Part 1: The 16-puzzle

Algorithm #2 is used for solving this problem.

Heuristic Functions tried (not all were admissible):

1. Permutation Inversion: This heuristic worked only till 6 moves, and was too time consuming.
2. Manhattan Distance: This heuristic function worked for board 10 but as the function is not admissible, it wasn't giving an optimal result.
3. Difference between tile's current position and required position (distance in 1d array): This heuristic worked only till 4 moves and is not admissible and consistent.
4. Number of misplaced tiles: This heuristic worked only till 6 moves.
5. Rotations (Manhattan considering wrap around moves): No of rotations is a customization of Manhattan distance for this particular problem. This customization made the function admissible, but it was still time consuming. For our initial version of program where we were not using a priority queue and were using algorithm #3, it took almost 10 minutes to solve a board which could be solved by 8 moves.
6. Tiles not in correct row + tiles not in correct column: This heuristic looked promising initially, but took 1 minute to solve board10. And could not solve board12 at all.
7. No of moves for sub problem: Here we considered single row as a sub problem and remaining rows as blank tiles and computed result for solving for each row, summation of which, resulted into a heuristic function. However, this couldn't even solve board2! Precomputing results would have helped, but that would have needed us to pre compute and store data in a file, so we did not work on it afterwards.
8. Manhattan Distance + Number of rotations: Considering cost of moving from one node to another to be 3 i.e. g(n) = 3, this heuristic gave the best results after optimizing the code for performance. This combination gave solution for board with 10 moves in 0.2 seconds. In this case, although we are adding two heuristic functions, still the result it gives is always optimal (at least for those inputs which we tried). This might be because we have considered g(n) = 3.

We also tried greedy instead of A* (just out of curiosity). It gave result for board12 within seconds, however, it was way too far from an optimal result (It took around 120 moves).

Approach used to improve performance:

To implement A* algorithm, we are exploring all nodes from the fringe having minimum cost. For example, if we have 4 nodes in fringe having lowest cost, we would pop all elements and find their successor instead of picking just one of those 4 nodes. Also, we are checking if a particular node to be added in fringe, is already visited and we are discarding all nodes that are visited and have more path length. In case if, Current node's path length < Visited node's path length, we are updating visited node's array and fringe with new node's values. Switching from Algorithm #3 to algorithm #2 also improved performance a bit. We also implemented heap queue for fringe which was suggested by Utkarsh Kumar.

Part 2: Road trip!

We have used 2 classes Node and Edge to store the data. Node class contains vertex as city name and latitude and longitude. Edge class stores, highway name, city1, city2, distance between 2 cities and speed limit.

We have defined graph as a dictionary with key as city and value as a list of dictionary where each dictionary object is a key value pair of neighboring city and edge connecting 2 cities.

While reading city-gps.txt and road-segment.txt we came to know about some noise in data and we have considered below assumptions to recover noise –

1. There are some road segments where speed limit is empty or 0. So we are discarding such rows.
2. There are many cities which are not present in city-gps.txt but are used in road-segment.txt. While reading such roads, we have added such cities in our node list initially with latitude and longitude as 0. After building a complete graph, we are calculating its GPS values based on its neighboring cities.

Algorithms –

Fringe structure – city, cost, predecessor

Where predecessor stores a path from start node to current node and it will be used in the end to get a complete path.

BFS – BFS works as first in first out. So graph nodes will be evaluated in a same manner without considering a lowest cost.

DFS – DFS works as last in last out. So nodes will be evaluated in a same manner without considering a lowest cost.

IDS – Iterating deepening search will work like DFS till a specified depth if no solution is found, a depth value is incremented and algorithm will start searching again with the new depth. We have used initial k (depth value) as 10 and if a solution is not found within given depth we are incrementing k value by 10.

Uniform – Uniform search takes cost value in consideration. For any node, algorithm will choose a successor having lowest cost where cost can be segment, distance or time as specified by the user.

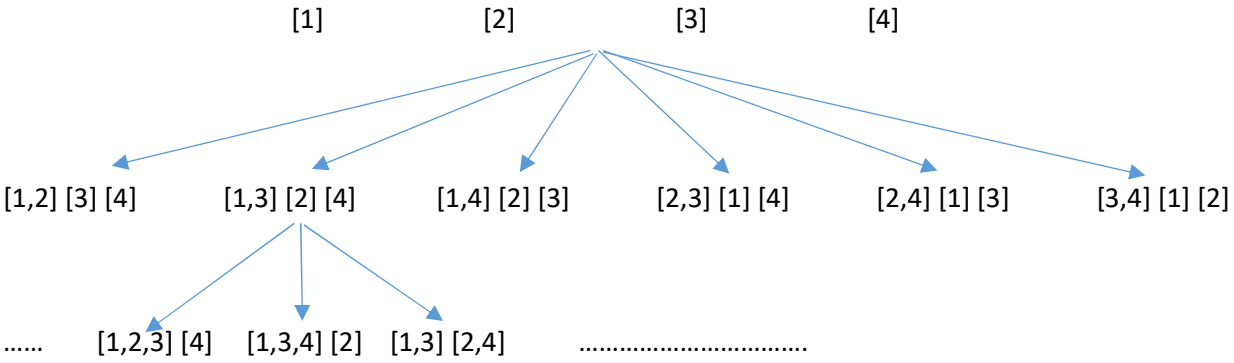
Astar algorithm – Astar algorithm uses cost function f(n) = g(n) + h(n)

In our implementation, we are using g(n) as a path cost(segment or distance or time) as specified by user and h(n) is a heuristic function which tells actual straight distance between start city and destination city. We took help of https://www.movable-type.co.uk/scripts/latlong.html to derive a formula of calculating distance using latitude and longitude.

Part 3: Group assignments

If we have 4 students (1, 2, 3, 4) in our class. Then we will start by considering set of groups having a single student in each group. So first combination will be: [[1], [2], [3], [4]]. Then in the next step we find successors of the state by merging all possible combinations of any two groups (Adding one condition to limit group size to 3) and keeping rest of the groups as they are.

One example is illustrated below:



Initially, we built a search tree considering all possible combinations using the strategy above. This version of program worked for input given by Prof. Crandall, however it wasn't working for input where 52 students were included (It ran for whole night and still was unable to give solution) due to the huge state space the program was working on.

So, in order to reduce time, we added only those nodes to fringe which had cost lesser than minimum of all nodes visited till now. Here set of 'visited' nodes contains all nodes that were added to fringe in past. This version of code gave answer for class having 52 students and gave result in approximately 30 seconds, but did not work for 104 students.

Reducing the search space further seemed to be difficult. So instead, we divided the students in 'n' number of groups, where ($n = \frac{no\ of\ students}{55}$). We chose 55 the program successful computed result for 55 students in less than a minute. This version of program computed answer in couple of minutes, but it does not guarantee an optimal solution for class having number of students greater than 55.

Assumptions made for Part 3:

1. Input file does not contain any spaces in the beginning or at end of every line
2. Only one space is present between student, preference list, etc.
3. All students have filled the form i.e. there is an entry for each and every student in the input provided.
4. Students have not made any spelling mistakes while entering their preferences.