CS 341 - Algorithms, Spring 2021. University of Waterloo

Lecture 10: Single Source Shortest Paths

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

Shortest Paths

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

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

We 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 a fastest way to drive from point s to point t, which is the type of querier that Google Maps answers every day.

It will be more convenient to solve a more general problem.

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

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

This problem seems to be harder, because each path could have $\Omega(n)$ edges, and so the output size could already be $\Omega(n^2)$.

But it will turn out that there is a succinct representation of these paths and this single source shortest path problem can be solved in the same time complexity as the shortest s-t path problem.

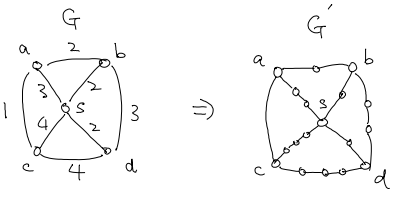
Breadth First Search and Dijkstra's Algorithm

We have seen in LO5 that BFS can be used to solve the single-source shortest paths problem, when every edge has the same length (so we count the number of edges on the paths).

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

Let's say all the edge lengths are positive integers.

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



Then there is an s-t path of length k in G iff there is an s-t path of length k in G'.

And then we can solve the problem in G' by simply doing BFS starting from s.

The only problem of this reduction is that it may not be efficient as G' may have many more vertices. The number of vertices in G' is $n + \sum_{e \in E} (l_e - 1)$, so when l_e is large then G' has many more vertices. And then a linear time algorithm for G' does not correspond to a linear time algorithm for G.

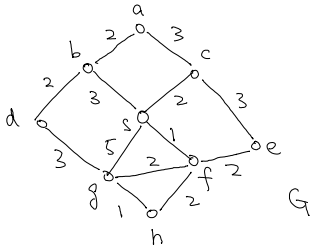
Physical Process

To design an efficient algorithm through this reduction, instead of constructing G' explicitly and run BFS on it, we just keep G' in mind and try to simulate BFS on G' efficiently.

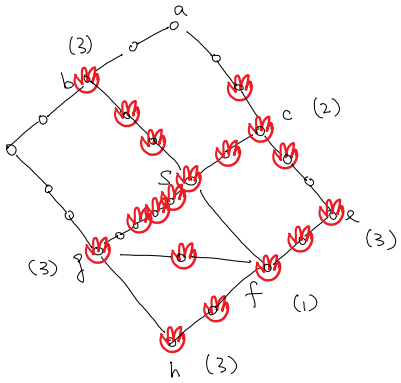
We can think of the process of doing BFS on G' as follows.

- We start a fire at vertex s at time 0. The fire will spread out.
- It takes one unit of time to burn an edge in G'.

Then the shortest path distance from s to t is just the first time when vertex t is burned.



G

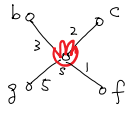


G' at time $t=3$

To simulate this process efficiently, the idea is that we just need to be able to keep finding out what is the next vertex to be burned and when (rather than simulating it faithfully on the paths).

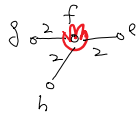
Initially, it is the source vertex s.

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



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

Knowing that f is burned at time 1, we know that g, h, e will all be burned in <u>at most</u> 2 time units (by time step 3) through the edges fg, fh, fe.



In the above example, we update the <u>upper bound</u> on g to be burned to be 3 (replacing the initial value of 5 which was set when s was burned), as vertex g will be burned earlier through the fire from g than the fire from s.

Then, with this updated information after g is burned, we find out the next vertex to be burned (in this example c) and then update the upper bound on the neighbors of c as the fire can now go from c to its neighbors.

Repeating this (ie. find out next vertex, update upper bound on neighbors) gives us an efficient simulation of the process of G'. and this is exactly Dijkstra's algorithm.

Dijkstra's Algorithm (or BFS simulation)

```
dist(v) = ∞ for every v ∈ V        // the initial upper bound on the time v is burned
dist(s) = 0                        // start a fire at vertex s at time 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
                                   //   i.e. find out the next vertex to be burned
    for each (out-)neighbor v of u // it works for directed graphs as well
        if dist(u) + l_uv < dist(v) // find a shorter way to get to v through u
            dist(v) = dist(u) + l_uv
            decrease-key(Q, v)
            parent(v) = u          // note that this may be updated multiple times
```

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

Correctness

I hope it is clear that the algorithm is an efficient simulation of BFS in G'.

Also we argued before that shortest paths from s in G' are shortest paths from s in G.

So, the correctness of this algorithm in G just follows from the correctness of using BFS to solve shortest paths in G' (where every edge is of the same length), which we proved in L05.

Anyway, we will do a more traditional and formal proof as well.

Time Complexity

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

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

All the enqueue, dequeue, decrease-key operations can be implemented in $O(\log n)$ time by a min-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 simulating BFS in the big graph is just an extra factor of $\log n$.

Take a look at [DPV 4.5] or [CLRS] on background of priority queues.

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

Analysis

Here we present a more traditional approach to prove the correctness of Dijkstra's algorithm. Besides being more formal, the proof technique is also useful in analyzing other problems, e.g. MST.

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 $dist(u)$ for $u \in R$ are computed correctly. Initially, $R = \{s\}$, and then at each iteration we add one more vertex to R.

Which vertex to be added in an iteration?

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

Algorithm

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

$R = \phi.$

While $R \neq V$ do

pick the vertex $u \notin R$ with smallest $dist(u)$. $R \leftarrow R \cup \{u\}$.

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 $dist(u) = \min_{v \notin R, w \in R} \{ dist(w) + \ell_{wv} \}$.

We leave it as a simple exercise that the two versions of Dijkstra's algorithm are equivalent.

Induction

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

Invariant: For any $u \in R$, $dist(u)$ is the shortest path distance from s to u.

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

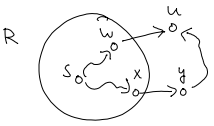
Induction step: Assume that the invariant holds for the current R.

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

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

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

$dist(u) = dist(w) + l_{wv} \leq dist(a) + l_{ab} \quad \forall\, a \in R$ and $b \notin R$.



Since $dist(w)$ is the shortest path distance from s to w, the shortest path distance from s to u is at most $dist(w) + l_{wu}$ (i.e. using the path from s to w and edge wu).

So, it remains to prove that the shortest path distance from s to u is at least $dist(w) + l_{wu}$.

Consider any path P from s to u.

Since $s \in R$ and $u \notin R$, there must be an edge xy in P such that $x \in R$ and $y \notin R$.

(Note that x could be w and/or y could be u.)

The length of path P is at least $dist(x) + l_{xy}$, as $dist(x)$ is computed correctly by the invariant.

★ Here, observe that we crucially use the fact that the length of each edge is non-negative.

But we know that $dist(w) + l_{wu} \leq dist(x) + l_{xy}$ as (w,u) is a minimizer.

Therefore, we conclude that $length(P) \geq dist(x) + l_{xy} \geq dist(w) + l_{wu} = dist(u)$.

This inequality is true for any path from s to u, and thus the shortest path distance from s to u is at least $dist(u)$. □

Note that the proof applies to directed graphs as is.

Shortest Path Tree

Now, we think about how to store the shortest paths from s for all $v \in V$.

From the proof, when a vertex v is added to R, any 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 throughout Dijkstra's algorithm, by always keeping a vertex that attains the minimum of $dist(u) + l_{uv}$ for $u \in R$ as the current parent, and will update whenew vertex is put in R and makes this value smaller.

After the algorithm finished (i.e. $R = V$), by tracing the parents until we reach the source s, we can 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. The proof is left as an exercise.

So, we have a succinct 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 clearly how the algorithm will fail if there are some negative edge lengths?

We will come back to this more difficult setting when we study dynamic programming algorithms.

References: [KT 4.4], [DPV 4.4]