David Olson

CS 312, Section 2

Project 5

Pseudocode and time complexity

Solve Problem Function:

```
Create an initial cost matrix and reduce it…O(n^2)

Make an initial bssf using the greedy algorithm….O(n^2)

Push the initial state onto the queue….O(1)


---------------------------------the time complexity of this main loop depends on how many states
are pulled off the queue, which depends on the specific problem.  However, it is always only a
constant factor lower than a full state tree, which is O(2^n) .  Also inside the loop we have
O(n^3) operations, so overall we have O(n^3(2^n))--------------------------------------------------------

While the queue is not empty:

    Remove the state with the lowest cost….O(1)

    If the state's path is complete:

        Update bssf and bestPath…..O(1)

        Clean out states in the queue where cost > bssf….O(logn)

    Else:

        Run split decision function, it chooses an edge with which to make 2 new states

        If cost to include < bssf:

            Push the include state on queue….O(n^2) [to make a copy matrix]

        If cost to exclude < bssf:

            Push the exclude state on queue…,O(1) [don't need a copy here]


When queue is empty or time is up, report the bssf and best path…O(n) [make arraylist]
```

Split Decision Function:

```
If the state only needs one more edge to make a path:
    Search matrix for a zero, return those coordinates…O(n^2)
Else:
    Search the matrix and for each zero:…………..O(n*n^2) →O(n^3)
        Copy the state’s matrix…O(n^2)
        Set to infinity the row, column, and any partial loop edges…O(n)
        Reduce the copy matrix to find the cost to include the edge…O(n^2)
        Find the cost to exclude the edge…O(n)
        Compute exclude – include….O(1)
    Return the coordinates that had the max exclude – include
```

Space Complexity

Space complexity is slightly better than time, because no matter how many times we loop through the matrices, the space to store them remains the same. Also, some temporary data is stored and deleted, but the overriding factor is that there are O(2^n) states possible at any time, and each state has to store a matrix and a path array, but that is just O(n^2). The total space complexity is O(n^2(2^n))

State Representation

Each state is contained in a node in the binary heap. Hence, in my code I’ve included my node and binary heap classes. Each node has a key, which is the lower bound cost. Each node also has a 2D array for the cost matrix, and a 1D array representing the path so far, where each index is the city with the same index in Cities, and the value is where the edge leads to. If there is no outgoing edge that value is a -1. Lastly, each node has a counter for number of edges in the path, which my program needs to know once we have just about all the edges included.

Queue Implementation

I wrote as small of a binary heap class as I could, based off the pseudocode in our book that we used back when we were doing Dyskstra’s algorithm.

Initial BSSF

I quickly coded up a greedy algorithm method, ran that on my initial cost matrix, then set bssf and bestPath to the results of that. I'm not sure if there were other great ideas for an initial bssf, but I was at a loss as to what I could very quickly do that was better than greedy.

Results Table (running time in release mode)

Run #	# Cities	Seed	Running time (sec.)	Cost of best tour found (*=optimal)	Max. # of Stored states at a given time	# ofBSSF updates	Total # of States Created	Total # ofStates Pruned
1	15	20	.597	2430*	1004	1	5825	2911
2	16	902	.082	3223*	327	1	679	338
3	19	20	2.221	3116*	7736	1	15695	7846
4	24	92	9.052	4158*	22995	1	46605	23301
5	30	1	30.005	5243	47894	0	95787	47893
6	30	565	30.002	6140	52478	0	104955	52477
7	34	859	12.230	4833	999	1	38207	19102
8	38	318	19.966	4771	999	1	49003	24500
9	42	172	22.345	5140	999	1	36031	18014
10	46	945	18.284	6258	500	1	26027	13012
11	50	87	13.996	6118	200	1	14323	7160

Analysis(with extra credit)

The first 4 rows shows that the algorithm ran fairly fast, and found the exact solution. This truly is an exponential algorithm, though, because any city size in the high 20's or above really puts a strain on my program. Assuming I calculated the numbers correctly, it looks as though by thirty seconds I don't ever find a path that improves my bssf. The time and space complexity are obviously growing exponentially. That is shown in rows 5 and 6. I drastically reduced the max size of my heap for the last 5 rows though. If I hit the limit, my program adds the include state, but not the exclude. These solutions were certainly not optimal, as some of the tree was cut off, but I found a better solution than greedy, and it was a pretty good route each time. Most importantly though, it didn't take a super long time to run.

Late Days

Please apply my last late day to this project, and any other late days that are in your heart