Sada Kurapati

Lost in ever growing technology…

OCT **15** 2014

Graphs & Trees – Shortest Path (BFS, Dijkstra, Bellman Ford)

Before exploring this topic, it will help if you go through my previous blog on graph basics (https://sadakurapati.wordpress.com/2014/01/06/graph-tree-algorithms/) which covers the different types of graphs, their representations & different searching techniques.
The motivation behind the shortest paths are many, for example finding a quick route between two cities, search engine crawling, data transfer to destination over the network, finding relationships on Facebook, Linkedin and many more. Following are the few key elements which we need to keep in mind while implementing any algorithm on graphs.

- The type of graph – Directed vs undirected, weights (+ve & -ve)
- Possibility of cycles – one of the most important
- Density of the graph – this will affect performance a lot
- Do we really need an optimal path – in some cases, it will take a lot of time to calculate this and we might be happy with sub optimal path itself
- And finally, the meaning/definition of shortest – number of edges or vertices, minimum weight in terms of weighted graphs etc.

Basically, if we don't have any weights and the shorted path is the minimum number of edges/vertices, then it is a straightforward problem as we can use Breadth First Search or Depth First Search algorithm to find it. The only additional information which we need to keep in track while doing BFS/DFS is the predecessor information so that we can keep track of the shortest path.

Shortest Path – by BFS

Breadth First Search can be used to find the shortest paths between any two nodes in a graph. To understand BFS in more details, check this post (https://sadakurapati.wordpress.com/2014/01/06/graph-tree-algorithms/)

Following is the pseudo instructions for this algorithm.

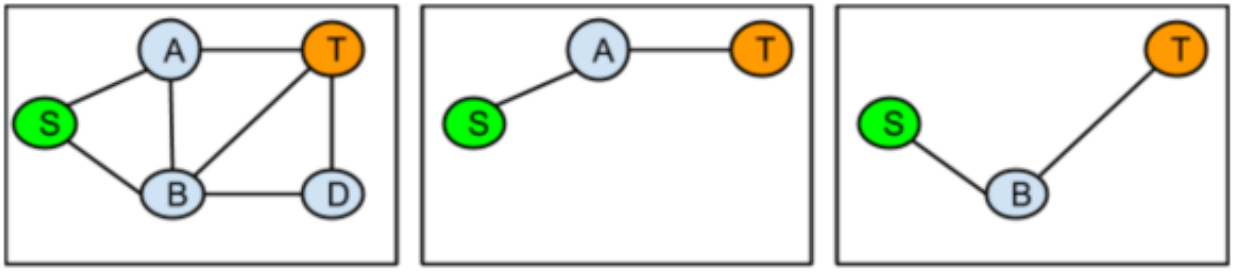
- start at the source node S
- put this in current level bucket
- while we have not ran out of graph (next level bucket is not empty) and not found our target node
  - take a node from current level bucket – call it C
  - for each node reachable from current node C – call it N
    - (we can also check to see if it is visited node and skip it to avoid the cycles)
    - mark the predecessor of N as C
    - Check if N is our target node (that is have we reached the target node)
      - If YES, then exit
  - put this N in next level bucket
  - mark next level bucket as current level

The code for this will be almost same as BFS except we keep track of predecessors so that we can build the path. For this, we can simply use a dictionary. The code for this can be found in my previous blog here (https://sadakurapati.wordpress.com/2014/01/06/graph-tree-algorithms/).

Dijkstra's algorithm

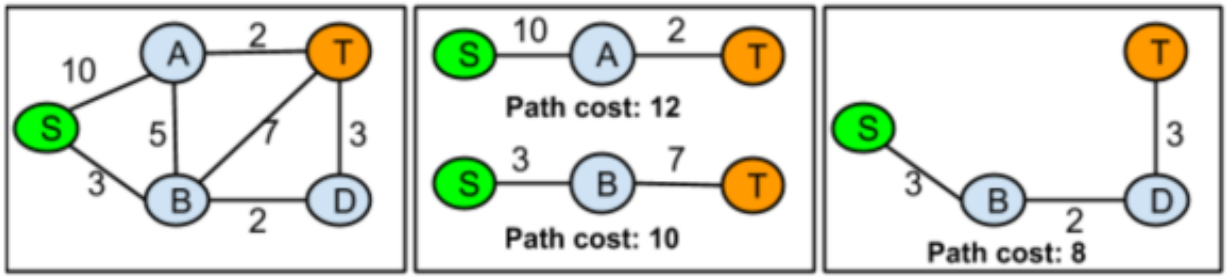
Dijkstra's shortest path algorithm is mainly used for the graphs with +ve weights and in the case of finding single source shortest paths.
What does it mean by single source shortest path?
It means, given a source node (S), this algorithm will provide us shortest paths to all of the nodes in a graph which are reachable from S.
If we need a shortest path between two nodes, we simply stop(exit) executing this algorithm when we find the shortest path to the target node. It is sometimes very hard to find the stopping point for shortest path between two nodes, so we mostly perform full algorithm and then find the shortest path between nodes S and T. There might be also cases where we just need suboptimal solution (a reasonable shortest path) and in this case, we could save some good amount of execution time by simply stopping when we reach our expected suboptimal path. By the way, this algorithm is named after Edsger Dijkstra as he came up with this.

First let us look into an example on why we can't use BFS for weighted graph. Consider the below graph without weights and let us assume we want to find a shortest path between source node S and target node T.



Graph - Shortest paths by BFS

(https://sadakurapati.files.wordpress.com/2014/10/sp_by_bfs.png)
So in this case, we could end up with any one of the path shown above shortest paths from S to T. We don't bother which path we choose as both are the shortest paths between S and T in terms of number of nodes in a path. Now let us assign some weights.



Graph with +ve weights

(https://sadakurapati.files.wordpress.com/2014/10/gp_negative_weights.png)
The above diagram clearly tells us that both the paths by BFS (S–>A–>T & S–>B–>T) are not shortest and the actual shortest path is with the cost 8 and it is S –>B–>D–>T.
Based on this example, we can clear say that,

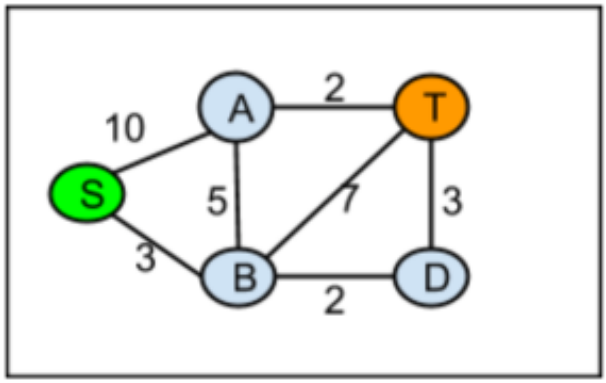
- Path with less number of nodes might not be the shortest path when there are weights
- BFS algorithm will not work in all of the cases when we have a graph with weights

Now let us look into the Dijkstra's algorithm and see how it finds out the shortest path. It basically takes each node and explores all of the paths from that node by calculating the cost and predecessors and performs this till we find the shortest paths. It will be exponential if we simply do the above approach but Dijkstra's algorithm uses the following two techniques to avoid exploration of exponential paths.

- Relaxation
  - Initialize the path cost of all of the nodes to Infinity
  - Then whenever we reach a node, we take the minimum of (older cost, path cost of predecessor + cost between predecessor and this node)
- Topological sort
  - The order of processing each node from graph hugely affects the total execution time. Will look into in detail in below sections on this.

Relaxation Technique

To explain this, let us consider the same example.



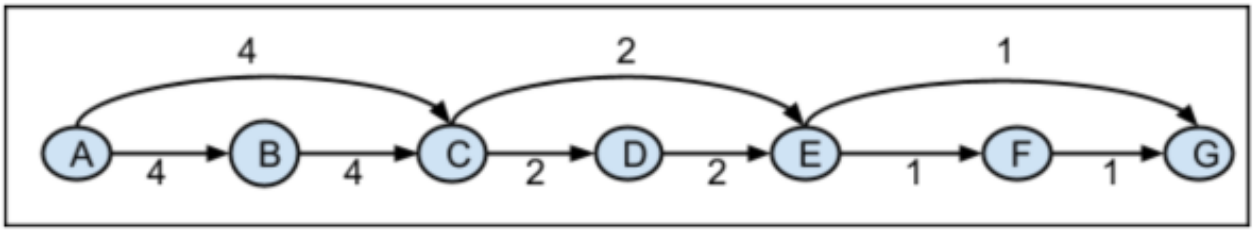
(https://sadakurapati.files.wordpress.com/2014/10/gp_weights_1.png)
There are many paths between S to T and for simplicity, let us assume that we reach node T in the below order.

- S → A → T (cost: 12)
  - In this case, we are reaching T from A, so the total cost is S→ A path cost + cost from A → T and the cost is 10 + 2 = 12
  - So this path cost is MIN (Infinite, 12) = 12. Note that we initialize the cost as Infinity. Update the predecessor of T as A.
- S → B → T (cost: 10)
  - Total cost is cost of S to B + cost of B to T. Cost is 3 + 7 = 10
  - So this path cost is MIN(12, 10) = 10. Found a new path. Update the predecessor of T as B.
- S → A → B → T (cost: 22)
  - Cost is MIN (10, 22) = 10. No change
- S → B → D → T (cost: 8)
  - Cost is MIN (10, 8) = 8. Found a new path. Update the predecessor of T as D.

So once we done with the algorithm execution, we will have the shortest path from S to T as S→ B → D → T.

Topological sort

Now you might be wondering why we need a topological sort at all. Does the relaxation technique won't sort out the problem completely? To answer this, we need to look into a well known example.



(https://sadakurapati.files.wordpress.com/2014/10/gp_topological_sort.png) In this, first let us calculate the weights from A to G by just taking the direct straight links. Then the shortest paths for the vertices B to G will be B-4, C-8, D-10, E-12, F13 & G-14. Now let us look at the relaxation. E → G, there is an edge with weight 1, so the shortest path will be 13 (A-E 12 and E to G 1 = 13). Now look at the edge from C to E. If we take that then the shortest path to E will become 10 (A → C 8, C → E 2), now the shortest path to E becomes 10, then we need to change the weights of F to 11 (from 13) and G to 11 (from 13). This is just an example where the dynamic paths can become exponential and it really hurts the performance of our shortest path algorithm.

Dijkstra uses a priority query which will help us picking up the vertices with minimum path which leads to a better topological sort.
Following is the pseudo instruction-
Dijkstra algorithm for shortest paths to all the vertices in a graph from source vertex 's'

- Initialize a dictionary **S** which holds the shortest paths already calculated.
- Add all of the vertices to a priority queue – **Q**.
- Set the distance from **s** to **s** as zero and to all other vertices as infinite.
- while the Queue is not empty
  - take a vertex '**u**' from query – extract min
  - add **u** to **S**
  - for each vertex **v** which is adjacent to **u**,
    - relax u → v. relaxation step. Minimum (shortest path to u + weight of edge u to v, existing shortest path)

Finally, **S** contains the shortest paths to all of the nodes from graph from source vertex **s**. Basically, it is a greedy algorithm. The topological sort it uses is the order of picking up a vertex from queue with minimum path weight.

Following is the pseudo code
G – graph, s – source, Q – priority query, dist – dictionary stores vertex and shortest path
function Dijkstra (G, s)

- Q, dist = ***InitializeGraph***(G, s)
- while Q is not empty
  - u = Q.extract_min() // retrieves the vertex with minimum path distance.
  - mark u as visited
  - for each adjacent vertex v of u
    - if v is not yet visited
      - *relax*(u, v)

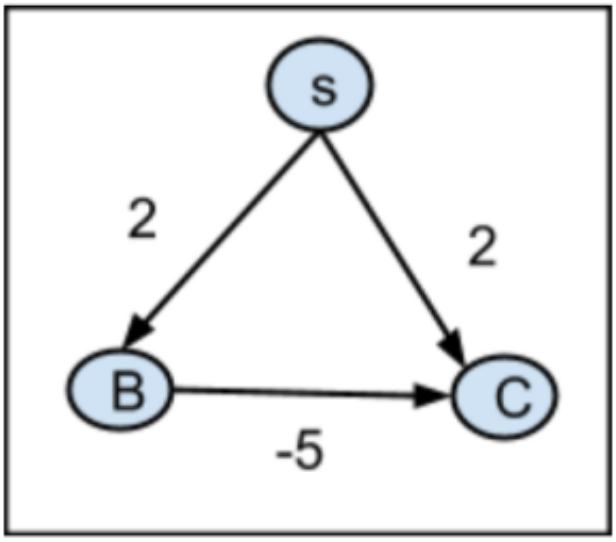
// initializes a graph and returns priority queue
function InitializeGraph(G, s)

- dist[s] =0
- for each v in G
  - if v not equal to s
    - dist[v] = infinity
  - Q.add(v, dist[v]) – priority queue

return Q, dist
function relax(u, v, Q)

- newWeight = dist[u] + weight from u to v
- if newWeight < dist [v]
  - dist[v] = newWeight
  - Q.decrease_priority(v, newWeight)

Why does Dijkstra won't work for a graph with negative edges?
The problem is the relaxation step and the condition of running the algorithm till we don't have any edges which can be relaxed.
Let us look at the following graph and think for a few seconds why it will fail.



(https://sadakurapati.files.wordpress.com/2014/10/gp_negative_weights_1.png)
Here, when we start at s (source),

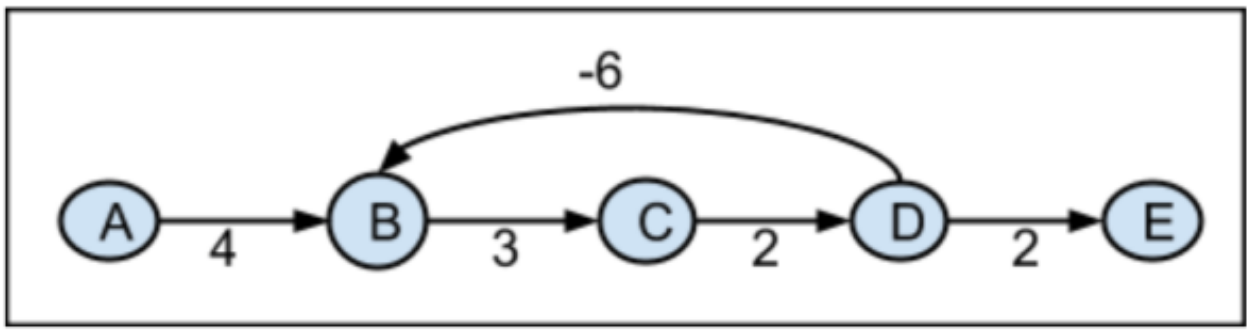
- we set s to s as 0 and all other (s → b & s → c) as infinite
- Then we pull c, we relax that edge (min (infinity, 2)), mark c as visited
- then we pull b, we relax that edge (min (infinity, 2)), mark b as visited.
- No more edges, the algorithm is done and it says the shortest path from s to c is 2.

But in fact, the shortest path from s to c is s → b → C is -3. The simplest reason is, Dijkstra algorithm assumes that the weights will only increase. In this case, when it visited(pulled out from Q) c, it marked it as visited and never going to visit again as it assumes that shortest path to c from anywhere else will be more than 2.

Bellman-Ford

Now let us look into Bellman Ford algorithm and see how it solves the shortest path problem for a graph with negative weights. This algorithm closely follows the Dijkstra algorithm and uses relaxation technique. The only difference is, it executes this $|V| - 1$ times for each edge

where |V| is the number of vertices in the graph.
There is also another aspect which it determines is the negative cycle paths. Let us look at the following example.



(https://sadakurapati.files.wordpress.com/2014/10/belman-ford.png)
In this example, can you determine the shortest path distance from B → D? No, because every time we cycle between B → C → D → B, it reduces by 1. So we can cycles through this infinite times and there is no way we can identify the shortest path from B → D in this graph. So in this case, Bellman-Form algorithm throws error saying it can't find the short path.
Following is the pseudo code for Bellman-Ford
V[] – vertices, E[] – edges, s – source vertex
function Bellman-Ford(V[], E[], s)

- // Initialize the graph
- dist[] = ***InitializeGraph***(V[], s)
- for 1 to size(V) -1 // This is the diffrence from Dijkstra
  - for each edge e (u, v) in E
    - ***relax***(u, v, dist[])
- if ***haveNegativeCycles***(E[], dist[])
  - throw new Error("Graph have negative cycles")

return dist

// initializes a graph
function InitializeGraph(V[], s)

- for each vertex v in V[]
  - dist[v] = infinity
- dist[s] = 0

return dist[]

// relaxation
function relax(u, v, dist[])

- newWeight = dist[u] + weight from u to v
- if newWeight < dist [v]
  - dist[v] = newWeight

function haveNegativeCycles(E[], dist[])

- for each edge e (u, v) in E
  - if dist[u] + weight from u → v < dist[v]

- return true // yes, if can find a edge to relax, then it contains negative cycles

return false // no negative cycles

So basically it follows the Dynamic Programming (DP) approach. When we relax all of the edges once, the graph (dist[]) will have shortest paths with 1 edge, after two cycles, we will have shortest paths with 2 edge paths and so on. After we perform this |V| – 1 times, we will have shortest paths from the source to all of the vertices and we simply throw the error when it contains negative cycles.

Thats all about shortest paths. These are just basics and we do have more complex ones with heuristics and NP-Hard problems like Traveling Salesman problem, but it will be very hard to describe them in a blog post.

Thank you for your time and feel free to provide your feedback through comments.

By Sada Kurapati • Posted in Algorithm, Graphs, Trees • Tagged algorithm, Belman-ford, BFS, Dijkstra, graphs, shortest path

One comment on “Graphs & Trees – Shortest Path (BFS, Dijkstra, Bellman Ford)”

1. **PINGBACK:** Recursion | Sada Kurapati