REPORT

DIJKSTRA'S ALGORITHM

DONE BY: MERIN PUTHUPARAMPIL

TOPICS

1. INTRODUCTION

Dijkstra's algorithm is called the single-source shortest path. It is also known as the single source shortest path problem. It computes length of the shortest path from the source to each of the remaining vertices in the graph.

The single source shortest path problem can be described as follows:

Let G= {V, E} be a directed weighted graph with V having the set of vertices. The special vertex s in V, where s is the source and let for any edge e in E, EdgeCost(e) be the length of edge e. All the weights in the graph should be non-negative.

Before going in depth about Dijkstra’s algorithm let’s talk in detail about directed-weighted graph.

Directed graph can be defined as an ordered pair G: = (V,E) with V is a set, whose elements are called vertices or nodes and E is a set of ordered pairs of vertices, called directed edges, arcs, or arrows. Directed graphs are also known as digraph.

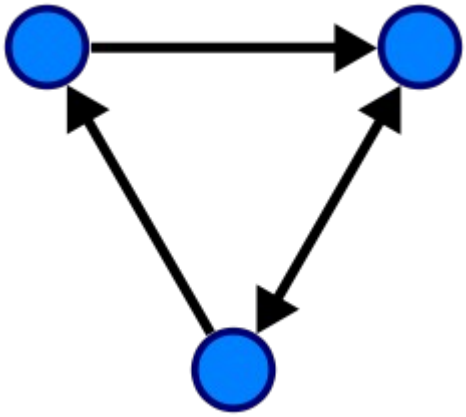


Figure: Directed graph

Directed-weighted graph is a directed graph with weight attached to each of the edge of the graph.

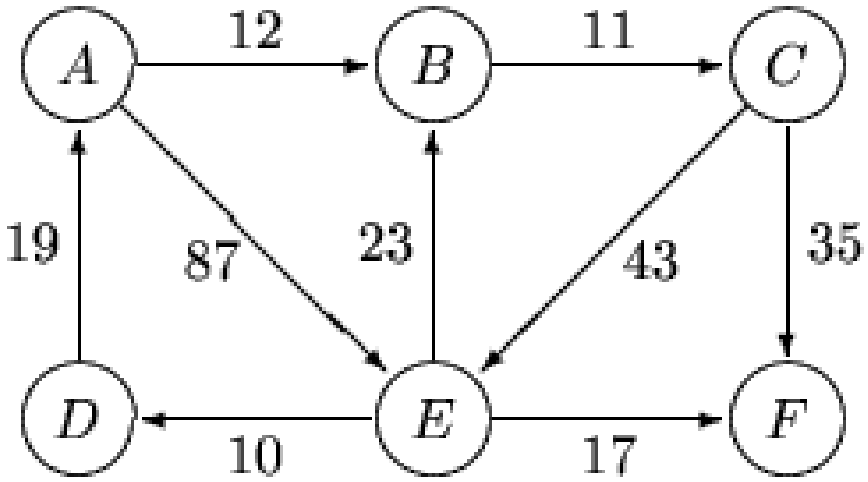


Figure: Directed-weighted graph

- *Dijkstra's –A Greedy Algorithm*

  Greedy algorithms use problem solving methods based on actions to see if there's a better long term strategy. Dijkstra's algorithm uses the greedy approach to solve the single source shortest problem. It repeatedly selects from the unselected vertices, vertex v nearest to source s and declares the distance to be the actual shortest distance from s to v. The edges of v are then checked to see if their destination can be reached by v followed by the relevant outgoing edges.

2. DESCRIPTION OF THE ALGORITHM

Before going into details of the pseudo-code of the algorithm it is important to know how the algorithm works. Dijkstra's algorithm works by solving the sub-problem k, which computes the shortest path from the source to vertices among the k closest vertices to the source. For the dijkstra's algorithm to work it should be directed- weighted graph and the edges should be non-negative. If the edges are negative then the actual shortest path cannot be obtained.

At the k^{th} round, there will be a set called Frontier of k vertices that will consist of the vertices closest to the source and the vertices that lie outside frontier are computed and put into New Frontier. The shortest distance obtained is maintained in sDist[w]. It holds the estimate of the distance from s to w. Dijkstra's algorithm finds the next closest vertex by maintaining the New Frontier vertices in a priority-min queue.

The algorithm works by keeping the shortest distance of vertex v from the source in an array, sDist. The shortest distance of the source to itself is zero. sDist for all other vertices is set to infinity to indicate that those vertices are not yet processed. After the algorithm finishes the processing of the vertices sDist will have the shortest distance of vertex w to s. two sets are maintained Frontier and New Frontier which helps in the processing of the algorithm. Frontier has k vertices which are closest to the source, will have already computed shortest distances to these vertices, for paths restricted upto k vertices. The vertices that resides outside of Frontier is put in a set called New Frontier.

3. PSEUDO-CODE OF THE ALGORITHM

The following pseudo-code gives a brief description of the working of the Dijkstra's algorithm.

```
Procedure Dijsktra (V: set of vertices 1... n {Vertex 1 is the source}
                    Adj[1…n] of adjacency lists;
                    EdgeCost(u, w): edge – cost functions;)
Var: sDist[1…n] of path costs from source (vertex 1); {sDist[j] will be equal to
the length of the shortest path to j}

Begin:
Initialize
{Create a virtual set Frontier to store i where sDist[i] is already fully solved}
        Create empty Priority Queue New Frontier;
        sDist[1]←0; {The distance to the source is zero}

        forall vertices w in V – {1} do {no edges have been explored yet}
                sDist[w]←∞
        end for;
        Fill New Frontier with vertices w in V organized by priorities sDist[w];
endInitialize;

repeat
v←DeleteMin{New Frontier}; {v is the new closest; sDist[v] is  already correct}
forall of the neighbors w in Adj[v] do
        if sDist[w]>sDist[v] +EdgeCost(v,w) then
                sDist[w]←sDist[v] +EdgeCost(v,w)
                update w in New Frontier {with new priority sDist[w]}
        endif
endfor
until New Frontier is empty
endDijkstra;
```

The algorithm illustrates Best-First-Breadth-First-Search. It is the Best-First because the best vertex is New Frontier is selected to be processed next. The search used is Breadth-First, since New Frontier consists of vertices that can be tried next and these vertices are one edge away from the explored vertices.

4. EXAMPLE

The above algorithm can be explained and understood better using an example. The example will briefly explain each step that is taken and how sDist is calculated.

Consider the following example:

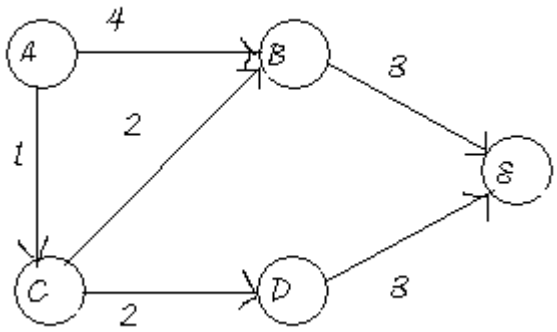


Figure: Weighted-directed graph

The above weighted graph has 5 vertices from A-E. The value between the two vertices is known as the edge cost between two vertices. For example the edge cost between A and C is 1. Using the above graph the Dijkstra's algorithm is used to determine the shortest path from the source A to the remaining vertices in the graph.

The example is solved as follows:

- Initial step
  sDist[A]=0 ; *the value to the source itself*
  sDist[B]= ∞, sDist[C]= ∞, sDist[D]= ∞, sDist[E]= ∞; *the nodes not processed yet*

- Step 1
  Adj[A]={B,C}; *computing the value of the adjacent vertices of the graph*
  sDist[B]=4;
  sDist[C]=2;

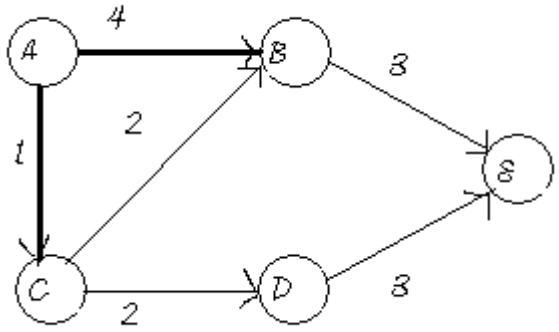


Figure: shortest path to vertices B, C from A

- Step 2
  Computation from vertex C
  Adj[C] = {B, D};
  sDist[B] > sDist[C]+ EdgeCost[C,B]
  4 > 1+2 *(True)*
  Therefore, sDist[B]=3;

  sDist[D]=2;

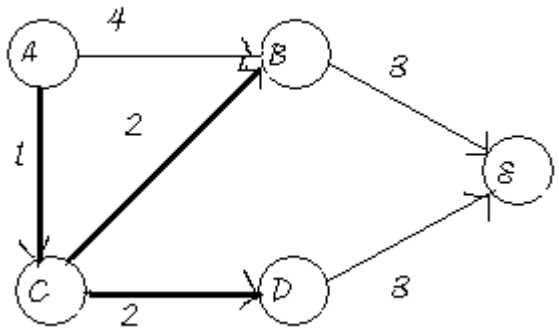


Figure: Shortest path from B, D using C as intermediate vertex

Adj[B]={E};
sDist[E]=sDist[B]+EdgeCost[B,E]
=3+3=6;

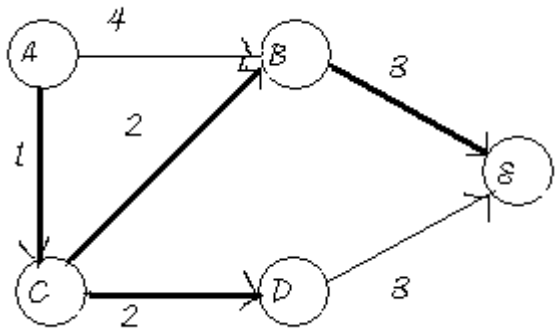


Figure: Shortest path to E using B as intermediate vertex

Adj[D]={E};
sDist[E]=sDist[D]+EdgeCost[D,E]
=3+3=6

This is same as the initial value that was computed so sDist[E] value is not changed.

- Step 4
  Adj[E]=0; means there is no outgoing edges from E
  And no more vertices, algorithm terminated. Hence the path which follows the algorithm is

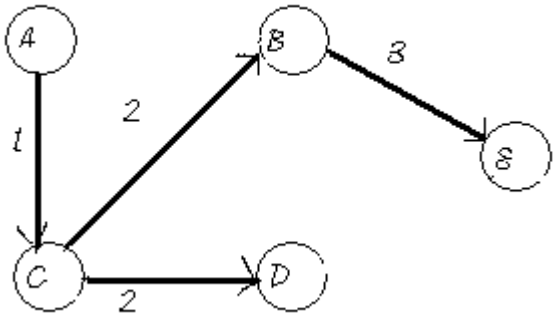


Figure: the path obtained using Dijkstra's Algorithm

5. PROOF OF THE DIJKSTRA'S ALGORITHM

The proof of the algorithm can be obtained by using proof of contradiction. Before proceeding further with proof few facts/lemma have to be stated.

- Shortest paths are composed of shortest paths. It is based on the fact that if there was a shorter path than any sub-path, then the shorter path should replace that sub-path to make the whole path shorter.

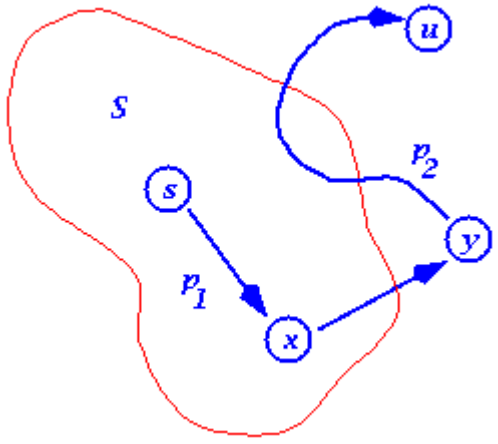
- If s ->...-> u -> v is a shortest path from s to v, then after u is added to Frontier then sDist[v] = EdgeCost[s, v] and sDist[v] is not changed. It uses the fact that at all times sDist[v] >= EdgeCost[s, v]

The distance of the shortest path from s to u is sDist[s, u]. After computing, we get sDist[u] = EdgeCost[s, u] for all u. Once u is added to S, sDist[u] is not changed and should be EdgeCost[s, u].

PROOF BY CONTRADICTION

Suppose that u is the first vertex added to S for which sDist[u] != EdgeCost[s,u]. Note:

- u cannot be s, because sDist = 0.
- There must be a path from s to u. If there were not, sDist[u] would be infinity.
- Since there is a path, there must be a shortest path.



Let s -(p1)-> x -> y -(p2)-> u be the shortest path from s to u. x is within S and y is the first vertex not within S.

When x was inserted into S, sDist[x] = EdgeCost[s, x] (since according to the hypothesise u was the first vertex for which this was not true).

Edge (x,y) was relaxed so that

sDist[y] = EdgeCost[s, y]
<= EdgeCost [s, u]
<= sDist[u]

Now both y and u were in V-S when u was chosen, so d[u] <= d[y]. Thus the two inequalities must be equalities,

sDist[y] = EdgeCost[s, y] = EdgeCost [s, u] = sDist[u]

So sDist [u] = EdgeCost [s,u] contradicting hypothesis.

Thus when each u was inserted, sDist[u] = EdgeCost[s, u].

6. EFFICIENCY

The complexity/efficiency can be expressed in terms of Big-O Notation. Big-O gives another way of talking about the way input affects the algorithm's running time. It gives an upper bound of the running time.

In Dijkstra's algorithm, the efficiency varies depending on |V|=n DeleteMins and |E| updates for priority queues that were used.

If a **Fibonacci heap** was used then the complexity is O(| E | + | V | log | V |) , which is the best bound. The DeleteMins operation takes O(log|V|).

If a **standard binary heap** is used then the complexity is O(| E |log |E|),| E | log |E| term comes from|E|updates for the standard heap .

If the set used is a priority queue then the complexity is $O(|E|+|V|^2)$. $O(V|^2)$ term comes from |V| scans of the unordered set New Frontier to find the vertex with the least sDist value.

7. DIS-ADVANTAGES

The major disadvantage of the algorithm is the fact that it does a blind search there by consuming a lot of time waste of necessary resources.

Another disadvantage is that it cannot handle negative edges. This leads to acyclic graphs and most often cannot obtain the right shortest path.

8. RELATED ALGORITHMS

- **A* algorithm** is a graph/tree search algorithm that finds a path from a given initial node to a given goal node It employs a "heuristic estimate" h(x) that gives an estimate of the best route that goes through that node. It visits the nodes in order of this heuristic estimate. It follows the approach of best first search.
- The **Bellman–Ford algorithm** computes single-source shortest paths in a weighted digraph. It uses the same concept as that of Dijkstra's algorithm but can handle negative edges as well. It has a better running time than that of Dijkstra's algorithm.
- **Prim's algorithm** finds a minimum spanning tree for a connected weighted graph. It implies that it finds a subset of edges that form a tree where the total weight of all the edges in the tree is minimized. it is sometimes called the DJP algorithm or Jarnik algorithm.

9. APPLICATIONS

- Traffic information systems use Dijkstra's algorithm in order to track the source and destinations from a given particular source and destination
- OSPF- Open Shortest Path First, used in Internet routing. It uses a link-state in the individual areas that make up the hierarchy. The computation is based on Dijkstra's algorithm which is used to calculate the shortest path tree inside each area of the network.

10. REFERENCES

- An inside guide to algorithms: their application, adaptation and design: By Alan R Siegel and Richard Cole.
- **http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html**
- **http://tide4javascript.com/?s=Dijkstra**
- **http://www.cs.sunysb.edu/~skiena/combinatorica/animations/dijkstra.html**
- **http://www.unf.edu/~wkloster/foundations/DijkstraApplet/DijkstraApplet.htm**
- **http://www.cs.auckland.ac.nz/software/AlgAnim/dijkstra.html**
- **http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic29/**
- **http://en.wikipedia.org/wiki/Dijkstra's_algorithm**
- **http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml**
- **http://www.cs.auckland.ac.nz/software/AlgAnim/dij-proof.html**
- **http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Graph/**
- **http://www.nist.gov/dads/HTML/dijkstraalgo.html**
- **http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html**
- **http://www.cs.usask.ca/resources/tutorials/csconcepts/1999_8/tutorial/advanced/dijkstra/dijkstra.html**
- **http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/dijkstraAlgor.htm**
- **http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html**