BFS for shortest paths

In the general case, BFS can't be used to find shortest paths, because it doesn't account for edge weights. If every edge weight is the same (say, one), however, the path that it finds *is* a shortest path. How can we use this to our advantage?

Say that we have a graph $G = (V, E)$ with small, positive integer edge weights ($\forall e \in E$, the cost $c(e) \leq k$) and we want the shortest s-t path. Normally we'd be thinking Dijkstra; we have nonnegative edge weights and we only want a single-source shortest path. Dijkstra's algorithm, as you recall, has time complexity $O(E + V \lg V)$.

Can we do better? A common technique in applying graph algorithms is to transform the graph $G = (V, E)$ into another graph $G' = (V', E')$ that is easier to run the algorithms we've studied on, and then to transform the result you found on G' back to the result you really wanted on G.

Here, we'll start with the same vertices V, but we'll replace each edge $e = (u, v) \in E$ with a u-v path of length $c(e)$. (Some extra vertices will be created, too, to link the edges in the path.) The total weight of this u-v path in G' will be $c(e)$, the same as the weight of the (single-edge) u-v path in G. Thus, a shortest s-t path in G' corresponds directly to a shortest s-t path in G, although we'll have to walk the path and discard all of the intermediate vertices that don't exist in G (and which we don't really care about, having created only for the purposes of solving the problem).

Most interestingly, every edge $e \in E'$ has $c(e) = 1$, so (like we discussed initially) we can use BFS to find a shortest path.

We know that BFS is $O(V+E)$ for $G = (V, E)$, but we ran it on G'. So it's more accurate to say that the time complexity of our algorithm is $O(|V'|+|E'|)$. How can we express this in terms of $|V|$ and $|E|$? Well, we create at most k edges in G' for each edge in G, so $|E'| = O(k|E|)$. We retain the original vertices and at at most $k-1$ intermediate vertices per edge, so $|V'| = O(|V|+(k-1)|E|)$. Thus, the BFS execution has time complexity $O(|V| + k|E|)$, which should make sense.

We also need to account for the time complexity of the transformation to and from G'. Here, creating G requires an $O(|V|)$-time operation (copying the original vertices) and an $O(k|E|)$-time operation (creating the $O(k)$ vertices and edges for each original edge). Transforming back (converting our shortest path in G' to a shortest path in G) just requires walking the path and discarding any vertex not in G, which is $O(k|E|)$, the maximum length of an acyclic path in G'. None of these things increases the time complexity of our algorithm, so we're done.

The algorithm we've described has total time complexity $O(|V| + k|E|)$, like was mentioned previously; Dijkstra is $O(|E| + |V| \lg |V|)$. Which is better, naturally, depends on your specific graph and how large k is. The general technique, however, of creating a graph like our G' that has an interesting property G doesn't have (here, equal edge weights) is useful both in theory and in practice.

Traffic planning

Ben Bitdiddle is planning to drive across a city, and has represented the road network as a graph. Each edge has a weight $c(e, t)$ describing how much fuel he'll burn idling at badly-timed traffic lights and honking at awful drivers. c is also a function of time because some roads are more congested at different times during the day. He needs to arrive at some point in the next 12 hours, but beyond that, his objective is to minimize fuel consumption.

Let's create a new graph G' that contains several copies of the vertices $v_i \in V$ from the original graph G. We'll use 13 copies, one for $t = 0$ and then for each hour up until Ben's deadline. We'll name these vertices $v_{i,t}$. We create an edge between two vertices $v_{i,t}$ and $v_{j,t'}$ if and only if it's possible to drive between points i and j in $t' - t$ time (suppose we have a time lookup table to find out about this). The weight of this edge is the weight of the (i, j) edge in G which depends on t as well. We can also create some other edges; for example, Ben is always free to turn his car off and wait. This advances time and consumes no fuel (i.e., is a zero-weight edge).

The shortest path from our starting location at $t = 0$ ($v_{s,0}$ in G') to our destination at $t = 12$ ($v_{d,12}$) will be a solution to the problem!

How I met your midterm

This problem appeared on Quiz 2 in Spring 2011.

Ted and Marshall are taking a roadtrip from Somerville to Vancouver (that's in Canada). Because it's a 52-hour drive, Ted and Marshall decide to switch off driving at each rest stop they visit; however, because Ted has a better sense of direction than Marshall, he should be driving both when they depart and when they arrive (to navigate the city streets).

Given a route map represented as a weighted undirected graph $G = (V, E, w)$ with positive edge weights, where vertices represent rest stops and edges represent routes between rest stops, devise an efficient algorithm to find a route (if possible) of minimum distance between Somerville and Vancouver such that Ted and Marshall alternate edges and Ted drives the first and last edge.

Solution

There are two correct and efficient ways to solve this problem. The first solution makes a new graph G'. For every vertex u in G, there are two vertices u_M and u_T in G': these represent reaching the rest stop u when Marshall (for u_M) or Ted (for u_T) will drive next. For every edge (u, v) in G, there are two edges in G': (u_M, v_T) and (u_T, v_M). Both of these edges have the same weight as the original.

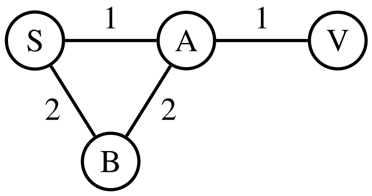
We run Dijkstra's algorithm on this new graph to find the shortest path from Somerville$_T$ to Vancouver$_M$ (since Ted drives *to* Vancouver, Marshall would drive next if they continued). This guarantees that we find a path where Ted and Marshall alternate, and Ted drives the first and last segment. Constructing this graph takes linear time, and running Dijkstra's algorithm on it takes $O(V \log V + E)$ time with a Fibonacci heap (it's just a constant factor worse than running Dijkstra on the original graph).

The second correct solution is equivalent to the first, but instead of modifying the graph, we modify Dijkstra's algorithm. Dijkstra's algorithm will store two minimum distances and two parent pointers for each vertex u: the minimum distance d_{odd} using an odd number of edges, and the minimum distance d_{even} using an even number of edges, along with their parent pointers π_{odd} and π_{even}. (These correspond to the minimum distance and parent pointers for u_T and u_M in the previous solution). In addition, we put each vertex in the priority queue twice: once with d_{odd} as its key, and once with d_{even} as its key (this corresponds to putting both u_T and u_M in the priority queue in the previous solution).

When we relax edges in the modified version of Dijkstra, we check whether $v.d_{\text{odd}} > u.d_{\text{even}} + w(u, v)$, and vice versa. One important detail is that we need to initialize Somerville.d_{odd} to ∞, not 0. This algorithm has the same running time as the previous one.

A correct but less efficient algorithm used Dijkstra, but modified it to traverse two edges at a time on every step except the first, to guarantee a path with an odd number of edges was found. Many students incorrectly claimed this had the same running time as Dijkstra's algorithm; however, computing all the paths of length 2 (this is the ***square*** of the graph G) actually takes a total of $O(VE)$ time, whether you compute it beforehand or compute it for each vertex when you remove it from Dijkstra's priority queue. This solution got 5 points.

The most common mistake on the problem was to augment Dijkstra (or Bellman-Ford) by keeping track of either the shortest path's edge count for each vertex, or the parity of the number edges in the shortest path. This is insufficient to guarantee that the shortest odd-edge-length path is found, and this solution got 2 points. Here is an example of a graph where the algorithm fails: once the odd-edge-count path of weight 1 to A is found, Dijkstra will ignore the even-edge-count path of weight 4 to A since it has greater weight. As a result, the odd-edge-count path to V will be missed entirely.



Another common mistake was to use Dijkstra, and if the path Dijkstra found had an even number of edges, to attempt to add or remove edges until a path with an odd number of edges was obtained. In general, there is no guarantee the shortest path with an odd number of edges is at all related to the shortest path with an even number of edges.

Some algorithms ran Dijkstra, and if Dijkstra found a path with an even number of edges, removed some edge or edges from the graph and re-ran Dijkstra. This algorithm fails on the following graph, where the shortest path with an odd number of edges uses *all* the edges and vertices (note that we visit A twice; the first time, Ted drives to A, and the second time, Marshall drives to A):

One last common mistake was to attempt to use Breadth-First Search to label each vertex as an odd or even number of edges from Somerville (or sometimes to label them as odd, even, or both). This does not help: the smallest-weight path with an odd number of edges could go through any particular vertex after having traversed an odd or even number of edges, and BFS will not correctly predict which. These solutions got 0 points.

Algorithms which returned the correct answer but with exponential running time got *at most* 2 points.