Songhao Li April 16, 2021

6.006 Problem Set

Collaborators: *Problem 1: Yukai Tan; Problem 2: Yukai Tan*

Problem 1

(1)
DAG relaxation:
(a,b),(a,c),(c,b),(c,e),(e,f),(b,d),(d,g)
Dijkstra:
(a,b),(a,c),(c,b),(c,e),(b,d),(b,f),(e,f),(d,g)

(2)
$\delta(a,c) = 1$,$\delta(a,b) = 2$,$\delta(a,e) = 3$
$\delta(a,d) = 6$,$\delta(a,f) = 5$,$\delta(a,g) = 9$

Problem 2

(1)Ben is wrong. Think about $w_{a,b} = -9, w_{b,c} = -9, w_{a,c} = -10$. The original shortest path from a to c is $a \to b \to c$, but after adding 10 to each edge weight, $a \to c$ has distance of 0 but $a \to b \to c$ has distance of 2

(2) Alyssa is correct.

We can decompose a shortest path to a sequence of "sub-path", where each sub-path contains at most one negative weight edge. There are at most 101 sub-paths for a shortest path from the starting vertex to destination vertex. The maximum happens when you start at starting vertex, record a sub-path every time you finish a negative edge; after 100 negative edges you are left with additional non-negative edges to destination vertex as the last sub-path.

Essentially, every run of Dijkstra will update exactly one sub-path, and "correct" distances all the way till finishing a negative edge (proof of below). As a result, since there are at most 101 sub paths, 101 Dijkstra runs will find all the shortest paths.

The reason that one Dijkstra can update one sub-path is as follows. By the nature of Dijkstra algorithm, we can explore and relax all the non-negative edges along a sub-path; and when the last non-negative edge is explored, we will go to the vertex on the sub-path with negative out-bounding edge. When this vertex is popped out of the priority queue, all its out-bounding edges will be relaxed including the negative edge; so the vertex(v) on the sub-path with in-bounding negative edge has the right distance. Essentially we have finished the sub-path with v as the destination. The next time we run Dijkstra algorithm, we will finish the sub-path with v as the starting point.

Problem 3

build a graph that has kn vertices:

$V = V_{i,j}$ where i from 1 to n and j from 1 to k

$V_{a,j}$ and $V_{b,j}$ has edge with weight d if and only if there is a path from a to b with weight d, and b has no demon commandar

$V_{a,j}$ and $V_{b,j+1}$ has edge with weight d if and only if there is a path from a to b with weight d, and b has a demon commandar

the number of edges are bounded by km. Because a city has k corresbonding vertices, and a road is "represented" at most k times by the outbounding edges of its vertices.

if Aquilegia has no demon commandar, we want to know the shortest distance of $V_{Iris,k}$. If Aquilegia has a demon commandar, we want to know the shortest distance of $V_{Iris,k-1}$. Either case we can run dijstra on such graph.

Since the number of vertices is bounded by kn and the number of edges is bounded by km, the runtime is $O(km + knlog(kn))$

(2) If there is a path and a loop, then its done

Run a modified version of BF. Now we want to find the "most profitable path" and the values we iterating over time are the maximum profit for all paths from A to the corresponding city.

initialze a list for profit p, where p[x] = 0 every city x. Initilize Reachable = false

```
for k in range(n-1): do
  for a in range(n): do
    for b in a.adjacents: do
      if Reachable == false and b == Iris: reachable = True then
        $p[b] = max(p[b], p[a] + p_{ab} - c_b)$
      end if
    end for
  end for
end for
if reachable == false then
  return false (there is no path from A to Iris)
end if
if p[Iris] >= 500 then
  return true (you already find a path)
end if
for a in range(n) do
  for b in a.adjacents do
    if p[a] + $p_{ab} - c_b > p[b]$ then
      return true(you find a positive loop that you can go infinitely to accumulate gold pieces)
    end if
end for
```

Problem 4

Your solution to problem 2 goes here.

1) s1= 0, s2 = -4, s3 = -15, s4 = -7

2) For the four constraints, the sum of left hand side is 0 but the sum of right hand side is -2, which leads to $0 < -2$, so the four constraints can not hold all together

3) Use a node to represet an assignment. If there is constraint $s_i - s_j <= w_{ji}$, construct a edge from j to i with weight w_{ji}.

4) Using the representation in part c, we run BF algorithm to either find "shortest distances" of all current nodes relative to node 1, which is also the earliest start time of assignment i relative to assignment 1. As we can observe, $s_i - s_j <= w_{ji}$ has the same structure as updating ("relaxing") the distance from s_j to origin in a graph. So during iteration, instead of updating "best estimate of shortest distance", we are updating the best estimate of earliest start time possible for assignment i.

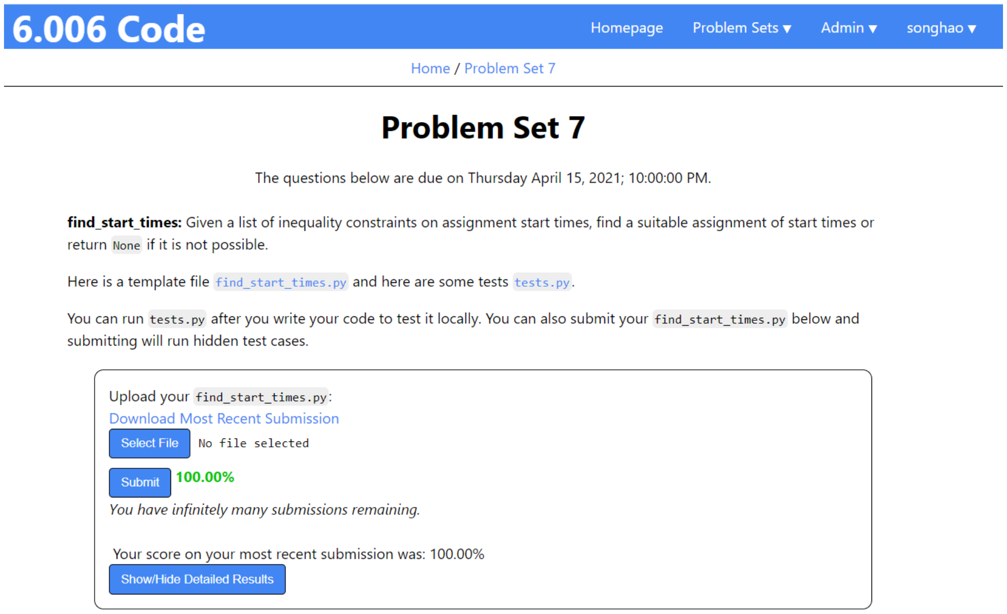
Correctness:

If the BF algorithm finds an negative loop, then follow the same logic in (2), adding both the LHS and RHS we can not satisfy the restrictions. On the other direction, if there is a set of restrictions like (2), their is essentially a negative loop in the graph.

If the BF algorithm run successfully, then we have a choice of starting times since every updates satisfies the restrictions, and there does not exist a negative loop, so the correctness of the algorithm is proven.

Runtime:

The runtime is essentialy the runtime of BF algorithm which is O(nm)



Problem Set 7

The questions below are due on Thursday April 15, 2021; 10:00:00 PM.

find_start_times: Given a list of inequality constraints on assignment start times, find a suitable assignment of start times or return `None` if it is not possible.

Here is a template file `find_start_times.py` and here are some tests `tests.py`.

You can run `tests.py` after you write your code to test it locally. You can also submit your `find_start_times.py` below and submitting will run hidden test cases.

Upload your `find_start_times.py`:
Download Most Recent Submission
No file selected
100.00%
You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%