Problem 1

Steps to run -

1) Ensure that the input files - road_segments.txt and city_gps.txt are in the same folder as the problem1_solution.py program file.

2) Use the command line format - python problem1_solution.py [Bloomington,_Indiana] [Chenoa,_Illinois] [distance] [ids]

How search problem was formulated
A given source city can lead to various cities, which can lead to more cities and so on. We have to find the destination city in this state space generated by the successor function. It is similar to a graph traversal search problem.

State space
The state space is a set of all the cities which can be led by the successor function from initial given source city to its neighbors and so on. Each neighbor city in turn leads to multiple cities with different costs and highways . Also, bi-directional routes exist between any given pair of neighboring cities. Entries in the state space only consist of

1) Those cities which are not already a part of the fringe
2) Bidirectional paths which form a loop to the parent of the item whose successors are being generated.
3) Any city which has been marked visited by the search algorithm

Successor
The successor function will generate all the cities that can be traveled from the given city via different highways and roads. This information about all the edges in the graph is retrieved from the road-segments.txt file and all edges are assumed to be bidirectional. Successors conform to the three criteria listed above.

Edge weight
For cost function we have assumed whatever the user provides from the command line argument to optimize the paths being discovered i.e. the routing options like distance, time, segments and scenic. All the edges are read from the road_segments.txt file and then stored in a dictionary data structure with two corresponding entries owing to the bidirectional nature of the edge. For segments we used a variable called as 'segments' for all the edges in a dictionary data structure, wherein we assigned a cost of 1 for each edge. For travel time we used another variable to calculate the time cost of an edge based on the distance of the edge and the speed limit for that edge. For 'scenic' we have taken another variable to be 1 if the edge is a highway as defined by the speed limit or else it is equal to 0. 'Distance' is already a given, so that is stored against each edge in the dictionary too. These variables help us to optimize the cost function which user provides from command line.

Heuristic

For heuristic function to estimate how far we are from the goal state we have calculated the geographical displacement between the current city(not source, any intermediate city that we discover might lead to goal) and the destination city using the latitude and longitude information from city-gps.txt file. For cities where that information was missing(e.g. Highway intersections) we have calculated the distance between the nearest city from that highway to the destination city and added the distance between the highway to the nearest city of highway to accommodate for the missing latitude longitude information for the highway intersections. This will always be admissible since it calculates the displacement which is the minimum distance possible.

For time as routing option we divided the above calculated distance from the highest speed limit that is possible on the roads of USA/Canada. This heuristic will always be admissible.

How the algorithms work

1) **BFS -** Expands all paths from immediate neighbors of a given city and examines their validity for being eligible for the fringe. It exits searching whenever a goal node is found . (reference to the Piazza post on relaxing conditions for optimality)
2) **DFS -** Does the same by probing neighbors of neighbors of one particular node until it reaches the edge of the graph, before examining other immediate neighbors of a given city. It exits when it reaches a goal state.
3) **IDS -** Leverages DFS iteratively to look for a goal at increasing depths of search. This also exits as soon as it reaches a goal state.
4) **A* -** The fringe here is implemented as a priority queue and it works by examining paths which have the least path cost at any given instant of exploration. If a better path exists, it is extracted from the fringe first and is explored before other successors in the fringe are examined. This provides for the generation of an optimal path. The heuristic employed is a simple geographical distance calculated based on latitude and longitude values from city_gps.txt values. A case where the search encounters a highway intersection is handled by including the path to the neighbor of the highway intersection which is closer to the goal state based on its latitude and longitude values.

Assumptions -

1) Entries with missing values of distance, speed etc are not valid entries and hence are eliminated
2) Entries with values 0 for distance and speed are invalid and hence omitted from the search space.

Q1.Which search algorithm seems to work best for each routing options?

Ans. A* seems to work best for each routing option according to our experiments because of the following reasons -

1) It gives an optimized goal path
2) Least run-time for different routing options
3) Least fringe memory (fringe is maintained as a priority queue)

Q2.Which algorithm is fastest in terms of the amount of computation time required by your program, and by how much, according to your experiments?
Ans. A* search seems to be the fastest algorithm as it keeps track of the lowest cost of every expanded path and switches paths based on the same measure.

Following are some run-times on the input:

python problem1_solution_github_synced.py [Chicago,_Illinois] [Schaumburg,_Illinois] [routing-option] [routing- algorithm]
where **routing-option** is one of **distance**, **segments**, **time** or **scenic** and **routing-algorithm** is one of **bfs**, **dfs**, **ids** or **astar**

	BFS	DFS	IDS	A-Star
Distance	0.200127840042	26.8855080605	0.47216796875	0.141630887985
Time	0.204982042313	25.8666880131	0.475928068161	0.17330789566
Segments	0.191943883896	25.7925200462	0.493514060974	0.173902988434
Scenic	0.194792032242	24.9982881546	0.472812891006	0.17181801796

Comparison of run-times for different algorithms with different routing options (all units in hours)

Q3.Which algorithm requires the least memory, and by how much, according to your experiments?
Ans. According to our experiments, the memory consumption (found by calculating maximum fringe size) is in the order - DFS > BFS > IDS > A-star

Q4.Which heuristic function did you use, how good is it, and how might you make it better?
Ans. We calculated the geographical distance between the current city and the destination city from the given latitude and longitude from city_gps.txt and in case when this information was missing from city_gps file for highway intersections we calculated the nearest towns to the highway intersection which is closest to the goal and substituted its latitude and longitude calculated (geographical) distance plus the distance from highway intersection to that nearest town as the current cost.

This heuristic works good for distance, scenic and segment routing options. For the time routing option we divided the lat-long calculated distance by the maximum speed limit to calculate the heuristic.

This heuristic function can be made better by keeping track of a metalevel-state-space and using a metalevel searching algorithm on the state space to avoid exploring unpromising paths.

Problem 2

How search problem was formulated

The initial board state will be read from the file and the goal state is to reach the board configuration where all tiles are arranged in increasing order from 1 to 15 on the board with last board position being empty tile. We have to search through the different states that can be generated by the successor function from given board configuration and find the goal board configuration.

State space

The state space is a set of all the board configurations which can be generated by the successor function from initial given configuration. Each board state generates four different states by moving the tiles left,right, up and down to the empty tile position . These states can further generate four different states each and so on.

Successor

The successor function will generate four different states by moving left right up and down to empty tile.

Edge weight

For cost function we have assumed that for moving one tile either left right up or down it takes 1 unit of cost.

Heuristic

Manhattan distance, was used as a heuristic for a given tile from its current position to its goal state position. This heuristic works well for most problem states.

Another possible heuristic function is the number of misplaced tiles on the board. Both the aforementioned heuristics are admissible, but the value given by the heuristic function of Manhattan distance, is at least as high the value given by the latter, which makes it the better heuristic function between the two.

Yet another possible heuristic function is a sort of a modified Manhattan distance, which takes into consideration the presence of the moves added to modified 15-puzzle problem. This makes the heuristic more accurate, but does not always make it stay consistent. It may improve the overall running time of the algorithm and help the program come across the solutions quicker.

Problem 3

How search problem was formulated

The input file provides the friendship graph of each member. The search is to performed through the disjoint-friend graphs. The goal state is reached when all the members have been placed on the board. To reach the goal state/complete the solution, for each given table, corresponding possible members should be seated.

State space

The state space consists of all possible arrangements between people such that each table has anywhere from 1 to <seats-per-table> number of people as mentioned in the input, and all members have been placed on a table.

Successor

The successor function checks for each member if they can be placed on the table, and generates a state which places a member such that he doesn’t know anybody at the table, or if the table is full it allows for creation of a new table with new members and the same principle.

Edge weight

The edge cost is one since, placing a member on the table does not depend on who is placed, so long as the constraints are satisfied.

Heuristic

No heuristic was used and the algorithm used to solve the function is bfs.