Assignment 3 - CS 301

G. Görkem Köse

April 2020

1 Recursive Formulation of the Traveling Problem

(i) Identification of Subproblems

To find the shortest path to each and every city, we should consider each possible city that could be a transportation change through the way to that particular target city. For example, if the passenger wants to go to city x, then all of the other cities except x and İstanbul could be an intermediate city to change bus/train. Therefore, to solve the problem of going city x from İstanbul, we should examine the way starting from İstanbul to intermediate city, and from the intermediate city to city x as two subproblems. Hence, at each step, we must consider splitting the way into 2 over and over again, whenever it's possible, to find the path that has the minimum cost. This splitting would be applied for the number of intermediate cities times for each starting city i and target city j. Therefore, the overall number of subproblems becomes $O(V^3)$ where V is the number of cities. Furthermore, due to the recursive formulation that is stated in part (vi), number of possible intermediate cities are also identifies the subproblems.

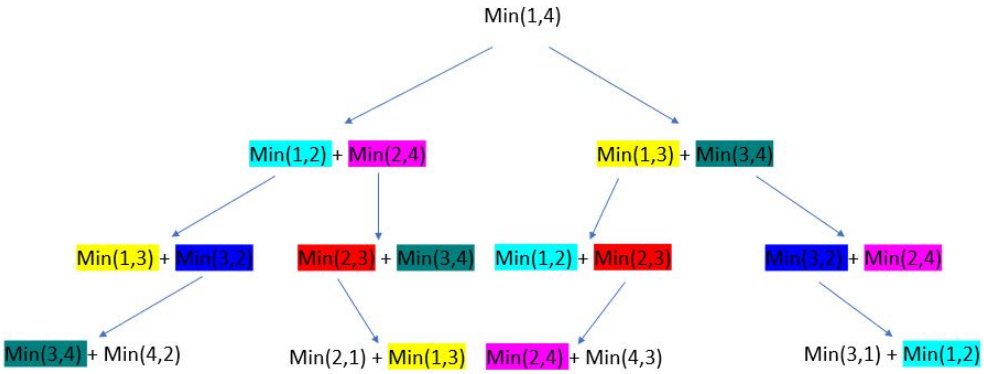
(ii) Optimal Substructure Property

To find the shortest path to cities, even if there is a direct path between them, we should always consider visiting the possible intermediate cities. For example, in order to go to city x from İstanbul, passenger could visit an intermediate city. If we manage to improve the cost of going to intermediate city from İstanbul and/or going to city x from the intermediate city, then the overall problem would be automatically improved.

(iii) Overlapping Subproblems

For 4 cities, city-1, city-2, city-3, and city-4, the following figure shows the subproblems created at each step in order to find the shortest path. The same colors indicate overlapping computations. Of course these would be doubled in the original problem since we should consider different transportation options as

train and bus as well.



(iv) Recursive Formulation Respecting a Topological Ordering
Recursive formulation of the traveling problem via two transportation vehicles is as follows:

For train:

$Train^{(k)}(i,j) = 0$ *if* $i = j$

$Train^{(k)}(i,j) = Train[i][j]$ *if* $k = 0$

$Train^{(k)}(i,j) = min(Train^{k-1}(i,k) + Train^{k-1}(k,j), Train^{k-1}(i,j), Bus^{k-1}(i,k) + Train^{k-1}(k,j))$ *if* $k \geq 1$

For bus:

$Bus^{(k)}(i,j) = 0$ if $i = j$

$Bus^{(k)}(i,j) = Bus[i][j]$ if $k = 0$

$Bus^{(k)}(i,j) = min(Bus^{k-1}(i,k) + Bus^{k-1}(k,j), Bus^{k-1}(i,j), Train^{k-1}(i,k) + Bus^{k-1}(k,j))$ if $k \geq 1$

Since each $Bus^{(k)}(i,j)$ *and* $Train^{(k)}(i,j)$ *depends on the k-1'th superscript of each of these functions and k decreases at each step of recursion, a cycle is not possible. Therefore, we obtain a topological order.*

2 Pseudocode of Naive Recursive Algorithm and Complexity Analysis

```
Procedure byBus(Initial, Target)
        If i = j
                Return 0
        ShortestArray ← []
        For each IntermediateCity
                BusI-K  ← byBus(Initial, IntermediateCity)
                BusK-J  ← byBus(IntermediateCity, Target)
                TrainK-J ← byTrain(IntermediateCity, Target)
                shortestP ← (BusI-K + BusK-J, BusI-K + TrainK-J + Transfer, Bus[i][j])
                Append shortestP → ShortestArray
        Return min(ShortestArray)
Procedure byTrain(Initial, Target)
        If i = j
                Return 0
        ShortestArray ← []
        For each IntermediateCity
                TrainI-K  ← byTrain(Initial, IntermediateCity)
                TrainK-J  ← byTrain(IntermediateCity, Target)
                BusK-J ← byBus(IntermediateCity, Target)
                shortestP ← (TrainI-K + TrainK-J, TrainI-K + BusK-J + Transfer,
                Train[i][j])
                Append shortestP → ShortestArray
        Return min(ShortestArray)
Procedure Naive()
        ShortestPaths ← []
        For each TargetCity
                shortestBus ← byBus(İstanbul, TargetCity)
                shortestTrain ← byTrain(İstanbul, TargetCity)
                Append min(ShortestBus, ShortestTrain) → ShortestPaths
        Print ShortestPaths
```

In the naive function, for each target city we make 2 function calls where the number of cities is n. At each call of byBus or ByTrain function, the possible intermediate cities are the cities which are neither target nor initial indeed. Therefore, the for loop inside of these functions will iterate (n-2) times in the first call where n is the total number of cities. Furthermore, in order to prevent an infinite loop, intermediate city cannot be the initial or target city of the previous call. Therefore until the deepest level of recursion, number of possible intermediate cities would be decremented by 1 at each call. And at each call, 3 more calls of either byBus or ByTrain functions are made. Let's say $k = n - 2$. *So since at each level of recursion, we remove the initial and target city but they would never be in the same function call other than the current one, we remove 1 possible intermediate city from the list we store. Therefore recursion becomes* $T(k) = 3.k.T(k-1)$. *If we continuously unfold that recursion:*
$T(k) = k.3.(k-1).3.T(k-2)$
$T(k) = k.3.(k-1).3.(k-2).3.T(k-3)$
$T(k) = k.3.(k-1).3.(k-2).3.(k-3).3.T(k-4)$
This recursion will end when $k = 0$. *Since for each target city we call byBus and ByTrain function in the Naive function, we should multiply T(k) with 2n since T(k) would be performed for 2n times. Recall that* $k = n - 2$, *hence the overall function would be as follows:*
$G(n) = 2.n.(n-2)!.4^{n-2}$. *Therefore the time complexity of the naive recursive algorithm would be* $T(n) = O(n!3^n)$.

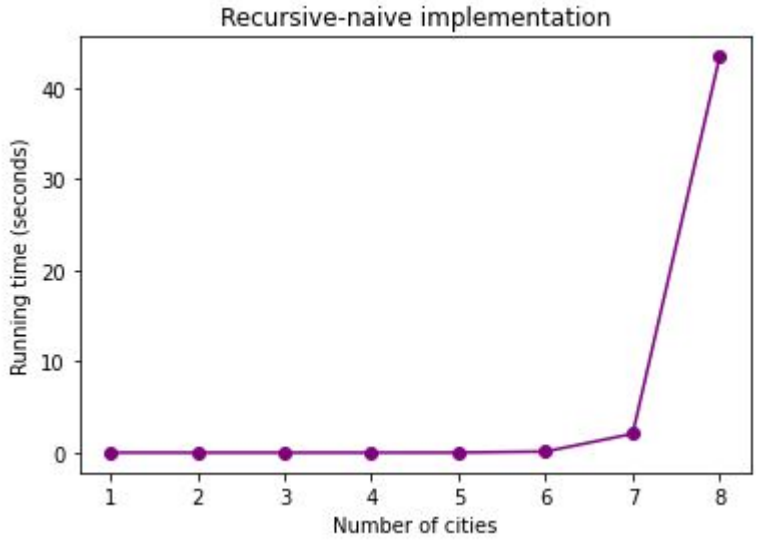
In pseudocode, there is only 1 list is created to store the results which has lenght n. Therefore, the space complexity is O(n). But, in python, lists are reference by default, so at each level of recursion it is necessary to deep copy the list of possible intermediate cities of each level. It's average length is n/2 and the number of copies is 1 for each call. Therefore the number of total copies of this intermediate cities list is 3^n, *we don't have n! here because the creation of new list is outside of the for loop. When we consider the average length of this list, the space complexity would look like* $O(\frac{n}{2}3^n)$.

3 Pseudocode of Dynamic Programming Algorithm and Complexity Analysis

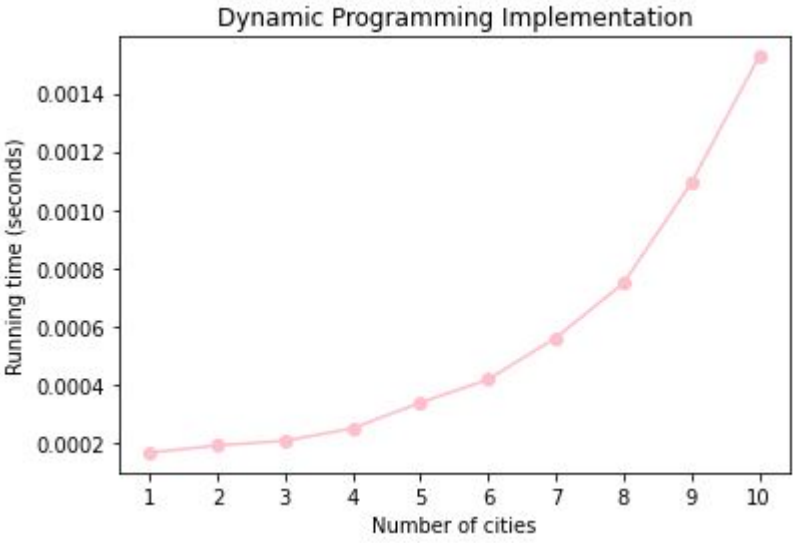
```
Procedure DynamicProgramming
    For each IntermediateCity
        For each InitialCity
            For each TargetCity
                Bus[i][j] ← min(Bus[i][j], Train[i][k]+Bus[k][j]+
                Transfer, Bus[i][k]+Bus[k][j])

                Train[i][j] ← min(Train[i][j], Bus[i][k]+Train[k][j]+
                Transfer, Train[i][k]+Train[k][j])

                Matrix[i][j] ←  min(Train[i][j], Bus[i][j])
    Print Matrix[0]
```

Bus and Train matrices are nxn matrices given as input. At each iteration of these 3 nested for loops, the current cell of matrix Bus representing the path from i to j is updated as the minimum of directly taking a bus, taking a bus to an intermediate city and taking a bus from that intermediate city, and lastly taking a train to the intermediate city and taking a bus from that intermediate city; for Train matrix same approach is also valid. At the end of each iteration, artificially created a Matrix's corresponding cell is updated as the minimum of Train and Bus' i'th row and j'th cell. Since there is 3 for loops that each of them iterates n times, obviously the complexity time complexity is $O(n^3)$ since inside of these for loops the statement we execute take constant amount of time. The space complexity is $O(n^2)$ since we created an nxn matrix, and all the computations are made on that matrix and on the matrices that were given as input.

4 Experimental Evaluations



This graph supports the asymptotic time complexity analysis. It is not even possible to give the input consisting of 9 or more number of cities, because this naive recursive algorithm is not able to run that size of inputs due to exponential time complexity.



This graph shows that this algorithm has indeed polynomial time complexity and grows as fast as n^3 function. So this is indeed expected.

Python implementation of both naive recursive algorithm and dynamic programming algorithm are submitted as a single .py file. The benchmark instances to verify the correctness of algorithms are submitted as a .txt file with their own explanations. To test the algorithms, one of the test cases could be selected from that text file, and given as input to the program. All instances are indeed making black-box testing but during the implementation I also made some kind of white-box testing and check line-by-line, debug my algorithm since it contains a deep recursion. For the performance testing, a 10x10 matrix for Bus and a 10x10 matrix for Train are used by decrementing its columns and rows. These two matrices are also included at the bottom of the text file.

For this homework, I collaborated with Deniz Cangı. Yet all of these solutions and explanations reflect my own understanding.