CS2040C Semester 1 2018/2019

Data Structures and Algorithms

Tutorial 11 - Shortest Paths

For Week 13 (Week Starting 12 November 2018)

Document was last modified on: November 8, 2018

1 Introduction and Objective

In this tutorial, we will discuss the last topic for this module: Single-Source Shortest Paths (SSSP) problem and continue talking about the 'graph modeling' soft skill, i.e. ability to model a seemingly random (non-explicit-graph) problem into a graph problem (specifically the SSSP problem for this tutorial).

We will use `https://visualgo.net/en/sssp` during our discussion in this tutorial.

The SSSP problem is quite easily found in many real life applications and it is the source of many interesting Computer Science problems, as you can see in this tutorial. Again, we recommend that you put some thought into it before discussing the potential solutions with your tutor.

As today will be the last tutorial session with your tutor, we shall end each tutorial group with a class photo.

Standard Stuff

During your self-study via VisuAlgo e-Lecture and in real life class discussions, you were presented with these SSSP algorithms: Bellman-Ford's algorithm (for general case, but also the slowest), BFS (only for unweighted graph), the original version of Dijkstra's algorithm (as defined by Dijkstra himself) and the modified version of Dijkstra's algorithm which uses **Lazy Deletion** technique on a priority queue.

First (or even optional), the tutor will (re-)demonstrate the executions of these algorithms on a small directed weighted graph using `https://visualgo.net/en/sssp` from a certain source vertex s. The tutor will re-explain when a certain algorithm can be used and when the same algorithm cannot be used. The tutor may invite some students to do this live demonstration using different source vertex s and/or using different graph. If needed, the tutor will explain **Lazy Deletion** again in the context of PS3. If the class has no issue with the basics, the tutor will proceed to discuss harder topics.

This part is left to the tutor and should not take too much time. These are some of the key 'troublesome concepts' to be highlighted (from historical records):

1. Bellman Ford's algorithm can be 'optimized' a bit from $O(VE)$ to $O(kE)$ where k is the minimum number of rounds **plus one more** that are needed to actually solve the SSSP problem. The maximum k is still V-1, but for many cases, it can be much smaller than V-1. This feature has been integrated in `https://visualgo.net/en/sssp`.
2. BFS will only work if the graph is unweighted, or... all edges have the same **positive** constant weight c that can be transformed back to edges with weight 1 by dividing all edges with that same constant weight c (not many realize this, try to emphasize this idea). Note that if the graph contains negative constant weight $-c$, we may NOT be able to do this transformation if there is at least one negative weight cycle present.
3. Both Dijsktra's algorithm variants are perfectly equal if the edge weights are non-negative and both run in $O((V + E) \log V)$.
4. The key difference of the two Dijkstra's algorithm variants that have to be highlighted is that the Original Dijkstra's cannot handle negative weight edges without potentially generating wrong answers but the Modified Dijkstra's cannot handle negative weight cycles without getting trapped in an infinite loop (and there is no way to give a good terminating condition due to the presence of Modified Dijkstra's killer test case below).
5. Which one should we use? The answer is 'it depends'. In `https://visualgo.net/en/sssp?slide=8-5`, we see that Modified Dijkstra's is not an 'all-conquering' SSSP algorithm as it can also 'be-killed' with 'that kind of test case'. The tutor will re-demonstrate 'that kind of test case' again.
6. 'Lazy deletion' is a technique such that when an element is supposed to be deleted from a Data Structure, we *be lazy* and do not delete it from the Data Structure immediately. If we can keep track of elements that are supposed to be deleted and only delete them when it affects our usage of the Data Structure, this can greatly simplify our code for deletion in many Data Structures. For 'lazy deletion' in priority queues, the key insight is that if any element that is *supposed to be deleted* (but not deleted) is not the at top of the priority queue, then it does not affect the usage of the priority queue. (i.e.: 'pq.top()' will return the same value regardless whether the item is deleted or not) However, when the element that is *supposed to be deleted* affects our usage of the Priority Queue (i.e. at the top of the heap), we can easily remove it as deleting from the top of the heap is much easier. This idea can be applied to many other linear or non-linear data structures as well. :)

Graph Modeling Exercises, via Past Paper Discussions

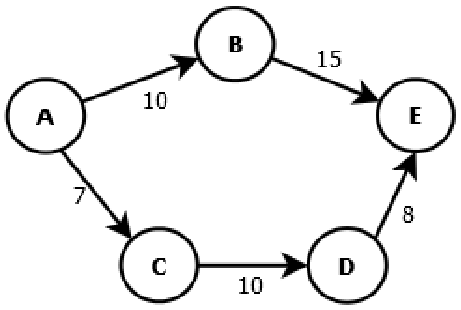
Q2. Please download CS2010 Written Quiz 2 Paper and solve a question titled: Money Changer. You may download the paper here: `https://www.comp.nus.edu.sg/~stevenha/cs2040c/tests/CS2010-2011-12-S1-WQ2-medium.pdf`

Key ideas only, detailed discussions by tutor:

- Each currency is as a vertex
- There is a directed edge from vertex u to vertex v if you can exchange the currency represented by vertex u to the currency represented by vertex v. The edge weight is the exchange rate.
- In this graph, we want to find at least one 'profitable' cycle, that is if we multiply all edge weight in such a cycle, the result > 1. If you understand this part, you can modify the standard Bellman Ford's algorithm to check for such 'profitable cycle' this way. But if you are not sure, read on.
- Now for any given profitable cycle $u_1 \sim u_1$,
  $weight(u_1, u_2) \times weight(u_2, u_3) \times ... \times weight(u_i, u_1) > 1$ iff
  $\log(weight(u_1, u_2)) + \log(weight(u_2, u_3)) + ... + \log(weight(u_i, u_1)) > 0$.
  This technique is likely has to spot unless you have been exposed to it first, like what we are doing to you now.
- Now if we negate the logarithms, we have:
  $-\log(weight(u_1, u_2)) + -\log(weight(u_2, u_3)) + ... + -\log(weight(u_i, u_1) < 0$.
- So if we transform the exchange rate to the negative of its logarithm and use it to represent the edge weights, a 'profitable' cycle will means a negative cycle. From here, the answer becomes 'trivial'.
- Using the standard $O(VE)$, or in this case $O(nm)$ Bellman Ford's algorithm, we can easily detect whether such a cycle exists by running 1 more iteration after the (V-1)-th iteration and checking if any vertex has its shortest path distance relaxed. If there is, there is a profitable cycle. Done.

Note that in real life, it is not really possible to gain profit using this technique due to the difference of buying and selling rates of each currency. Doing this will just make the money changer happier (and you simply lose your money).

Q3. A salesman frequently needs to drive from one city to another to promote his products. Since time is of the essence, he wants to use the shortest route to get from one city to another. However in every city he passes he will have to pay a toll fee. The toll fee is the same for every city and it is a positive unit. Therefore, given two different routes of the same distance (positive unit) to get from city A to city B, he will prefer the one which passes through fewer cities. An example is shown below:



To get from A to E, route A,B,E is preferred over route A,C,D,E even though both have the same cost 25, since A,B,E goes through fewer cities.

Figure 1: An Illustration

Propose the *best* algorithm using what you have learnt so far (and a bit more), so that the salesman will choose a route from any source city A to any destination city B such that it has the shortest distance and also passes through the fewest cities. What is the running time for your algorithm?

Near complete answer (we shall prevent to give full details as it is related to PS5 and Lab TA will complete this discussion during PS5 debrief): Modify the priority queue of Dijkstra's algorithm a bit. From storing pairs $(d[v], v)$, we store triplets $(d[v], h[v], v)$, where $h[v]$ stores how many hops/vertices have been used in the shortest path so far. On the sample graph above, the execution of Original Dijkstra's is shown below. Observe the last step involving vertex E.

1. PQ = {(0, 1, A), (∞, ∞, B), (∞, ∞, C), (∞, ∞, D), (∞, ∞, E)} // process (0, 1, A)
2. PQ = {(7, 2, C), (10, 2, B), (∞, ∞, D), (∞, ∞, E)} // process (7, 2, C)
3. PQ = {(10, 2, B), (17, 3, D), (∞, ∞, E)} // process (10, 2, B)
4. PQ = {(17, 3, D), (25, 3, E)} // process (17, 3, D), it will NOT? change (25, 3, E) as (25, 3, E) is better? than (25, 4, E) from (17, 3, D)
5. PQ = {(25, 3, E)} // do nothing
6. PQ ={} // done

”vl”:”0”:”cx”:120,”cy”:140,”text”:”0”,”state”:”default”,”1”:”cx”:260,”cy”:100,”text”:”1”,”state”:”default”,”2
Now, is that implementation (that is correct for THIS question and runs in $O((V + E) \log V)$ as with the standard time complexity of Original Dijkstra's) also 100% correct **for PS5 Subtask C**?

Q4. Please download CS2040C Final Examination and solve Question C.4 and C.5: Shrinking Tunnel. You may download the paper here: `https://www.comp.nus.edu.sg/~stevenha/cs2040c/tests/CS2040C-2017-18-S1-final.pdf`.

For C.4, basically the weights of all edges are constant, so we can just use BFS to count the minimum number of edges/hops to reach junction $n-1$ from the source junction 0 (SSSP on constant-weighted, a.k.a. unweighted graph). Let's say the answer is k edge(s). Raise the constant weight f to this power k as the final answer.

Common mistakes: Using Dijkstra's for this part (not the best algorithm by extra $\log n$ factor), output $f \times k$ instead of f^k. ^ in C++ is **not** the power function, but actually exclusive-or (xor).

First, we need to transform the weight f to its log, as $\log f$ will be negative when $0.0 < f < 1.0$ and multiplication of values change to summation of logarithmic values. This is similar to the Money Exchange problem (Q2).

In an all-negative weighted graph, the SSSP problem is ill-defined BUT the Single-Source Longest Path (SSLP) problem is perfectly fine (recall that we want our final height to be as HIGH/max as possible). We modify Dijkstra's to take the max instead of the min to solve this SSLP problem. See the code below:

Source: `https://open.kattis.com/problems/getshorty` (CC-by-3.0).

Class Photo and CS2040C Facebook Group

Let's take a class photo with your tutor.
In addition, there is a Facebook group for previous batches of CS2040C students (maintained by Dr. Steven Halim). Now that this iteration of the module is coming to an end, we would **like to invite you** to join this Facebook group, which contains many seniors of CS2040C and **will be used for the next iteration of CS2040C**. The link is `https://www.facebook.com/groups/NUS.CS2040C/`. Please do answer the question asked when you request to join, for verification purposes.

All the best for your final assessment of this module and of your other modules.