Homework A (15 pts)

1. [5 pts] Below is the pseudocode for Johnson’s Algorithm. Show that the running time is O(VE lg V). (You may, but need not, do this by marking the running time of each line [or the number of times a loop is executed] on the left. In any case, you’ll want to explain at the end.) You may assume that the Bellman-Ford algorithm runs in time O(VE) and the Dijkstra/LCFS algorithm runs in time O(E lg V).

*\_O(V)\_* Compute G’ as V[G’] = V[G] ∪ {s} and E[G’] = E[G} ∪ {s,v}, ∀v∈V[G]

*O(VE)* If (Bellman-Ford( G’, w, s ) == false // cycle

*O(1)*  Report negative-weight cycle and exit

*O(1)*  Else

*\_O(V)\_* For each vertex v∈V[G’]

*O(1)*  h(v) = δ(s,v) as computed by Bellman-Ford

*O(E)*  For each edge (u,v) ∈E[G’]

*O(1)*  ŵ(u,v) = w(u,v) + h(u) – h(v)

*O(V)\_* For each vertex u∈V[G]

*O(ElgV)* Run Dijkstra(G, ŵ, u) to get đ(u,v) ∀v∈V[G] // wt under ŵ

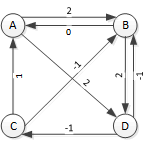
*O(V)\_* For each vertex v∈V[G]

*O(1)\_* δ(u,v) = đ(u,v) + h(v) – h(u)

***Solution:***

The running times of the lines are listed above (for a loop, it lists the number of times the loop is entered). The part of the algorithm that takes the most time is highlighted. For each vertex O(V) you do Dijkstra’s algorithm (O(E lg V)) for O(VE lg V) work.

1. [3 pts] Complete the Floyd-Warshall algorithm for the graph that we did in class. Here are the D0/π0 and D1/π1 matrices. Fill out the D2/π2 and D3/π3 and D4/π4 matrices below. For the purpose of this problem, nodes 1-4 are nodes A-D, respectively.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **D0** | **A** | **B** | **C** | **D** |
| **A** | 0 | 2 | ∞ | 2 |
| **B** | 0 | 0 | ∞ | 2 |
| **C** | 1 | -1 | 0 | ∞ |
| **D** | ∞ | -1 | -1 | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **D1** | **A** | **B** | **C** | **D** |
| **A** | 0 | 2 | ∞ | 2 |
| **B** | 0 | 0 | ∞ | 2 |
| **C** | 1 | -1 | 0 | 3 |
| **D** | ∞ | -1 | -1 | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **D2** | **A** | **B** | **C** | **D** |
| **A** | 0 | 2 | ∞ | 2 |
| **B** | 0 | 0 | ∞ | 2 |
| **C** | -1 | -1 | 0 | 1 |
| **D** | -1 | -1 | -1 | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **D3** | **A** | **B** | **C** | **D** |
| **A** | 0 | 2 | ∞ | 2 |
| **B** | 0 | 0 | ∞ | 2 |
| **C** | -1 | -1 | 0 | 1 |
| **D** | -2 | -2 | -1 | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **D4** | **A** | **B** | **C** | **D** |
| **A** | 0 | 1 | 1 | 2 |
| **B** | 0 | 0 | 1 | 2 |
| **C** | -1 | -1 | 0 | 1 |
| **D** | -2 | -2 | -1 | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **π0** | **A** | **B** | **C** | **D** |
| **A** | ∅ | A | ∅ | A |
| **B** | B | ∅ | ∅ | B |
| **C** | C | C | ∅ | ∅ |
| **D** | ∅ | D | D | ∅ |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **π1** | **A** | **B** | **C** | **D** |
| **A** | ∅ | A | ∅ | A |
| **B** | B | ∅ | ∅ | B |
| **C** | C | C | ∅ | A |
| **D** | ∅ | D | D | ∅ |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **π2** | **A** | **B** | **C** | **D** |
| **A** | ∅ | A | ∅ | A |
| **B** | B | ∅ | ∅ | B |
| **C** | B | C | ∅ | B |
| **D** | B | D | D | ∅ |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **π3** | **A** | **B** | **C** | **D** |
| **A** | ∅ | A | ∅ | A |
| **B** | B | ∅ | ∅ | B |
| **C** | B | C | ∅ | B |
| **D** | B | C | D | ∅ |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **π4** | **A** | **B** | **C** | **D** |
| **A** | ∅ | D | D | A |
| **B** | B | ∅ | D | B |
| **C** | B | C | ∅ | B |
| **D** | B | C | D | ∅ |

1. [5 pts] Rewrite the pseudocode for the Floyd-Warshall algorithm so it includes the pseudocode to calculate the predecessor matrix. Here is the pseudocode without that calculation. (You can call the predecessor matrix Π and call each element πij as in the text and notes, or you can skip the Greek and call the matrix P and each element pij. Your choice.) (Su 18)

***Solution:***

Lines added, deleted, changed as shown below.

Floyd-Warshall(W)

n = W.rows

D(0) = W

Π(0) = ∅ ***// add***

for (k = 1 to n)

for ( i = 1 to n )

for ( j = 1 to n )

dij(k) = min(dij(k-1), dik(k-1) + dkj(k-1) ) ***// del***

if (dij(k-1) ≤ dik(k-1) + dkj(k-1)) ***// add***

dij(k) = dij(k-1) ***// add***

πij(k) = πij(k-1) ***// add***

Else***// add***

dij(k) = dik(k-1) + dkj(k-1) ***// add***

πij(k) = πkj(k-1) ***// add***

return (D(n), Π(n))***// modify***

1. [3 pts] *Dynamic programming review* Seeing as how the Covid-19 pandemic completely upended the traveling habits of the public, suppose that the airlines abandoned their fancy demand pricing models and decided that the cost to fly nonstop from city *cj* to city *ck* is some constant *djk* dollars. But a nonstop flight may not be the cheapest way to get from *cj* to *ck*. Dynamic programming to the rescue.

Write the recurrence equation to find the minCost(j,k).

***Solution:***