Assignment 3 - CS301

Duygu Tümer

24882

a) For my algorithm, I am making an adaption on the Dijaktra's Algorithm. However, in this algorithm, cities are vertices and all cities have a bus station and/or train station. If a city has both a bus and a train station, then there must be a path between them, so this path must be also considered since we can start in a bus station and end in a train station in another city. This algorithm should consider these two paths (one for bus and the other one for train). If a city has both a train and bus station then, the shortest paths in both stations will be stored for the other subproblems which have a path on this city.
Let's say that Istanbul-Harem bus station is "s" (starting point) and our finishing points are "f1, f2, …, f80" since we have 81 cities in Turkey. We can describe our subproblem as moving from one city to another city. We are looking for $\varphi_k(s, f)$ where "s" is our starting point and f is our finishing point (it can be f1, f2, f3,...,f80) using at most k edges (paths). After all, we sum them up to find all cities from calculated results.

Recursive Formulation:

$\varphi_k(s, s) = 0 \; for \; every \; k = 0, 1, \ldots, |v| - 1$(base case)

$\varphi_k(s, f) = min\{\varphi_{k-1}(s, u) + w(u, f)\} \; (u, f) \in E$

Subproblems:

$$\varphi_k(s, f)$$

$$\varphi_{k-1}(s, -)\varphi_{k-1}(s, -)\ldots\varphi_{k-1}(s, -)$$

$$\varphi_{k-2}(s, -)\varphi_{k-2}(s, -)\varphi_{k-2}(s, -)\ldots\varphi_{k-2}(s, -)$$

$$\ldots$$

Overlapping Subproblems: At the end, it can be clearly understood that for some cities' shortest paths, it uses the same subproblems (same paths) to calculate the shortest paths for a node. For all nodes, it calculates the same subproblems over and over because we don't remember them.
It has a topological ordering since we can't reach the upper subproblems from lower subproblems which means that there are no cycles.

Optimal substructure: It has an optimal substructure since an optimal solution ($\varphi_k(s, f)$) to a problem contains optimal solutions to subproblems. We can find it using the cut & paste method for correctness. For example, s->u->f is the shortest path from s to f. Then, s->u is also the shortest path from s->u. Suppose otherwise, Then there is another shortest path from s->f which is a contradiction.

That is why, it can be said that it has an optimal substructure and there are overlapping subproblems, so we can use dynamic programming.

b) A naive recursive function:

```
Visited := []
def NaiveRecursion(graph, s, f):
        infinity = ∞
        currentMethod := bus or train

        # Base cases
        if s == f:
                return 0
        if graph[s][f].currentMethod != infinity:
                return graph[s][f].currentMethod
        result = infinity

        Visited.append(s)
        for all neighbours u of s:
                if u not in Visited:
                        recResult = NaiveRecursion(graph, u, f)
                        if recResult != infinity:
                                result = min(result, graph[s][u].currentMethod + recResult)
        return result

for node in f1 to f80:
        NaiveRecursion(graph, s, node)
```

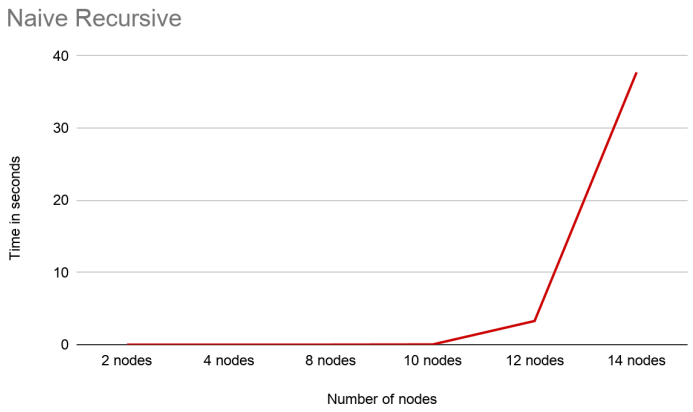
The complexity for a node will be $O(b^{|V|-1})$ where b is a branching factor and our recursive tree's height is |V|-1 . So, we have $O(|V|^{|V|-1})$. At the end, for all nodes our complexity will be $O(|V|^{|V|})$.

c) Memoization:

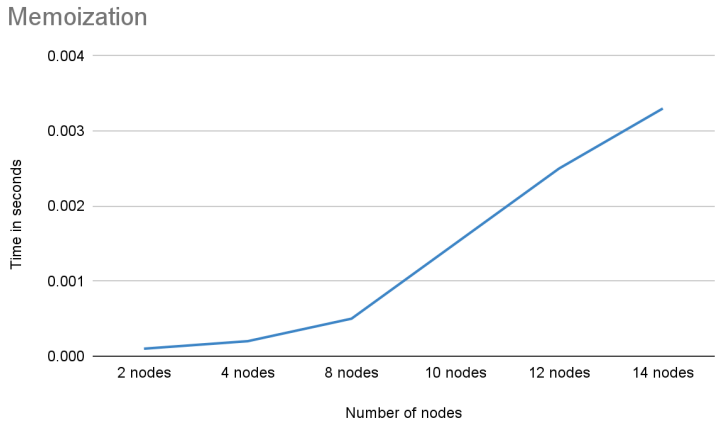
```
def Memoization(graph, s, f):
    global V, infinity = ∞
    currentMethod := bus or train
    sp = [[None] * V for i in range(V)]
    for i in range(V):
        for j in range(V):
            sp[i][j] = [None] * (V)

    for e in range(V):
        for i in range(V):
            for j in range(V):
                sp[i][j][e] = infinity
                if (e == 0 and i == j):
                    sp[i][j][e] = 0
                if (e == 1 and graph[i][j].currentMethod != infinity):
                    sp[i][j][e] = graph[i][j].currentMethod
                if (e > 1):
                    for neighbour a of i:
                        if (sp[a][j][e - 1] != infinity):
                            sp[i][j][e] = min(sp[i][j][e], graph[i][a].currentMethod+ sp[a][j][e - 1])
                for b in sp[s][f]:
                    if b != None and  b != infinity:
                        return b
    return sp[s][f][e]
```

The complexity of memoization bottom up fashion algorithm for all nodes will be $O(V^3 a)$ where a can be 1 to |V|-1. At the end, for all nodes our complexity will be $O(V^4)$in the worst case which is a lot better than the naive algorithm.

d) Experimental evaluations of these two algorithms:



This graph supports the asymptotic time complexity analysis. Due to exponential time complexity, 16 nodes and more is hard to compute the asymptotic time complexity.



Also, this graph supports the asymptotic time complexity analysis. It has a polynomial time complexity, which is expected.

Note:
In my memoization algorithm, there is a problem about some edge test cases. It cannot calculate the exact path between the train and the bus station, so it gives an output less than the naive recursive algorithm. However, this doesn't affect the time complexity.
Also, my benchmark instances are in the same py file with my algorithms. I can change benchmarks with changing “vertices” variable in my algorithm, so it generates random test cases.