2019 ADA HW 3

b07902064 資工二 蔡銘軒

December 19, 2019

Problem D

(1)

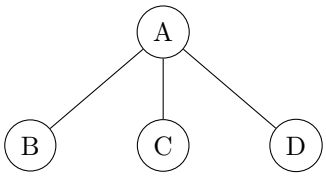

Figure 1: The traversal order is A-B-C-D for both DFS and BFS

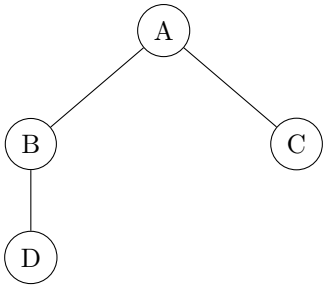

Figure 2: The traversal order is A-B-D-C for DFS, and A-B-C-D for BFS

(2) No, the two trees are not the same. Consider the following graph:

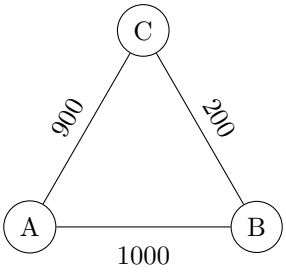

Figure 3: The orignal graph

We compare the minimum spanning tree and the shortest path tree rooted at node A, we see that they are different:

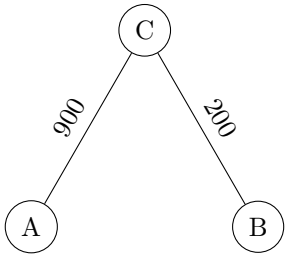

Figure 4: MST for the graph

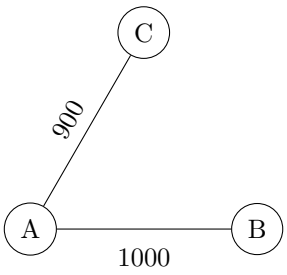

Figure 5: SPT rooted at A

(3)

(a) The steps of the algorithm are as follows:

- Read in the input of cities and roads
- Construct a new graph G by replacing $r(u,v)$ with a new function $r'(u,v) = r(u,v) + c(v)$ for every $(u,v) \in E$, and ignore the function c in G.
- Perform topological sort on the new graph G.
- Initialize an array dis. Let the capital be v', we set $dis[v'] = c(v')$ and $dis[u] = \infty$ for $u \neq v'$
- From v', follow the topological order of the vertices and check every edge $(u,v) \in E$. If $dis[u] + r'(u,v) < dis[v]$, update $dis[u]$.
- Output the answer for each city other than the capital.

(b) The analysis of the time complexity of the algorithm is as follows:

- Read in the input $\Longrightarrow O(V+E)$
- Relabel the length of the roads to be $r'(u,v) = r(u,v) + c(v) \Longrightarrow O(E)$
- Run topological sort on the new graph $G \Longrightarrow O(V+E)$
- Initialize the dis array $\Longrightarrow O(V)$
- Update the distance to each vertex $\Longrightarrow O(V+E)$
- Output the final answers $\Longrightarrow O(V)$

So the overall time complexity is $O(V+E)$.
As for the correctness, first we note that, if we take the road (u,v), we will definitely end up in city v and spend $c(v)$ days in the city. We can equivalently define $r'(u,v) = r(u,v) + c(v)$ for every road, and ignore the c function thereafter. The problem has then become a single-sourced shortest path problem. And since there is no cycle in the graph, we can perform topological sort and update the distance to each vertex following the topological order. The update is based on DP, and the properties are as follows:
Base case: Let the capital be v', we have $dis[v'] = c(v')$, as we will spend $c(v')$ days in the captical at the beginning.
Overlapping subproblem: For every node $v \in V$, $dis[v]$ is an overlapping subproblem.
Optimal substucture: For every node $v \in V$, let A be the set of v's parents. Since we our update is based on topological order, it is guaranteed we have updated every $u \in A$. Suppose $dis[u]$ is optimal for $u \in A$, and since v must be reached from a vertex $u \in A$, it follows that $dis[v] = \min_{u \in A}(dis[u] + r'(u,v))$ is optimal.

Problem E

(1) Since the graph is undirected, it is straightforward that the shortest path from vertex u to vertex v is the same as the shortest path from vertex v to vertex u. So the shortest path to and from each factory is simply twice the shortest distance from the source to the factory. We apply Dijkstra's algorithm:

```
Dijkstra():
   Read input
   Create dis[N] to store the shortest distance to each vertex
   Initialize a priority queue pq that extracts the minimum element
   dis[0] = 0 \\the distance to source is initialized to 0
   for i from 1 to N - 1:
      dis[i] = INF \\distance to other vertices are initialized to INF
   Push pair(0, 0) into pq \\the first element of the pair is distance, the second is the index
       of vertex
   while pq is not empty:
      extract the pair P with smallest distance in pq and pop it
      current_dis = P.first
      current_vertex = P.second
      if current_dis != dis[current_vertex]:
         continue \\the information in pq is outdated, we skip it
      for every neighbor x of current_vertex:
         if dis[current_vertex] + cost(current_vertex, x) < dis[x]:
            dis[x] = dis[current_vertex] + cost(current_vertex, x) \\update distance
            push (dis[x], x) into pq
Output():
   for i from 1 to N - 1:
      print(2 * dis[i]) \\print answer for each vertex
```

(2) The shortest distance from the source to any factory can be obtained directly from the above algorithm. As for the shortest path from each factory back to the source, we reverse the direction of each edge and run the above algorithm on the new graph.
Let the original graph be G, and the new graph with all edges reversed be G'. For a factory f, consider the shortest path from f to the source s in G. Suppose the path is: $t \to v_1 \to v_2 \to \cdots \to v_m \to s$, then in G', we will also have the path $s \to v_m \to \cdots \to v_2 \to v_1 \to t$ and this path will be discovered by Dijkstra's algorithm on G', so the shortest distance to and from f is the sum of $dis[f]$ in both G and G'.
For the time complexity, since we speed up Dijkstra's algorithm with a priority queue, we have:

- Run Dijkstra's algorithm on $G \Longrightarrow O(ElogE)$
- Reverse all edges to derive $G' \Longrightarrow O(E)$
- Run Dijkstra's algorithm on $G' \Longrightarrow O(ElogE)$

So the overall time complexity is $O(ElogE)$.

(3) When there are negative edges in a graph, Dijkstra's algorithm is no longer applicable. Instead, we apply Bellman-Ford algorithm to find out the answer.

```
Bellman(G, source, dis[]):
   set all entries in dis[] to INF except dis[source] = 0
   update = false
   for i from 0 to n - 1:
      update = false
      for each edge (u, v):
         if dis[u] != INF and dis[u] + cost(u, v) < dis[v]:
            dis[v] = dis[u] + cost(u, v)
            update = true
   return update
main():
   read input and initialize dis1[] and dis2[]
   if Bellman(G, source, dis1) == true:
      print("I am rich!") and return
   else:
      reverse all edges in G to derive G'
      Bellman(G', source, dis2)
      output dis1[i] + dis2[i] for every node i
```

For the time complextiy, it is clear in the pseudo code that the nested loops run in $O(N \cdot E)$. Reversing the edges takes $O(E)$, so the overall time complexity is $O(N \cdot E) + O(E) + O(N \cdot E) = O(N \cdot E)$.
As for the correctness, we have discussed the effect of reversing the edges, and since any shortest path cannot contain more then $n - 1$ edges, the relaxation should be completed within $n - 1$ iterations in Bellman-Ford algorithm. If in the *n-th* iteration, relaxation still happens, it implies the existence of a negative cycle.

(4) Let V be the number of vertices, and E be the number of edges.
Time complexity: Dijkstra's algorithm runs in $O(ElogV)$ with a priority queue, and Bellman-Ford algorithm runs in $O(V \cdot E)$.
Space complexity: We use an adjacency list, a priority queue, and an array dis in Dijkstra's algorithm, the space complexity is $O(V + E)$. In Bellman-Ford algorithm, we only need to store the information of the edges, which cost $O(E)$. If we take account of the array dis, then the space complexity is also $O(V + E)$.
Advantages and disadvantages: For Dijkstra's algorithm, with the help of a priority queue or other data structure, it typically runs faster than Bellman-Ford algorithm. However, its use is limited to the graphs without negative weighted edges. For Bellman-Ford algorithm, although it's a slower algorithm compared to Dijkstra's algorithm, it works well on general graphs. It can even find the vertices that form a negative cycle.

(5) Let $c(v)$ be the value on vertex v, and $r(u, v)$ be the value on the edge (u, v). The algorithm works as follows:

- For each edge (u, v), we define $r'(u, v)$ as $K \cdot r(u, v) - c(u)$, and ignore the original $r(u, v)$ and $c(v)$.
- Run Bellman-Ford algorithm on the new graph and check whether there exists a negative cycle.

For the time complexity, relabeling the edges takes $O(E)$, and running Bellman-Ford algorithm takes $O(N \cdot E)$, so the overall time complexity is $O(N \cdot E)$.
As for the correctness, consider a trustful cycle with n vertices. Suppose the vertices are $v_1, v_2, \ldots, v_n$ in clockwise order, and the edges are $(v_1, v_2), (v_2, v_3), \ldots, (v_n, v_1)$. Then we have:

$$\begin{aligned} K &< \frac{c(v_1) + c(v_2) + \cdots + c(v_n)}{r(v_1, v_2) + r(v_2, v_3) + \cdots + r(v_n, v_1)} \\ \implies & (K \cdot r(v_1, v_2) - c(v_1)) + (K \cdot r(v_2, v_3) - c(v_2)) + \cdots + (K \cdot r(v_n, v_1) - c(v_n)) < 0 \\ \implies & r'(v_1, v_2) + r'(v_2, v_3) + \cdots + r'(v_n, v_1) < 0 \end{aligned}$$

So after relabeling the edges, the problem has become finding a negative cycle on the graph, where Bellman-Ford algorithm is applicable.

Problem F

Suppose there are n variables $x_1, x_2, \ldots, x_n$, we construct a graph G with $n+1$ vertices. The vertex set V and edge set E are defined as:

- V: The vertices are numbered from 0 to n.
- E: For every equation $x_i + x_{i+1} + \cdots + x_j = c$ in the pool, there is an edge with weight c between vertex $i-1$ and vertex j.

The problem can now be reduced to finding the MST of G. We first show that any spanning tree of G represents a set of equation that can solve the system:
First, to solve a system of n variables, we need at least n equations. In a spanning tree of G, which has $n+1$ vertices, there are exactly n edges. As each edge represent an equation, the number of edges meets the requirement.
Next we show any spanning tree represents a non-singular system. Consider two edges that represent the equations $x_i + x_{i+1} + \cdots + x_j = c$ and $x_k + x_{k+1} + \cdots + x_j = d$. In graph G, it is visualized as vertex j connected with vertices $i-1$ and $k-1$. WLOG, assume $k > i$, then the connection between $i-1$ and $k-1$ is equivalent to the equation $x_i + x_{i+1} + \cdots + x_{k-1} = c - d$, which can be derived from the previous two equations.

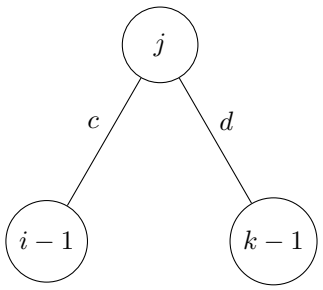

Figure 6: The two equations

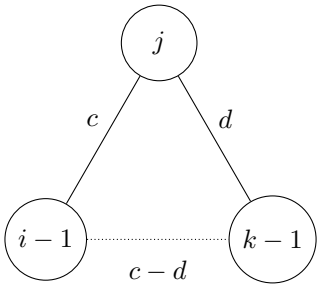

Figure 7: The underlying third equation

Thus in a spanning tree, every two different edges can yield a third equation, and we can use the n edges to span the whole equation pool, which guarantees the solution to the n variables. This also shows that the equations must form a spanning tree. If the n edges form a cycle, there must be an edge that can be "yielded" by other edges and thus is redundant. Therefore the n edges should not form any cycle, which implies a tree structure.
Finally, our goal is to find the minimum spanning tree, which is the answer to the problem.
As for the time complexity, for a system of n variables, there are $n+1$ vertices and $C_2^{n+1} = \frac{n^2+n}{2}$ edges in the corresponding G, we can apply Prim's algorithm, which finds a MST in $O(n^2)$, to solve the problem.

Reference

- Problem D: B07902055 謝宗晅
- Problem E: B07902132 陳威翰 B07902133 彭道耘
- Problem F: B07902132 陳威翰 B07902133 彭道耘