CS 301 - Assignment 4 - Tourist Problem

Rebah Özkoç
29207

May 19, 2022

Introduction

We need to find the shortest itinerary from a given city and station to each and every city. To achieve this we need to find the shortest path from the given city to every other city separately like a graph problem. In our graph, the nodes are the cities, and the edges are the paths between the cities. Since there are different stations in every city, we need to create two different graphs for every type of station and merge them when needed. Also, since our graph is not a directed acyclic graph we need to limit of our recursion depth by the number of the vertices to prevent infinite loops while solving the problem.

Recursive Definition of The Problem

We can solve this problem by recursion. We can introduce a notation to express the problem in a recursive way. We can define the sub-problems like

$$P_k^b(s, v)$$

where can be read as the shortest way from the city s to city v in at most k vertices by using bus stations and

$$P_k^t(s, v)$$

where can be read as the shortest way from the city s to city v in at most k vertices by using train stations.

We can define the transfer time in a city s with:

$$Transfer(s)$$

So, the subscript of the problem denotes the at most number of edges that we can use while solving this problem, whereas the superscript denotes the transportation way, which can take only two values, namely bus (b) and train (t).

We can denote the problem by using sub-problems with a recursive definition.

- $P_0^t(s, v) = \infty for s \neq v$

- $P_0^b(s, v) = \infty for s \neq v$

- $P_k^t(s, s) = Transfer(s)$ if the starting station for s is bus station, else 0 for every $k = 0, 1, ..., v - 1$

- $P_k^b(s, s) = Transfer(s)$ if the starting station for s is train station, else 0 for every $k = 0, 1, ..., v - 1$

These two parts were the base cases of the problem. Now we can look to the recurring part of the problem.

- $P_k^b(s, v) = min_{((u,v)\in E)}(P_{k-1}^b(s, u) + w^b(u, v),\ P_{k-1}^t(s, u) + w^t(u, v) + Transfer(v))$

- $P_k^t(s, v) = min_{((u,v)\in E)}(P_{k-1}^t(s, u) + w^t(u, v),\ P_{k-1}^b(s, u) + w^b(u, v) + Transfer(v))$

Merging the Two Main Sub-problem

Until this point, we divide the main problem into two sub-problems. $P_k^t(s,v)$ and $P_k^b(s,v)$ where s is the starting city, v is the ending city and k is the maximum number of edges we can use which is equal to number of cities in the graph. After solving these two recursions we need to find the quickest path from the given station in the city v. It can be computed like this:

If the starting station is train $\rightarrow$ Result = min($P_k^t(s,v)$, $P_k^b(s,v) + Transfer(s)$)

If the starting station is bus $\rightarrow$ Result = min($P_k^t(s,v)$ + Transfer(s), $P_k^b(s,v)$)

Analysing the Correctness

Optimal Substructure

The recursive definition of the problem relies on the optimal substructure property. This problem have optimal substructure because an optimal solution can be constructed from optimal solutions of its sub-problems. It can be proven by the cut and paste method.

Assume that $P_k^b(s,v)$ is the quickest path from s to v using bus where

$P_k^b(s,v) = min_{((u,v)\in E)}(P_{k-1}^b(s,u) + w^b(u,v),\ P_{k-1}^t(s,u) + w^t(u,v) + Transfer(v))$ and

$min_{((u,v)\in E)}(P_{k-1}^b(s,u) + w^b(u,v),\ P_{k-1}^t(s,u) + w^t(u,v) + Transfer(v)) = P_{k-1}^b(s,u_1) + w^b(u_1,v)$

Assume that there exists another path to u_1, $P'^b_{k-1}(s,u_1)$ which is shorter than the current path $P_{k-1}^b(s,u_1)$. In this case, we can cut this path and paste it to the original solution and we can get a shorter path. This is a contradiction with the first assumption, hence the algorithm has optimal substructure property. We can extend this proof to the train recursion without loss of generality.

Termination of The Problem

According to the definition of the algorithm, if the k is zero the algorithm returns ∞, zero, or $Transfer(s)$. Also, if the starting and ending nodes of the problem is equal than the algorithm returns $Transfer(s)$ or 0. We need to show that the recurring part of the algorithm converges to one of these conditions and stops. Recurring parts of the problem were like these:

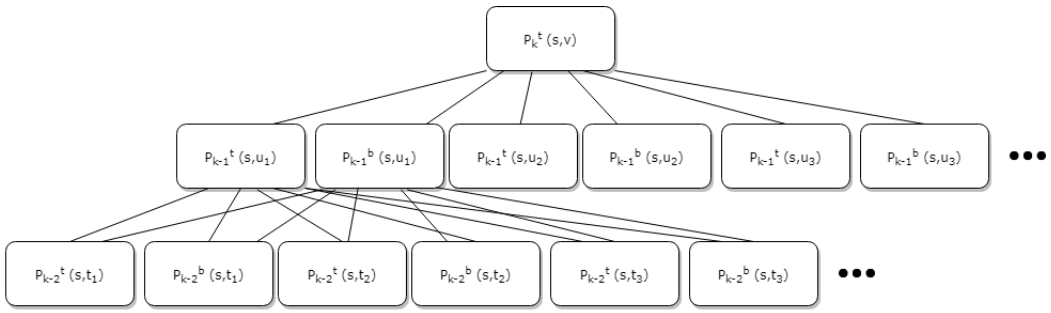
- $P_k^b(s,v) = min_{((u,v)\in E)}(P_{k-1}^b(s,u) + w^b(u,v),\ P_{k-1}^t(s,u) + w^t(u,v) + Transfer(v))$

- $P_k^t(s,v) = min_{((u,v)\in E)}(P_{k-1}^t(s,u) + w^t(u,v),\ P_{k-1}^b(s,u) + w^b(u,v) + Transfer(v))$

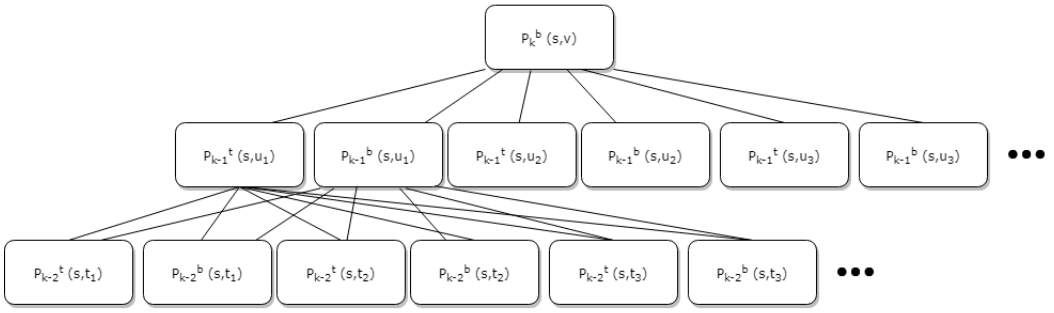
In this recursion, in every step we go back one node previous and decrease the k by 1 for every node that is neighbour to the current node which is finite number of nodes. The recursion stops when the k is equal to zero or when we arrive to the starting node. Hence the algorithm terminates.

Using Dynamic Programming

To solve the problem with dynamic programming we need to show that the recursion has overlapping sub-problems. We can draw the recursion tree to find the overlapping sub-problems.



Train station recursion



Bus station recursion

As can be seen from the graphs, there are overlapping sub-problems between the two main recursion trees and inside of these recursion trees. If we want to reduce the time complexity of the problem we can we solve the each sub-problem at most one time and we can store the results. The problem that we want to solve can be denoted with $P_k^b(s,v)$ and $P_k^t(s,v)$. In this problem k ranges from 1 to $|V|$, s stays same as the starting city and v takes value as the all cities in the graph. Thus the total number of sub-problems is $O(2 * V * V) = O(V^2)$.

Pseudo Code

We can modify and adapt the Bellman Ford algorithm to solve this problem.

```
# Create city graphs
# Create transferTime dictionary
# Create distance dictionaries db (bus), dt (train)

for each v  V do:
db[v] = infinity
dt[v] = infinity

if startType is train:
dt[s] = 0
db[s] = transferTime[s]
else:
dt[s] = transferTime[s]
db[s] = 0

for k = 1 to |V|:
for each edge in (u,v) belongs to E:
if dt[v] > dt[u] + wt[u, v]:
dt[v] = dt[u] + wt[u, v]
if db[v] > db[u] + wb[u, v]:
db[v] = db[u] + wb[u, v]
if dt[v] > db[u] + wb[u, v] + transferTime[v]:
dt[v] = db[u] + wb[u, v] + transferTime[v]
if db[v] > dt[u] + wt[u, v] + transferTime[v]:
db[v] = dt[u] + wt[u, v] + transferTime[v]
```

```
# Check if any negative circle exist

for each edge in (u,v) belongs to E:
if dt[v] > dt[u] + wt[u, v]:
report negative cycle
if db[v] > db[u] + wb[u, v]:
report negative cycle
if dt[v] > db[u] + wb[u, v] + transferTime[v]:
report negative cycle
if db[v] > dt[u] + wt[u, v] + transferTime[v]:
report negative cycle

# Merge the results and find min
initialize resultDict
for each vertex in V:
if dt[v] < db[v]:
resultDict[v] = dt[v]
else:
resultDict[v] = db[v]
```

Asymptotic Time and Space Complexity

Time Complexity

We can analyse the time complexity of the pseudo code part-by-part. After we get the graphs and empty dictionaries, we can initialize the db and d dictionaries in $O(V)$ time. The if block runs in $O(1)$ time. In the main loop the outer loop iterates $O(V)$ time and the inner loop iterates $O(E)$ time. Since the db, dt, wb, wt, and transferTime collections are dictionaries, we can assume that the if statements takes $O(1)$ time. So the main loop takes $O(V * E)$ time. The negative cycle detection part takes $O(E)$ time. Creating the resultDict takes $O(v)$ time.

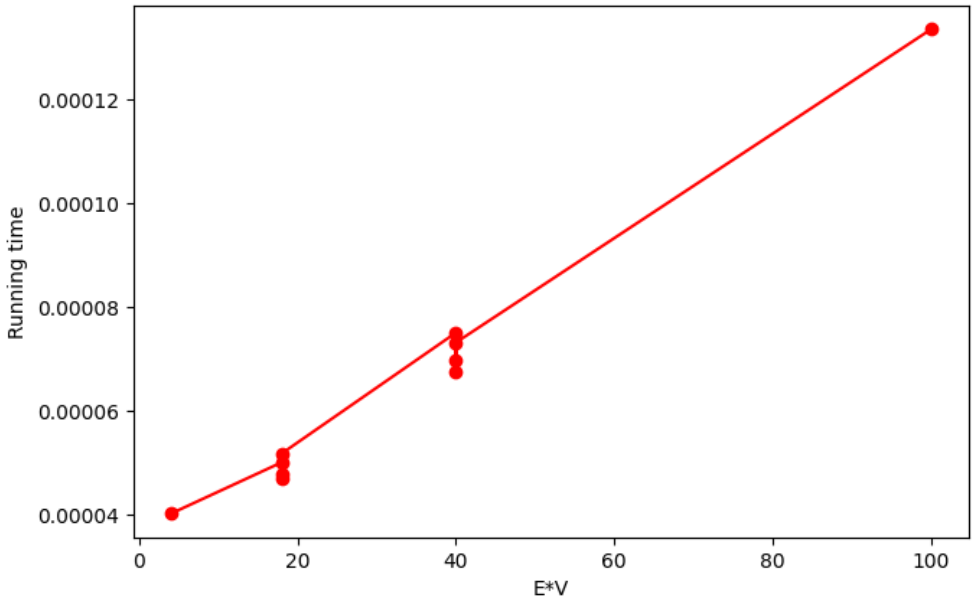
As a result, we can solve the given problem in $O(E * V)$ time in the worst case.

Space Complexity

If we consider the auxiliary space that we use we create several dictionaries to store the intermediate results. Since the graph data structures and transfer time dictionaries as a part to the problem we don't need to count them as a part of the auxiliary space. Beside that we create two dictionaries to store the distances namely db and db which both takes $O(V)$ space. At the end of the problem we also create one more dictionary to store the results namely resultDict which takes $O(V)$ space. If we sum up those auxiliary spaces we get $O(3 * V) = O(V)$ space in the worst case.

Experimental Results

If we implement the algorithm with a dynamic programming we get the expected results.



As we can see from the figure, the algorithm runs with $\Theta(E * V)$ time complexity. This result confirms our time complexity calculations.

References

Lecture Slides of CS 301 Algorithm Course