CS3510-A: Design and Analysis of Algorithms, Spring 2021

Homework-7

March 2, 2021

DUE DATE: Tuesday, March 9, 11:59pm

Note-1: Your homework solutions should be electronically formatted as a single PDF document that you will upload on Gradescope. If you have to include some handwritten parts, please make sure that they are very clearly written and that you include them as high resolution images.
Note-2: Please think twice before you copy a solution from another student or resource (book, web site, etc). It is not worth the risk and embarrassment.
Note-3: You need to **explain/justify** your answers. Do not expect full credit if you just state the correct answer. This includes their correctness and runtime.
Note-4: You will get 2 extra points if you submit electronically typed solutions instead of hand-written.

Problem-1 (30 points)

We are given a directed graph with positive link costs. Design an algorithm that computes the minimum-cost cycle in the graph. If the graph is acyclic, the algorithm should report 0. The algorithm should be $O(n\,X)$, where $O(X)$ is Dijkstra's shortest path algorithm runtime complexity and n is the number of vertices in the graph. **Show the runtime in terms of m and n** (do not use X; we use it so as to not give away the runtime complexity).

Note: You must show your algorithm's running time in terms of n and m.

Solution:

Algorithm 1 Shortest Paths with Label Jumps (modified Dijkstra's algo)

$m \leftarrow Undefined$ (minimum-cost cycle)
$c_m \leftarrow -\infty$ (cost of minimum-cost cycle)
for each vertex v in the graph **do**
  $c \leftarrow$ Dijkstras$(graph, v, v)$
  $c_c \leftarrow$ cost of cycle c
  if c is a valid cycle and $c_c < c_m$ **then**
    $m \leftarrow c$
    $c_m \leftarrow c_c$
  end if
end for

if m defined **then**
  return: m
else
  return: 0
end if

Explanation:

By running Dijkstra's path finding algorithm with a single node as both the source and the target, the output will be the smallest possible cycle in the graph that circles back to the node. By keeping track of the smallest cycle and then trying every node in the graph, the algorithm will find the smallest cycle that exists in the graph by eventually iterating to a node in that cycle as the start node.

Runtime:

Dijkstra's algorithm (with priority queue) has a runtime of $O(mlog(n))$. This step is done n times (in the for loop) therefore, the whole algorithm has a runtime of $O(nmlog(n))$.

Problem-2 (35 points)

Suppose you're a consultant for the networking company CluNet, and they have the following problem. The network that they're currently working on is modeled by a connected graph $G = (V, E)$ with n nodes. Each edge e is a fiber-optic cable that is owned by one of two companies – creatively named X and Y – and leased to CluNet.

Their plan is to choose a spanning tree T of G and upgrade the links corresponding to the edges of T. Their business relations people have already concluded an agreement with companies X and Y stipulating a number k so that in the tree T that is chosen, k of the edges will be owned by X and $n - k - 1$ of the edges will be owned by Y.

CluNet management now faces the following problem. It is not at all clear to them whether there even *exists* a spanning tree T meeting these conditions, or how to find one if it exists. So this is the problem they put to you: Give a polynomial-time algorithm that takes G, with each edge labeled X and Y, and either (i) returns a spanning tree with exactly k edges labeled X, or (ii) reports correctly that no such tree exists.

Solution

Algorithm 2 Spanning tree with k edges for company X

```
1: x ← 0
2: t ← run Prim's algo (treat Y edges as weight 0 and X edges as 1)
3: increment x during Prim's when an X edge is added to the tree
4:
5: if x > k then
6:    return: no possible tree
7: end if
8: if x = k then
9:    return: t (this tree has k X edges)
10: end if
11: if x < k then
12:    x ← 0
13:    t ← Empty
14:    t ← run Prim's algo until x = k (treat Y weights as 1; X as 0)
15:    if t is includes all nodes then
16:       return t
17:    else
18:       t ← continue Prim's algo until t includes all nodes (treat Y weight as 0; X as 1)
19:       if x = k then
20:          return: t
21:       else
22:          return: no possible tree
23:       end if
24:    end if
25: end if
```

Explanation:

By weighing X edges as 1 and Y edges as 0, the algorithm will only consider an X edge if it is absolutely necessary in order to add a node to complete the tree. By this process if there are more than k edges in the tree created by running Prim's algorithm the first time then there is no possible spanning three with k total X edges. If there are less than k X edges from this first run of Prim's then the algorithm tries again, now flipping the weights of the edges until there are k total X edges. At this point the weights should flip again so that Prim's will complete the tree with a preference for Y edges. If when this final iteration of Prim's finishes there are not still k total X edges, then it is because the algorithm go to a point where there was no option but to add another X edge to include a node to complete the tree. In this case it is not possible to build a spanning tree with k total X edges, otherwise the algorithm should return the tree.

Runtime:

Because there are only edges with weights 0 or 1, Prim's can be modified to more quickly sort the edges by weight. For example a linked list with a tail pointer can be built to sort the edges in $O(m)$ time before starting. While running each iteration of Prim's the algorithm can simply iterate over the edges in the linked list, adding edges with a node that is not yet included in the tree and ignoring edges that connect two nodes already in the tree (this can be done in $O(m)$ time). If the second iteration of Prim's is needed, then Prim's can be run the same way, but iterating over the linked list backwards (in $O(m)$ time with no need to alter or rebuild the list). At the point in line 18 in the algorithm, when weights should be swapped the algorithm can simply move to the front of the list, iterate forward, and be sure not to pass the point reached while iterating backwards. Nodes that are already included in the tree can be kept track of with a boolean array, thus checking if a node on an edge has been included in the tree yet can be done in $O(1)$ time and the array can be built in $O(n)$ time. All these steps, except building the array only include work that can be done in $O(m)$ and are independent of eachother. Thus the runtime would be $O(n) + 2O(m)$, which would simplify to $O(n + m)$. However for dense nets this could be treated as $O(m)$.

Problem-3 (35 points)

Suppose we are given a connected graph $G = (V, E)$ with $|V| = n, |E| = m$. Each edge $e \in E$ has a positive weight $w(e) > 0$. Each vertex $v \in V$ has a label $a_v \in \{1, 2, \ldots, k\}$. Each label $i \in \{1, 2, \ldots, k\}$, has a cost c_i. At each vertex $u \in V$, you can perform two types of operations:

1. You can travel from u to some vertex v in its neighborhood, i.e. any v such that $e = (u, v) \in E$. The cost of this operation is $w(e)$.

2. You can travel from u to some vertex v with the same label, i.e. any v such that $a_u = a_v$. The cost of this operation is c_{a_u}.

Given a source vertex s and a sink vertex t, design a $O(X)$ algorithm that computes the minimal cost of traveling from s to t where $O(X)$ is Dijkstra's shortest path algorithm runtime complexity on a graph with n vertices and m edges.

Solution:

Algorithm 3 Shortest Paths with Label Jumps (modified Dijkstra's algo)

```
1:  Q ← empty set
2:  for v in graph do
3:      dist[v] ← ∞
4:      prev[v] ← Undefined
5:      Q.add(v)
6:  end for
7:  dist[source] ← 0
8:
9:  while Q not empty do
10:     u ← vertex in Q with min dist[u]
11:     Q.remove(u)
12:
13:     s ← set with each vertex v where v is a neighbor of u and v is in Q
14:     add each vertex v where a_u = a_v and v is in Q to s
15:     for each vertex v in s do
16:         if a_u = a_v then
17:             alt ← dist[u] + c_{a_u}
18:         else
19:             alt ← dist[u] + weight(u, v)
20:         end if
21:
22:         if alt < dist[v] then
23:             dist[v] ← alt
24:             prev[v] ← u
25:         end if
26:     end for
27: end while
28: return: dist[target]
```

Explanation:

The algorithm is a modified version of a Dijkstra's traversal algorithm. Instead of only considering paths by connections the algorithm also considers paths by taking "jumps" between vertexes with the same label. These "jumps" can be though of as another connection between two vertexes, where the cost of the weight of the connection is simply the cost associated with the label. After completing the main while loop, the algorithm will have identified the shortest possible cost to get from the source to the target node. By repeatedly updating the distance to all nodes to be the smallest possible, by the end of the loop the value will be the smallest possible cost to reach the target.

Runtime:

If *dist* is tracked using a priority queue, then updating values for distance to a node can be done in $O(log(n))$. This will at most be done for each edge (m times), thus the runtime for this opertaion is $O(mlog(n))$.

Additionally, removing a vertex from the queue can be done in $O(log(n))$. This operation will be done a total of n times because vertexes are removed from Q until Q is empty, thus the runtime for this operation is $O(nlog(n)))$.

The combined runtimes will be $O(mlog(n) + nlog(n))$ which is the same as $O((m + n)log(n))$. This is the same runtime as a normal Dijktra's implementation (with a priority queue).