Kev Sharma
kks107

I have not discussed this exam with anyone except the professor and the TAs of CS 344. I have not used any online resources during the exam except for those accessible from the Canvas website.

1. (a) Part 1

Table: row 1 is the order in which the vertices are explored. row 2 is d(v_i) corresponding to when that vertex v_i is explored.

	1	2	3	4	5	6	7
row 1	s	a	e	d	c	f	b
row 2	0	3	4	7	8	9	14

(b) Part 2

- topological ordering 1: f, a, d, b, c, g, e
- topological ordering 2: f, a, d, b, c, e, g

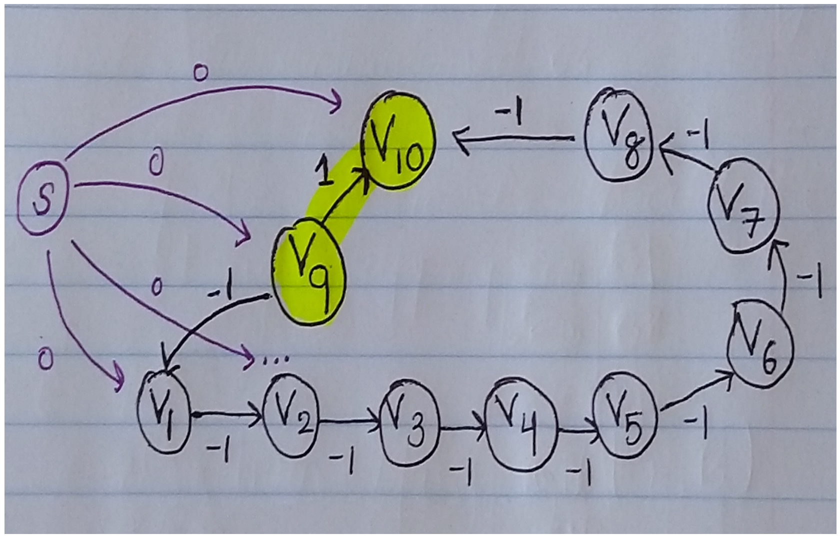
2. (a) Answer: w'(x,y) = w(x,y)

    - Recall that w'(x,y) = w(x,y) + ϕ(x) - ϕ(y).
    - In our Johnson's algorithm, we create a pseudo source vertex s and draw an edge from s to each vertex v $\in$ V.
    - Note here that if all edge weights are non-negative, Bellman-Ford computes the distance from s to each v to be exactly 0.
    - That is, $dist_{G'}$(s,v) for all v $\in$ V equals 0.

    - Justification:
        - $dist_{G'}$(s,v) cannot be greater than 0 since we know of the path from s to v which we pseudo inserted, to be of distance 0.
        - $dist_{G'}$(s,v) cannot be less than 0 since that would violate our assumption that all edge weights are non-negative.

    - $dist_{G'}$(s,v) = 0 for all v $\in$ V, $\therefore$ ϕ(v) = 0 for all v $\in$ V (Johnson's Algorithm).
    - Hence each w'(x,y) computation reduces to w'(x,y) = w(x,y) + 0 - 0.
    - As such, each w'(x,y) equals w(x,y).

(b) Answer: The largest possible edge weight in G' is exactly 10.

i.
- Take the example graph G, where the pseudo source vertex s in written in purple and vertices x, y, and w(x,y) are highlighted in yellow.
- Note that v9 = x and v10 = y.
- Note that w(v9, v10) is set to 1:
- 

ii. In order to maximize w'(x,y), we must maximize [w(x,y)] and [-ϕ(y)]:

A. We should set w(x,y) to its maximum edge weight value of 1

B. We should pick a destination vertex y, such that $dist_{G'}$(s,y) is the most negative distance of a path from s to another vertex in G'.

iii. Given the graph G', Bellman-Ford gives us:

A. $dist_{G'}$(s, v10) comes out to -($|V|$ - 1) = -9 by walking the path from v9 to v1 to ... to v8 to v10.

B. $dist_{G'}$(s, v9) comes out to 0.

iv. Computing such a maximum w'(x,y) we have:

A. w'(x,y) = w(v9, v10) + ϕ(v9) - ϕ(v10)

B. w'(x,y) = 1 + $dist_{G'}$(s, v9) - $dist_{G'}$(s, v10)

C. w'(x,y) = 1 + 0 - (-9)

D. w'(x,y) = 10

3. (a) T[i][j] equals the number of red colored items our subset uses to sum to j.

   (b) DP relation: T[i][j] = min(T[i-1][j] , T[i-1][j-S[i]] + isRed).

   Note: isRed is valued at 1 if S[i].color is red and 0 if S[i].color is blue. If we use an existing subset which sums to j-S[i] and add S[i] to derive the minimum reds used to sum to j, we need to account for the color of S[i].

   (c)
   - T[0][0] = 0.
   - T[0][S[0]] = 1 or 0 depending on whether S[0] is red or blue respectively (only set if S[0] $\leq$ B to avoid column overbound).
   - T[0][col] = 101, where col ! = 0, S[0].
   - Row indexing goes from 0 to n-1 inclusive. Column indexing goes from 0 to B inclusive.

   (d) T[n-1][B] stores an integer reflecting the minimum number of reds used to sum to B after considering all items in set S. If this integer exceeds 100, we return false. Otherwise we return true.

   (e)

```
procedure Problem3(S, B)
    n ← |S|
    T ← 2D array [n][B+1]
    ▷ Initialization Step for first row ◁
    for col = 0, ..., B do                                          ▷ O(B)
        T[0][col] ← 101
    T[0][0] ← 0
    if S[0] ≤ B then
        T[0][S[0]] ← S[0].color == red ? 1 : 0
        ▷ Ternary operator: returns 1 if S[0].color is red, 0 if blue. ◁
    ▷ DP-relation: go from second row onward to last row (n-1). Traverse
      each column from 0 to B inclusive. ◁
    for i = 1, ..., n-1 do                                          ▷ O(n)
        isRed ← S[i].color == red ? 1 : 0
        for j = 0, ..., B do                                        ▷ O(B)
            T[i][j] ← min( T[i-1][j] , T[i-1][j-S[i]] + isRed )
            ▷ If a column is computed to be < 0, T[i-1][j-S[i]] returns 101. ◁
    ▷ Return step ◁
    return T[n-1][B] ≤ 100
```

Run-time: O(nB) as a result of two nested for loops.

4. (a) T[i][j] corresponds to the number of paths from s to v_i that use exactly j number of edges. Please also note:
   - T is of dimension n by 101. T's rows are indexed from 1 to n inclusive, and each row's columns are indexed from 0 to 100 inclusive.
   - Row k corresponds to the vertex at index k in the topologically sorted set of vertices V.
   - A (key,value) pair for each (vertex, it's index in V) has been inserted into a Dictionary D to quickly look up indexes for arbitrary vertices (used for in-neighbor look-ups).
   - For example, calling D.search(s) gives the index of s in V.
   - Calling T[D.search(x)][k] gives the number of paths from s to x using exactly k edges.

   (b) DP relation: computing T[i][0] for i!=D.search(s) is unnecessary. Because our inner DP loop starts at j=0, our DP relation stresses T[i][j+1]:

$$T[i][j+1] \leftarrow \sum_{x \in In(v_i)} T[D.search(x)][j]$$

   (c) Initialization Step:
   - T[D.search(s)][0] = 1.
   - T[i][j] = 0 for all j and for all i!=D.search(s).
   - DP relation builds off of the fact that T[D.search(s)][0] = 1.

   (d) We return

$$\sum_{j=0}^{100} T[D.search(t)][j]$$

(e)

```
procedure Problem4(DAG G, s, t)
    TopSort(G)                                                ▷ O(|E|)
    D ← new Dictionary
    ▷ a dictionary to store (vertex, index of said vertex in V after top sorted).
      Helps with constant time lookup of a vertex's index in V. ◁
    index ← 1
    for all v ∈ V do                                          ▷ O(|V|)
        D.add(v, index)
        index ← index + 1
    ▷ All vertices in V are indexed (1 through n) in D. ◁
    if D.search(s) > D.search(t) then
        return 0            ▷ Since graph is DAG, no way to get to t from s
    n ← |V|
    Initialize T[n][101]      ▷ T[1 to n inclusive][0 to 100 inclusive]
    ▷ Initialization step ◁
    for i = 1, . . . , n do                                   ▷ O(|V|)
        for j = 0, . . . , 100 do
            T[i][j] ← 0
    T[D.search(s)][0] ← 1
    ▷ DP relation step ◁
    start ← D.search(s) + 1             ▷ index of vertex after s in V
    finish ← D.search(t)          ▷ go up to and including vertex t in V
    for i = start, . . . , finish do                          ▷ O(|V|)
        for j = 0, . . . , 99 do                              ▷ O(100)
            for all x ∈ In(v_i) do                            ▷ O(In(v_i))
                T[i][j+1] ← T[i][j+1] + T[D.search(x)][j]
    ▷ Return value step ◁
    totalPaths ← 0
    for j = 0, . . . , 100 do                                 ▷ O(100)
        totalPaths ← totalPaths + T[D.search(t)][j]
    return totalPaths
```

Run-time Analysis:

The run-time is O(TopSort + indexing + Initialization + DP + Return).
That is O($|E|$ + $|V|$ + $|V|$ + 100*$\sum_{i=1}^{n} |In(v_i)|$ + 100).

$\sum_{i=1}^{n} |In(v_i)|$ was discussed in class to sum to O($|E|$).

= O(100$|E|$) derived from DP Step. This gives us a run-time of O($|E|$).

5.