Nalongsone Danddank Student ID : 14958950 StarID: jf3893pd

Email: [nalongsone.danddank@my.metrostate.edu](mailto:nalongsone.danddank@my.metrostate.edu)

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).

*C*  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

*C*  Report negative-weight cycle and exit

Else

For each vertex v∈V[G’]

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

For each edge (u,v) ∈E[G’]

*O(E)*  ŵ(u,v) = w(u,v) + h(u) – h(v)

For each vertex u∈V[G]

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

For each vertex v∈V[G]

*O(V)*  δ(u,v) = đ(u,v) + h(v) – h(u)

***Solution:***

We can see the last nest loop we have the for loop of vertex v∈V[G], it run time is O(VE), which come from the running of Dijkstra algorithm, O(E lg V), the outside of this loop.

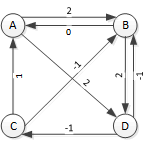
So the runtime of this nest loop is O(E lg V) \* O(V) = O(VE lg V)

Then, The total of the pseudocode should be the sum of all the running time that marking of each line like below:

C + O(VE) + C + O(VE) + O(E) + O(E lg V) \* O(V) => O(VE lg V)

Proof that: the time complexity is O(VE lg V).

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 | 0 | 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** | C | C | D | ∅ |

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

Floyd-Warshall(W)

n = W.rows

D(0) = W

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) )

return (D(n))

Solution:

Floyd-Warshall(W)

n = W.rows

P = new empty matrix same size to W *//initialize P with empty matrix.*

//looping to find out the original of P matrix following the Graph.

for ( i = 1 to n)

for ( j = 1 to n)

If ( i != j and wij is not ∞ ) // check matrix table

Pij = i // put the original of before node to it.

D(0) = W

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) )

If ( dij(k) != dij(k-1))//if the value change

P(k)ij = k //then change to the new node by K, like A, B

return (D(n) , P(n))

1. [4 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:

Recurrence Equation:

minCost(j, k) = min( minCost(j -1, k), minCost(j, k-1)) + djk

Pseudocode :

minCost(j, k)

if (j < 0 || k < 0)

return ∞

else if ( j == 0 && k == 0)

Return djk

else

return djk + min(minCost(j-1, k), minCost(j, k-1))