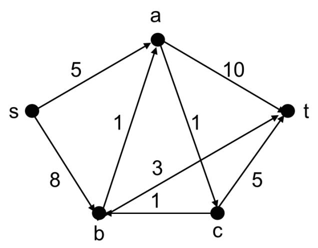
COMPSCI 330: Design and Analysis of Algorithms Spring 2019

Lecture 12: Shortest Paths

Lecturer: Rong Ge *Scribe: Shweta Patwa*

Given a graph $G = (V, E)$ with edges of length $w(u, v) > 0$ (for now, but in some cases you might want to consider negative weights). These weights can represent lengths or cost or travel time, etc. Length of a path equals the sum of the w's for edges on that path.

Goal: given source s and destination t, find paths with minimum length.
Let us consider the following example:



We want to find a path connecting s to t that has the smallest possible length. Different paths from s to t:

1. s to a to t: length is 15
2. s to a to c to t: length is 11
3. s to a to c to b to t: length is 10

The third one is the shortest path for the given example. Note that this graph does not satisfy triangle inequality, so you can get a better path by taking a small detour.

Different setting of the shortest path problem:

1. Find the shortest path from s to t
2. Single-source shortest path from s to all other vertices. Not necessarily harder than the first problem, say when t is the farthest from s.
3. All pair shortest path problem

Today, we focus on the first two problems.

Property of shortest path: Given a shortest path from s to t, any sub-path is still a shortest path between its two endpoints.
Proof sketch (by contradiction). Suppose a sub-path is not the shortest path between its two endpoints. Then, we can use this to get a better shortest path between s and t. But we started with the assumption that this is the shortest path between s and t.

Thus, if you remove a part of the optimal solution, then it is still optimal for some other subproblems. Which basic design technique has this property? Usually dynamic programming has this property. Recall how we solved the knapsack problem. Remaining solution is still an optimal solution for a different knapsack subproblem. This suggests that if we want to find the shortest path from s to t, then we can first find the shortest path from s to v and proceed from there.

The decision that we are making at each step is where do I go next? This means that the last decision would be to choose which edge to take to get to vertex t. There can be many options at each step,

$$d[v] = \min_{(u,v)\in E} w(u,v) + d[u]$$
$$d[s] = 0$$

where $d[u]$ is the length of the shortest path from s to u. Note that we need to know the distances to predecessors of v to be able to correctly compute $d(v)$. In the above figure, $d[a] = 5, d[b] = 6$ and $d[c] = 7$. Then $d[t] = \min\{d[a]+10, d[b]+5, d[c]+3\} = 10$.

Aside: Why might greedy fail? Can always follow the edge with minimum cost. But that is not the best approach. This might not even give you a valid path to t. We leave this as an exercise for students.

Problem: Graph can have cycles. What ordering do we use?

0.1 Dijkstra's algorithm

The correct ordering is an ascending order of distance from the source.

Intuition: To get to a point v, if the last step of the shortest path is (u,v), then u should always be closer to s than v (since edge lengths are positive).

$$d[v] = \min_{\substack{(u,v)\in E\\ d[u]<d[v]}} w(u,v) + d[u]$$
$$d[s] = 0$$

How do we know which vertices have $d[u]$ is smaller than $d[v]$?

Dijkstra's algorithm:

```
Initialize $dis[u]$ to be all infinity, $prev[u]$ to be $NULL$
For neighbors $u$ of $s$, $dis[u] = w(s,u), prev[u] = s$
Mark $s$ as visited
For $i = 2$ to $n$
    Among all vertices not visited, find the one with the smallest $dis[u]$
    Mark $u$ as visited
    For all edges $(u,v)$
        if $dis[v] > w(u,v) + dis[u]$
            $dis[v] = w(u,v) + dis[u]$
            $prev[v] = u$
```

- Mark a vertex u to be visited if we know the shortest path from s to u.
- Maintain $d[u]$ to be the shortest path from s to u that only uses visited vertices as intermediate points.
- At every step, pick the vertex that has not been visited and has the smallest $dis[u]$.

  Claim. $dis[u] = d[u]$.

  - Mark u as visited.
  - Update the data structure for $dis[.]$ accordingly

(For an example run, watch the lecture video.)

Proof of correctness. Base case: Initially know shortest path to s, which is the only visited vertex. $dis[u] = w(s,u)$ if $(s,u) \in E$, and $+\infty$ otherwise.

Induction hypothesis: at ith iteration, know shortest path to i vertices, and for any vertex v that has not been visited, $dis[v]$ stores the length of the shortest path to v through only visited vertices.

The vertex v we choose in the induction step of the proof must satisfy the condition that $dis[v]$ is the shortest possible, and that there is no other shorter path from s to v. Suppose not, i.e., suppose there is a shorter path. We know that the predecessor of v along the shortest path must have the correct $dis[prev[v]]$. Thus, to have an even shorter distance to v, the path needs to use some intermediate vertex v'' that is outside of the visited vertices. Let this be the first unvisited vertex along this path. Length of path to v'' is at least $dis[v'']$ because that did not use any intermediate vertices outside of the visited vertices. We also know that the distance to v'' cannot be smaller than the distance to v. So going to v'' first and then to v cannot yield a shorter path to v.

Lastly, we need to prove that $dis[v]$ values are maintained correctly. Suppose the newly visited vertex is u, and there is another vertex v that lies outside of the set of vertices that have been visited already. Here are the two possible cases:

1. Suppose the path to v using only visited vertices does not use u. Here, the new update to the value of $dis[u]$ does not impact the computation for v. Such paths to v have already been considered in previous iterations of the algorithm.
2. Suppose there is an outgoing edge from u to v. Thus, in the current iteration, we will update $dis[v]$ if the path through u gives a better, shorter path.

   It cannot be the case that the shortest path to v that goes through u could first visit a different already visited vertex v'. Observe that v' was visited before u, so the shortest path to it (from s) cannot go through u. Thus, we do not make any errors in calculating $dis[v]$. ■

Running time analysis

We find the closest vertex n times, and update weights m times. Each edge is only considered once, when its starting visited has been marked visited.

- Naive implementation: implement $dis[]$ as an array. This takes $O(n)$ per vertex, and $O(1)$ per edge. This gives overall $O(n^2 + m)$ or $O(n^2)$ time.

- Binary heap for $dis[]$: heap is a data structure that allows you to find the smallest element in $O(\log n)$ time. Updating one element in a heap also takes $O(\log n)$ time. This gives overall $O(n \log n + m \log n)$ or $O(m \log n)$ time (if the graph is connected). This is not much better than the first implementation because m could be $O(n^2)$.
- Fibonacci heap for $dis[]$: finding the smallest element takes $O(\log n)$ time. However, when updating a distance, it can only decrease because we update distances only if there is a better/shorter path. Thus, updates take $O(1)$ time (average). Hence, we get an overall runtime of $O(m + n \log n)$.