CS 341 - Algorithms, Winter 2016. University of Waterloo

Lecture 10: Single source shortest paths

We study Dijkstra's algorithm to compute shortest paths from a single vertex to all vertices, on graphs with non-negative edge weights.

Shortest paths

Input: Directed graph $G=(V,E)$, with a non-negative edge length l_e for each $e \in G$, two vertices s,t.

Output: A shortest path from s to t, where the length of a path is equal to the sum of edge lengths.

You can think of the graph as a road network, and the edge length represents the time needed to drive pass the road (may depend on traffic).

Then the problem is to find the fastest way to drive from point s to point t.

This is the type of queries that Google Maps answers every day.

To solve this problem, it turns out to be more convenient to solve a more general problem.

Input: Directed graph $G=(V,E)$, with a non-negative edge length l_e for each $e \in G$, and a vertex s.

Output: A shortest path from s to v, for every vertex $v \in V$.

It seems this problem is harder, because each path could be of length $\Omega(n)$, and so the output size could already be $\Omega(n^2)$.

It turns out that there is a succint representation of all these paths, and the general problem can be solved in the same time complexity as the shortest s-t path problem.

We call the general problem the single source shortest paths problem.

Breadth first search

We have seen that BFS can be used to solve the shortest path problem, when every edge of the graph has the same length (as we just counted the number of edges in the graph).

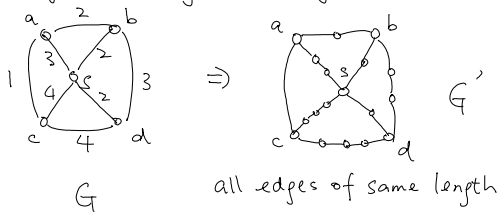
It is not difficult to "reduce" the non-negative length problem to the same length special case.

Say all the edge lengths are positive integers.

We can reduce our problem to the special case by replacing each edge of length k by a path

say all the edge lengths are positive integers.

We can reduce our problem to the special case by replacing each edge of length k by a path of length k.



It is easy to see that this reduction works: there is a path of length k in G iff there is a path of length k in G'.

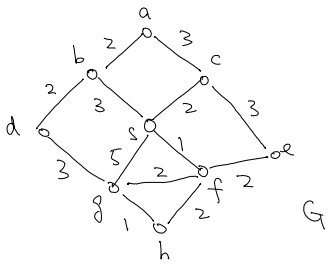
However, this is not an efficient reduction because G' may have too many vertices; the number of vertices in G' is $n + \sum_{e \in G} l_e$, and so when l_e is large G' is too large, and then a linear time algorithm in G' would not correspond to a linear time solution in G.

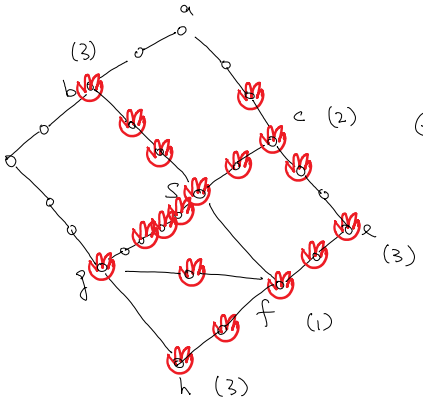
Physical process

Instead of constructing G' explicitly, we can just keep G' in mind and simulate the process of doing BFS in G'.

We can think of the process of doing BFS in G' as follows.

- We set out a fire at vertex s at time 0. The fire will spread out.
- It takes one unit of time to burn an edge of G'.
- The shortest path distance from s to v is just the time when vertex v is burned.





Algorithm

There is a simple way to simulate the fire process without explicitly constructing the graph.

Dijkstra's algorithm (or BFS-simulation)

dist(v) = ∞ for every v∈V.

dist(s) = 0

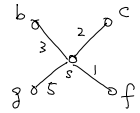
```
Q = make-priority-queue(V)        // all vertices are put in the priority queue,
                                  // using dist[v] as the key value (priority value) of v.
while Q is not empty do
    u = delete-min(Q)             // dequeue the vertex with minimum dist-value
    for each out-neighbor v of u
        if dist[u] + l_uv < dist[v]
            dist[v] = dist[u] + l_uv.   decrease-key(Q, v).   parent[v] = u.
```

This algorithm is syntactically very similar to the BFS algorithm, except that we use a priority queue (which can be implemented by a min-heap) to replace a queue.

The idea is to find out what is the next vertex to be burned.

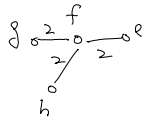
Initially, it is the source vertex s.

Then, knowing that s is burned at time 0, in the example above, we know that b, c, g, f will be burned in at most 3, 2, 5, 1 time steps respectively.



Then, the next vertex to be burned is vertex f.

Knowing that vertex f is burned at time 1, we know that vertices g, h, e will all be burned in 2 time units (by time step 3).



In the example, we use it to update the time of g being burned to be 3 (instead of the initial value of 5 when s is burned), as vertex g will be burned quicker through the fire from f than the fire from s.

Then, we again use delete-min to find out what is the next vertex to be burned, and do the updating the neighbor burning time, and repeat.

Correctness

It should be clear that the algorithm is faithfully simulating the BFS process in G' to compute the burning times.

And also computing BFS in G' is equivalent of computing shortest path distance in G.

So, the correctness of the shortest path algorithm just follows from the correctness of BFS. Okay, we will do a more traditional proof for the cautious reader.

Time complexity

Each vertex is enqueued (in the beginning) and dequeued (when it is burned) once.

When a vertex u is dequeued, we check every edge uv and may use the value $dist[u] + l_{uv}$ to update the value $dist[v]$ (decrease-key).

All the enqueue, dequeue and update operations can be implemented in $O(\log n)$ time using heap, and thus the total time complexity is $O\left(\left(n + \sum_v outdeg(v)\right) \log n\right) = O((n+m)\log n)$ time.

So the cost of simulation is only a factor $\log n$. If you have forgot about priority queues, you can take a look at [DPV 4.5].

With more advanced data structures (i.e. Fibonacci heap), the runtime could be improved to $O(n \log n + m)$. See [CLRS] for Fibonacci heaps.

Analysis

Here we present a more traditional way to analyze the correctness of Dijkstra's algorithm.

The proof technique will be useful in solving other problems as well, e.g. minimum spanning tree.

The algorithm will turn out to be the same, but the way of thinking about it is slightly different.

This is also the way we think of Dijkstra's algorithm as a greedy algorithm.

The idea is to grow a subset $R \subseteq V$ so that $dist[v]$ for $v \in R$ are computed correctly.

Initially, $R = \{s\}$, and then each vertex we add one more vertex to R.

What is the vertex to add in an iteration?

We add the vertex which is closest to R, so in a sense we are growing R greedily.

Algorithm

$dist(v) = \infty \quad \forall v \in V. \quad dist(s) = 0.$

$R = \phi.$

While $R \neq V$ do

pick the node $u \notin R$ with smallest dist-value.

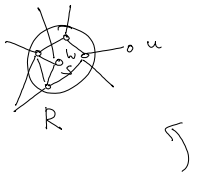
for each edge $uv \in E$

if $dist[u] + l_{uv} < dist[v]$

for each edge $uv \in E$

if $dist[u] + \ell_{uv} < dist[v]$

$dist[v] = dist[u] + \ell_{uv}$



Equivalently, $u \notin R$ is the vertex with $dist[u]$ that $\min_{v \notin R, w \in R} \{ dist[w] + \ell_{wv} \}$.

Exercise: Check that the two algorithms are indeed equivalent.

Induction

We prove the correctness by induction, maintaining the following invariant.

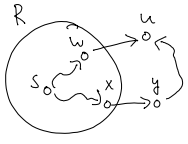
Invariant: The distances $dist[v]$ are computed correctly for any $v \in R$.

Base case: It is true when $R = \{s\}$.

Induction step: Assume that it is true for the current R.

We would like to prove that the invariant remains true when a new vertex u is added to R.

We need to argue that $dist(u) = \min_{v \notin R, w \in R} \{ dist[w] + \ell_{wv} \}$ is the shortest path distance from s to u.



Let $w \in R$ be the vertex in the minimizer, i.e.,

$dist[w] + \ell_{wu} \leq dist[a] + \ell_{ab}$ for all $a \in R$, $b \notin R$.

Since $dist[w]$ is computed correctly by the invariant, we have that $dist[u] \leq dist[w] + \ell_{wu}$.

Consider any other path P from s to u.

There must be an edge xy in P such that $x \in R$ and $y \notin R$ (y could be w).

The length of path P is at least $dist[x] + \ell_{xy}$, since $dist[x]$ is correct by the invariant.

Here, we crucially use the fact that all edges are of non-negative length.

But we know that $dist[w] + \ell_{wu} \leq dist[x] + \ell_{xy}$ as (u,w) is the minimizer.

Therefore, $length(P) \geq dist[x] + \ell_{xy} \geq dist[w] + \ell_{wu}$.

This inequality is true for any s-u path, and thus the shortest path length from s to u is $\geq dist[w] + \ell_{wu}$.

Hence, $dist[u] = dist[w] + \ell_{wu}$, and we have computed $dist[u]$ correctly. □

Remark: We have finally proved the correctness of the shortest path algorithm formally. It is a relief since we didn't do it so formally for BFS. So, we finally put BFS in a firm foundation.

Shortest path tree

Now, we think about how to find the shortest paths from s to v for all v.

From the proof, when a vertex v is added to R, the vertex u that minimizes $dist[u] + l_{uv}$ is the parent on a shortest path from s to v.

We keep track of this parent information in the Dijkstra's algorithm, by always keeping the vertex that attains the minimum of $dist[u] + l_{uv}$ for $u \in R$ as the current parent (and will update if a new vertex is put in R and makes this value smaller).

By tracing the parents until we reach the source vertex s, we find a shortest path from s to v.

Every vertex can do the same to find a shortest path from s.

As in BFS tree, the edges $(v, parent[v])$ form a tree.

We can prove it by induction. Assume that it forms a tree in R. It remains to be a tree in R after we add a new vertex to R.

So, we have a succint way to store all the shortest paths from s, using only $n-1$ edges to store n paths!

This shortest path tree is very useful.

Question: Can you see where the algorithm fails if there are some negative edge length? We will come back to this when we study dynamic programming algorithms.

References: [KT 4.4] [DPV 4.4]