CS 101 Fall 2024 - Quiz 10
December 9, 2024 - 20 Minutes **Name:** **Student ID:**

---

1. (1 points) Honor Code

I promise that I will complete this quiz independently and will not use any electronic products or paper-based materials during the quiz, nor will I communicate with other students during this quiz.

I will not violate the Honor Code during this quiz.

√ True ◯ False

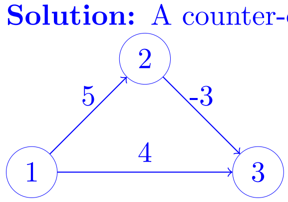
2. (9 points) True or False

Determine whether the following statements are true or false.

(a) (1') The shortest path in DAG can be computed in $O(|V| + |E|)$ via a modification method of topological sort. However, Dijkstra's algorithm may fail to DAG with negative-weighted edges.

√ True ◯ False

Solution: A counter-example:



(b) (1') Given a directed graph G with no negative-weight edges, and a shortest path P from node s to node t if we negate the weight of one edge in the path P (i.e, multiply it by -1), Dijkstra's algorithm can still find the correct shortest path from s to t. ◯ True √ False

Solution: True if there isn't any negative cycle after we negate the edge. Otherwise, the original path is no longer the shortest path.

(c) (1') Given a directed graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$, and G has no negative cycle . In Bellman-Ford's algorithm, after k out-most iterations, the shortest path from v_1 to v_n that consists of at most k edges is computed.

√ True ◯ False

(d) (1') After applying Bellman-Ford's algorithm on node v, if there are no negative cycles, we have the minimum distance between any two different nodes v_i and v_j. ◯ True √ False

(e) (2') On a graph with n vertices and m edges, if all edges have positive weights, Bellman-Ford's algorithm uses $O(mn)$ iterations to find the shortest distance path of a single source.

√ True ◯ False

(f) (1') In any connected graph without a negative cycle, A* tree-search algorithm with consistent Heuristics can always find the shortest path between two nodes.

√ True ◯ False

(g) (1') A* Graph Search algorithm returns the optimal shortest path if the heuristic function is admissible. ◯ True √ False

Solution: The heuristic function should be consistent.

(h) (1') In A* graph search algorithm with a consistent heuristic function, if vertex u is marked visited before v, then $d(u) + h(u) \leq d(v) + h(v)$, where $d(u)$ is the distance from the start vertex to u.

√ True ◯ False

3. (8 points) Lets code!

We want to find a single source min distance with Dijkstra's Algorithm and A* **graph search** algorithm. **Suppose all edges have positive weight and the heuristic function is consistent**. The graph mentioned in this problem is a simple directed graph.

Note: 'w' is a weight map, where you can get any edge (u, v)'s weight by using 'w(u, v)'. 'h' is a consistent heuristic function, you can get the heuristic value of a node u by using 'h(u)'.

dist[i] represents the shortest distance from s to i, pre[i] represents the previous node on the shortest path from s to i.

Q is a min-heap storing a tuple: (key, value), and sorted by value. And you have the following operations for Q:

1. **Q.push({u, val})**: put a tuple (u, val) into the heap.
2. **{u, val} = Q.pop()**: get the tuple with minimum value in the heap, and then pop the tuple out of the heap.
3. **Q.update({u, val})**: find the tuple in the heap whose key is 'u', and update its value into 'val'.

Algorithm 1 Single Source Shortest Path Algorithm

Input: Weight map w, min-heap Q, Source node s, heuristic function h .
Output: The shortest distance from s to all other nodes, and their previous node in the shortest path.
for $i \leftarrow 0$ **to** V **do**
    dist[i] $\leftarrow$ Inf
    pre[i] $\leftarrow$ NULL
    Q.push({i, dist[i]})
end for
dist[s] $\leftarrow 0$
Q.update({s, 0})
while Q is not empty **do**
    Fill this part with your pseudo code
    ...
end while
return dist, pre

What you need to do is to write some **pseudo code** to fill in to implement the algorithms with the given operations.

(a) (4') Implement Dijkstra's algorithm.

Solution:

```
{u, _} = Q.pop()
for each neighbor v of u:
    if  dist[v] > dist[u] + w(u, v):
        dist[v] = dist[u] + w(u, v);
```

```
            pre[v] = u;
            Q.update({v, dist[v]})
```

(b) (4') Implement A* **graph search** algorithm. You can use $h(v)$ to get the heuristic value for any node v.

Hint: The algorithm should not differ too much from Dijkstra's algorithm.

Solution:

```
{u, _} = Q.pop()
for each neighbor v of u:
    if  dist[v] > dist[u] + w(u, v):
        dist[v] = dist[u] + w(u, v);
        pre[v] = u;
        Q.update({v, dist[v]+h(v)})
```