CS2521 Assignment – Report

Ola Stefan Rudvin 51549217

This is the report for the CS2521 Assignment to evaluate student's ability to create, analyse, implement and evaluate algorithms. The code for the Taxi Allocation and Passenger Routing sections are found under their respective folders with all required code named by question numbers.

Note: If the code is run with a different set of inputs form a file, the trailing spaces on each line must be identical to the provided examples.

1.1 Taxi Allocation

1.1.1 Tasks

1. *Describe a brute-force algorithm which addresses the taxi allocation problem, and identify and prove its time complexity.*

For the taxi allocation problem, a brute force algorithm goes through every possible combination of taxis with passengers, without using the same passenger for two taxis in the same combination. This provides time complexity n!.

To prove this, consider a table of n * n, where n represents the number of taxis and passengers. Since each taxi can only take one passenger and one passenger can only take one taxi, the combination decreases by (n-1) for each combination. The total combination then adds up to (n)*(n-1)*(n-2)*(n-…) = n!. For a table of 3 * 3, the maximum number of combinations is 6, which is n! = 3*2*1.

2. *a) What is the time complexity for a brute force approach to implementing Step 3? Explain why.*

Brute forcing step 3, drawing the minimum number of lines in each column we find a time complexity of 2^n. This is due to the approach drawing every combination of lines with an n*n matrix, then checking whether those combinations of lines cover all of the zero's. The minimum value is saved and returned at the end.

2. *b) Provide the pseudocode for, prove the correctness of, and implement, a more efficient version of step 3. Identify/prove your algorithm's time complexity.*

Function step3(array)

For each row:

If row has exactly one 0 excluding values under lines:

Draw a line through the 0's column

If all zero's are not covered:

For each column:

If column has exactly one 0 excluding values under lines:

Draw a line through the 0's row

This algorithm is implemented as part of the complete algorithm in the taxi allocation folder.

Firstly, the algorithm terminates – it terminates once all 0's have been covered- since it goes through columns and rows.

One can of course also look at a simple base case n(2), a 2*2 matrix with 0's at (0,1) and (1,0). The algorithm goes through the first row, finds a 0 at (0,1), then moves to the 2^{nd} row where it finds a 0 at (1,0). In total it draws two horizontal lines, which is the minimum possible with the given inputs.

The time complexity for this task is $O(n^2)$ for its worst case.

2. *c) Provide the pseudocode for an algorithm to achieve step 4, prove your algorithm's correctness, and identify/prove this algorithm's time complexity.*

Function step4

If number of lines does not equal number of passengers/taxis:

Subtract minimum value of the uncovered values from all uncovered values and add the same value to the intersection points of where lines meet. Return two previous step.

Else: Optimal solution is found!

For a 2*2 matrix, if there are two lines – and with step 3 we can see that they are not overlapping & parallel, and that they do not mark the same node, the algorithm picks the lowest unique cost passengers for each taxi correctly. If the number of lines is not 2, the algorithm repeats the earlier steps after adding to the uncovered values and subtracting form the intersections. This makes sure that the intersecting values are not selected as unique again, and making the uncovered values less favourable to be selected since they are most probably not the most efficient paths anyways.

The complexity for this step is 1) finding uncovered values 2) finding min value 3) subtracting from them 4) Finding intersection 5) Adding to them.
This totals to $O(n^2) + O(n) + O(n) + O(n) + O(1) = O(n^2)$

2. *d) Implement the entire algorithm and empirically evaluate its average time complexity. Such an empirical evaluation should consider the runtime over different numbers of passengers and taxis. Remember also that a single run may not be representative of average algorithm performance.*

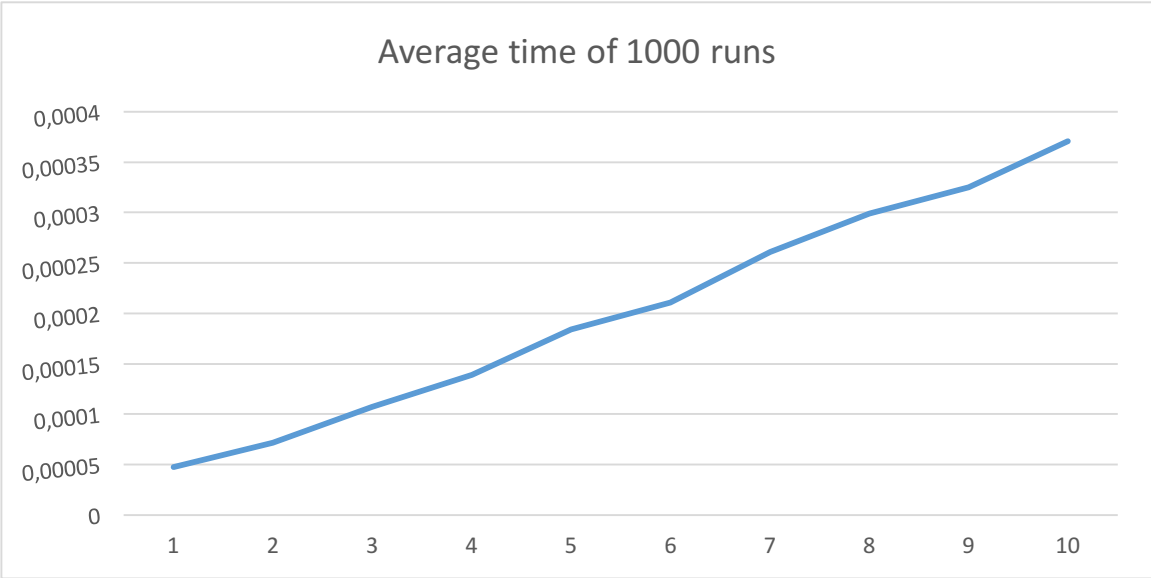
Complexitites:

Row reduction: $O(n^2)$

Column reduction: $O(n^2)$

Worst case complexity: $n*(n^2) = n^3$

The following graph shows the runtimes of 1000 runs with incrementing passenger and taxi numbers. The n*n matrixes are created out of random values from 0-20. The graph should give an n^3 result but with such small values the results appear to be linear.



1.2 Passenger Routing

Note: The Dijsktra's algorithm used is heavily inspired by a stack overflow post. This is also shown in comments in the code.

1. *a) identify the total travel time for all the passengers (by implementing an algorithm to do so). Describe the algorithm you used for this (justifying its use), and empirically evaluate its time complexity, comparing it to its worst case complexity.*

To identify the total travel time for all passengers I used Dijkstra's algorithm. I used Dijkstra as it is one of the most efficient pathfind algorithms, and unlike Floyd's it does not need to compute all shortest paths, just the shortest paths that are needed. It also uses less memory. Also there is no information on distances between nodes, so other algorithms such as A-star could not be used. The worst case complexity of my implementation of Dijkstra's algorithm is $O(E + V^2)$ which dominates to $O(V^2)$ as I did not use a Fibonacci heap.

1. *b) Can you suggest a O(n) algorithm which can solve this problem? Explain why is such an algorithm infeasible for large maps?*

By precomputing the shortest distance form every node to each other using Floyd's algorithm and storing them in an n*n matrix, we can reach an O(n) solution. The traversing takes O(n) through the matrix, as all the shortest paths are already calculated.

This algorithm is infeasible for large maps due to it using a large amount of memory (e.g. with a million rows/columns). Also it requires the precomputing of the matrix, which is $O(n^3)$ complexity which obviously takes a long time.

2. *a) Timetabling*

To solve the 'timetable' public transport problem, I used a different method than the one described. Rather than creating a massive amount of nodes for each arrival or departure at a specific location, I used a different method. Each node is created as a graph without regard for time. Current time is tracked in a separate array, as well as each departure for each location in a separate dictionary. As a neighbour is evaluated, the algorithm finds the next available departure and adds the wait time to the edge weight. Current time is then adjusted for each node separately.

2. *b) Given n nodes in the original map, and m lines in the public transport options file, determine what the maximum size of the resultant expanded graph could be, and justify your answer.*

The maximum size for the resultant expanded graph would be 3nm. This is true because for each node that is added, a maximum of three nodes are created: One for the main node, one for a travel node and one for a 'stay' node. Furthermore, this is multiplied by m, depending on how sparse the graph is.

2. c) Empirically evaluate the average time complexity of your algorithm with regards to different public transport frequencies while routing one passenger

The following graph shows how adding more public transport routes affects the running time of the algorithm. It was difficult to evaluate which routes should be added, since adding routes that will not be visited won't affect the runtimes. Therefore, the results should be taken with a grain of salt. Essentially this graph should be of n^3 time complexity.