 Checking whether u is in V-S becomes constant time so line becomes (n2) + O(m)

THEN OVERALL RUNNING TIME BECOMES ------> (m + n2)

m comes from lines 11 & 12 & 13. We have to update vertices m times.

n2 comes from line 8. We have to find smallest of the vertices. We can reduce this running time by using the priority queue (finding smallest is easy).

Implementation

Text, letter

Description automatically generated



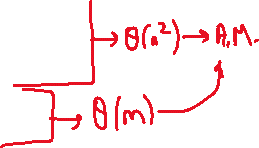
OVERALL FOR A.M. 🡪 (m + n2)

OVERALL FOR A.L. 🡪 O(mn)

For an adjacency list (A.L.) representation, modify the code:

Text, letter

Description automatically generated



Order is different. Here we find the edge first, then check whether it is in V-S or not. This is good for A.L.

In the first version, we iterate through all the vertices in V-S and check whether it is adjacent or not. This is good for A.M.

After you use this implementation and use A.L., only n2 comes from rainbow circle in the previous page (for loop). It can be modified by using priority queue.

By using A.L., priority queue, and HashSet (good for sparse graphs) 🡪 running time can be : O(mlogn)

By using A.M., and HashSet (good for dense graphs) 🡪 running time can be : O(n2)

CHECK GRAPHS SLIDE STARTING FROM 223 FOR PRIM’S ALGORITHM