Thomas Neyman
CS 4412 - Advanced Algorithms
April 14 2022
Project 5

TSP Branch and Bound

Time and Space Complexity

```
        adjacency_matrix = []
        for i in range(N):
            temp = []
            for j in range(N):
                temp.append(cities[j].costTo(cities[i]))
                if j == N - 1:
                    adjacency_matrix.append(temp)
```

In the first branchAndBound function, the only subsection of code noting is the code that initializes an adjacency matrix for costs of paths. It runs in n^2 time where n is the length of the list of cities. It also takes up n^2 space, as it is a 2D array.

```
    def BeginSearch(adjacency_matrix):
        # initialize the current_path
        current_path = [-1] * (N + 1)

        # create the initial bound
        current_bound = TSPSolver.initBound(adjacency_matrix)
        # Rounding off the lower bound to an integer
        current_bound = math.ceil(current_bound / 2)

        # Set the first values for the current path and visited cities
        visited_cities[0] = True
        current_path[0] = 0

        # Call to TSPRec for current_weight equal to 0 and level 1
        TSPSolver.RecursiveBranch(adjacency_matrix, current_bound, 0, 1,
current_path, visited_cities)
```

All of the assignments in this subsection will be O(1) time. The current path and visited cities variables will only take up O(n) space. The separate functions however take separate amounts of time and space.

```
    def initBound(adjacency_matrix):
        result = 0
        for i in range(N):
            a = maxsize
            for k in range(N):
                if adjacency_matrix[i][k] < a and i != k:
                    a = adjacency_matrix[i][k]

            b, c = maxsize, maxsize
            for j in range(N):
                if i == j:
                    continue
                if adjacency_matrix[i][j] <= b:
                    c = b
                    b = adjacency_matrix[i][j]

                elif(adjacency_matrix[i][j] <= c and
                    adjacency_matrix[i][j] != b):
                    c = adjacency_matrix[i][j]
            result += a + c
        return result
```

This subsection function contains two nested for loops. The loops run for n iterations, making the time complexity O(n^2 + n^2) which simplifies to O(n^2). The space complexity does not change here. The only thing taking space is the result variable which takes up an integer with of space.

```
    def RecursiveBranch(adjacency_matrix, current_bound, current_weight, level,
current_path, visited_cities):
        global final_result
        global pruned
        global total_states
        global count
        global max_queue

        # base case
        # Once all levels have been reached we are done
        if level == N:
            # check if there is an edge from last city in path back to the first
city
```

```
            if adjacency_matrix[current_path[level - 1]][current_path[0]] != 0 
and adjacency_matrix[current_path[level - 1]][current_path[0]] != maxsize:
                # current_result has the total weight of the solution
                current_result = current_weight + 
adjacency_matrix[current_path[level - 1]][current_path[0]]
                if current_result < final_result:
                    final_path[:N + 1] = current_path[:]
                    final_path[N] = current_path[0]
                    final_result = current_result
                    count += 1
            return
```

Next in the recursive branch, the base case only utilizes constant time calls for a time complexity of O(1). A new final path variable is created that will be of size n, so the space complexity is O(n).

```
        for i in range(N):
            total_states += 1
            # Consider next city if it is not the same (diagonal entry in 
adjacency matrix and not visited_cities already)
            if (adjacency_matrix[current_path[level-1]][i] != maxsize and 
visited_cities[i] == False):
                temp = current_bound
                current_weight += adjacency_matrix[current_path[level - 1]][i]

                if level == 1:
                    current_bound -= ((TSPSolver.getFirstCost(adjacency_matrix, 
current_path[level - 1]) + TSPSolver.getFirstCost(adjacency_matrix, i)) / 2)
                else:
                    current_bound -= ((TSPSolver.getSecondCost(adjacency_matrix, 
current_path[level - 1]) + TSPSolver.getFirstCost(adjacency_matrix, i)) / 2)

                # current_bound + current_weight is the actual lower bound for 
the node that we have arrived on.
                # If current lower bound < final_result, then explore the node 
further
                if current_bound + current_weight < final_result:
                    max_queue += 1
                    current_path[level] = i
                    visited_cities[i] = True
```

```
                    TSPSolver.RecursiveBranch(adjacency_matrix, current_bound,
current_weight, level + 1, current_path, visited_cities)

                # prune the node by resetting all changes to current_weight and
current_bound
                current_weight -= adjacency_matrix[current_path[level - 1]][i]
                current_bound = temp
                pruned += 1

                # reset the visited_cities array
                visited_cities = [False] * len(visited_cities)
                for j in range(level):
                    if current_path[j] != -1:
                        visited_cities[current_path[j]] = True
```

This section of code is where the true time complexity of the algorithm is determined. Depending on how quickly the best path is found, the algorithm could run in O(n logn) time, as the recursive search cuts down the iterations logarithmically. However in the worst case, the algorithm is equivalent to a brute force attempting every possible option before finding the best solution. This would result in a O(2^n) time complexity. This does not change the space complexity. Only current path and visited cities lists are being created and updated, no bigger than n the length of cities.

Data Structures:

visited_cities: a queue like list that stores the nodes traveled to so far for the current level

current_path: a queue like list that holds the best found path so far and updates when a more optimal solution is found.

Priority Queue:

The priority queue implementation is simply a list with the optimal solution found so far. It stores the path values found through recursion and puts the best path currently at the beginning of the queue.

Initial BSSF:

The initial bssf algorithm simply starts at the first city and calculates the lower bound from there. The approach is simple and takes a while but gets a good starting point for the algorithm.

Cities	Seed	Running Time	Cost	Stored States	BSSF updates	States created	States pruned
15	20	1.077	9284	81	67874	1010700	332213
16	902	3.643	7124	85	220846	3504288	1082472
20	32	46.429	9936	144	1686458	33690580	11739913
25	64	60.000	10634	156	2658904	53540574	5432634
24	54	15.3002	8430	127	123909	2578930	432893
30	100	60.000	21054	385	3458906	89856748	4256439
35	212	60.000	26430	948	4850320	4489043677	57648743
17	15	7.680	7810	79	418129	7079548	2227096
18	42	6.857	8411	83	310135	5577300	1935725
19	37	38.500	8538	85	1687936	32030656	10071471

Table Results:
The amount of time taken to solve larger problems increases significantly and not in a linear manner. This is to be expected, considering how the algorithm works. The more cities that need to be traveled, the more possible branches need to be explored and because the algorithm can get close to a worst case search, the time increases by a great amount. For example, simply going from a worst case of 2^10 to 2^11 is a very large magnitude for only increasing by 1 city. This is why searching for a path in only 18 cities vs 19 cities in the last two rows can have such a great time difference. Searching for that specific case of 19 cities had a very unlucky start and had to prune many many branches before the optimal solution was found. Overall the table results make sense for how the algorithm is working.

Mechanisms:
In order to determine whether or not a branch needed to be pruned, I simply checked if the weight for the current path was larger or smaller than the weight for the next path that could be taken. If it was larger, it was pruned and ignored, saving lots of time by not traveling down that entire branch with all of its possibilities. It seems to be pretty effective, but it does take a while sometimes if the initial bound isn't very useful.