CSCI 270 Homework 2

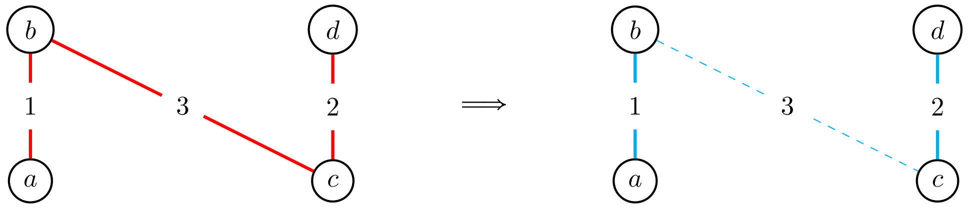
Qilin Ye

September 22, 2022

In both questions, after assuming the graph is connected (since KT does in his book), the number of edges is at least $n-1$. Therefore, we can safely replace $\mathcal{O}(n)$ by $\mathcal{O}(m)$.

Problem 1: MST via Contracted Graph

Solution for 1(a). (V, E') is not necessarily connected, as shown in the example below. The resulting graph does not contain the edge (b, c).



However, (V, E') is guaranteed to be acyclic. Suppose for contradiction that there exists a cycle in this graph. Let the cycle be $(V_c, E_c) = \{(v_i)_{i=1}^n, \{(v_1, v_2), (v_2, v_3), \ldots, (v_n, v_1)\}\}$. Note (v_1, v_2) and (v_n, v_1) are both incident on v_1. We WLOG assume (v_1, v_2) has smaller weight (and in fact smallest weight among all v_1-incident edges in the original (V, E)). Since (v_2, v_3) is also in E_c, the edge cost of (v_2, v_3) must be smaller than that of (v_1, v_2). More generally, for $1 \leqslant i \leqslant n-2$, (v_{i+1}, v_{i+2}) is present in E_c given (v_i, v_{i+1}) is, we must have $c_{(v_{i+1}, v_{i+2})} < c_{(v_i, v_{i+1})}$. Chaining these inequalities we see

$$c_{(v_1,v_2)} > c_{(v_2,v_3)} > \cdots > c_{(v_{n-1},v_n)} > c_{(v_n,v_1)}.$$

This contradicts our assumption that $c_{(v_1,v_2)} < c_{(v_n,v_1)}$. Hence (V, E') cannot contain any cycle.

Proof of 1(b).

- (The output is a spanning tree.) The number of connected components strictly decreases after each iteration: every time when an edge is drawn from a connected component to outside, two originally disjoint components now merge to form one new connected component. (We show a stricter bound in the runtime section.) Since we start with finite number of components (n), the algorithm will end up having only one connected component, namely V, and when this happens, T is a spanning tree.

- (The output is minimal.) Let the output of this algorithm be T and suppose T', different from T, is a/the MST. Since T' is optimal and T is a spanning tree, it is impossible to have $T \subset T'$. Hence there exists edges in T that are absent in T'. Pick $e = (v_1, v_2) \in T \backslash T'$. By the algorithm, e is the cheapest edge coming out of the connected component containing either v_1 or v_2. It follows that e_1 is the cheapest cut between this connected component and its complement, so by the cut property it must also be contained in T', contradiction. Therefore T is the (unique) MST.

- (Runtime.) If we start with k connected components during the beginning of an iteration, we end up with at most $\lfloor k/2 \rfloor$ connected components after this iteration: each component is connected to another component, and the worst case is if all new connections are made pairwise (if we start with odd number of components, at least one newly merged components will consist of $\geqslant 3$ original components). There are therefore $\log n$ iterations with m checks per iteration. (Since we are only considering runtime complexity but not space complexity, we can get as many arrays as we want! After each iteration, do BFS and identify the connected components. Initiate an array of size m and label each node in the array according to the components. If there are k components, initialize an array of size k, where the j^{th} index stores the current cheapest edge coming out of the j^{th} component. Iterate through the edges and update the component array if the edge is a new cheapest candidate extending that component. Once this iteration is done, the k edges stored in the array (or $< k$ if there are duplicates, which can be dealt with by iterating through this array again) are the new edges to be added. Here each operation we just described is of $\mathcal{O}(m)$ so in each iteration of the algorithm, we take $\mathcal{O}(m)$ time.) Therefore the runtime is $\mathcal{O}(m \log n)$. □

Problem 2, Dijkstra on Transportation Networks

Solution. We assume the graph is connected and that a graph-wide solution is guaranteed (after all, KT assumes the same.) Otherwise, extra code for testing edge case will be needed (e.g. no solution due to schedule conflict or no path exists between two vertices). The algorithm goes heuristically as follows. A more detailed implementation is discussed later.

```
// This algorithm is heavily based on the Dijkstra's algorithm provided in KT.
Let S be the set of explored nodes
  for each $u \in S$, we store a distance $d(u)$
Initialize $S \leftarrow \{s\}$ and $d(s) \leftarrow T$
while $S \neq V$
  Select a node $v \notin S$ for which:
    (i) there exists an edge $(u, v, t_u, t_v)$ with $u \in S$ and $d(u) \leqslant t_u$
    (ii) $v$ minimizes $t_v$ among all candidate edges (different quadruplets $(u, v, t_u, t_v)$) satisfying (i)
  Add $v$ to $S$ and define $d(v) \leftarrow t_v$
End
```

Proof of correctness. To prove correctness, we claim the following.

For each $u \in S$, $d(u)$ outputs the earliest arrival time among all conflict-free paths from u to s contained in S.

By setting $S = V$ we see the algorithm provides the earlier arrival time to any vertex. We use induction on $|S|$. The base case $|S| = 1$, i.e., $S = \{s\}$, is trivial.
We now assume that the claim holds for S of size k and we let v be the new vertex chosen by the algorithm. Our goal is to show $d(v)$ cannot be made smaller among all other paths from s to v inside S. By (i), there exists an edge (u, v, t_{u_0}, t_{v_0}) connecting u to v. Let P be the path from s to v consisting of P_u, the shortest path from s to u given by the inductive hypothesis, and (u, v). Suppose P' is an arbitrary path from s to v. Since $s \in S$ and $v \notin S$, P needs to leave S at some point. Let y be the first note after P leaves S. Since the algorithm did not pick

y, we know that

$$t_{v_0} \leqslant \text{arrival time for all edges from inside } S \text{ to } y.$$

Clearly it takes nonnegative time to travel from y to v, so in the end the arbitrary path can arrive no earlier than t_{v_0}. Finally, to check P is a valid path, we simply need to note that $d(u) \leqslant t_{u_0}$ by (i), so we leave u no earlier than we arrive at u. All previous stops are also conflict-free by the inductive hypothesis.

Implementation. To implement the algorithm, we follow the following pseudocode, where we make use of a min-heap to efficiently look for minimal values in a set as well as updating the values.

```
1  Initialize an empty min-heap $Q$ with int or double priority
2  for all vertices $v \in V$ do // initialize the min-heap
3    $d(v) \leftarrow \infty$ if $v \neq s$ else $d(v) \leftarrow T$
4    Add $(v, d(v))$ to $Q$
5
6  Initialize empty $S$ // set of nodes/vertices we have visited so far
7  while $S \neq V$
8    Extract min element $u$ from $Q$
9    for each edge $(u, v, t_u, t_v)$ with starting vertex $u$ do
10     if $d(u) \leqslant t_u$ and $d(v) > t_v$ // valid departure time and new best arrival time
11       $d(v) \leftarrow t_v$ // update $v$'s best arrival time
12   Add $u$ to $S$ // done with $u$
```

Runtime.

- Line 2-4: inserting n nodes into an empty heap takes $\sum_{i=1}^{n} \mathcal{O}(\log n) = \mathcal{O}(\log n!) = \mathcal{O}(n \log n)$ time. This also dominates the $\mathcal{O}(n)$ time taken by line 3.
- Loop from line 7 onward:
  - Line 8: extracting min n times from a heap of size n takes $\mathcal{O}(n \log n)$ time.
  - Line 9: There are a total of $|E| = m$ edges; each will be used at most once. Therefore, the evaluation at line 10 and the revalue reassignment at line 11, both of which can be done in $\mathcal{O}(\log n)$ time using KT's implementation, will take no more than $\mathcal{O}(m \log n)$ time.
- Line 12 will take some time but by no means dominate the previous lines. For example if S is implemented as a set, inserting n elements is $\mathcal{O}(n \log n)$. If we know $|V|$ prior to running the program, we can even replace S simply by a counter and terminate when count $= n$.
- Combining everything (and reasonably assuming $m \geqslant n$), the total runtime is $\mathcal{O}(m \log n)$.