Artificial Intelligence PLH311
First Programming Assignment

Team LAB31146761
Diamantis Rafail Papadam - 2017030044
Alexandros Konstantinos Valtatzis - 2017030097

March 28, 2021

Introduction (Translating the problem to code)

The first step towards implementing a solution for the first programming assignment was reading the input. For that purpose we created a file under the name **"Reader.h"** which contains all of the functions that take care of reading the input for us. Specifically the function named **"readInput(string buf)"** takes as a parameter the whole input file in the form of a string and returns a custom data structure **"struct input"** which contains everything that has been read in a convenient form for us to process. The next basic step was to describe the problem in the form of data structures. A big part of that was the graph representation that we were going to use. We concluded that it is a better option to go with an **adjacency matrix** rather than an adjacency list even though the latter takes less memory to store. To get to that conclusion we considered all of the possible operations that our program will need to execute:

Operations	Adj. Matrix	Adj. List
Adding a road	O(1)	O(1)
Removing a road	O(1)	O(E)
Finding road between intersections (i and j)	O(1)	O(V)

Note that E corresponds to the number of roads and V corresponds to the number of intersections.

Considering the comparisons in the previous page, it is clear that for the purposes of this specific project and the way we have decided to implement the solution, going with the adjacency matrix structure will result in lower running time when executing operations on the graph.
We don't mean to get into much technical detail but it is useful to note that we also used 3 hashmaps, **"roadToIndex**", **"nodeToIndex"** and **"indexToNode"** so that we are able to switch from string representation of a road or intersection to integer representation (and vice versa for intersections) conveniently and quickly.

Pruning the graph

The only time we pruned the graph was right after reading the input. To be more precise, we took rid of unnecessary roads before we even created the graph. The complexity of doing such thing cost us $O((\#roads)^2)$ where the symbol "#" means "number of". The function doing this is "**discardUnnecessaryRoads(roads)**" and it is used at line 21 of our main located at "**project1.cpp**". What this function does is to compare every pair of roads $(road[i], road[j])$, such that $i \neq j$ and check if road[i] on heavy traffic is "cheaper" than road[j] on low traffic. If such pair is found, then it is obvious that road[j] can be discarded and not used in the graph at all.

Probability Distribution

We are given by the problem statement a probability distribution that can take 3 values, p1, p2 and p3. p1 is defined as the probability that the given prediction is correct whereas p2 and p3 are the probabilities that the current predictions should take one of the other two values. What we weren't given is the way of mapping p2 and p3 in the set $\{low, normal, high\}$ according to the prediction. Therefore it was in our judgment to decide how to do that and we came up with the following convention:

$P_X(x)$	low	normal	heavy
$x = 1$	low	normal	heavy
$x = 2$	normal	heavy	normal
$x = 3$	heavy	low	low

As for the algorithm that changes p1, p2 and p3 daily, things are pretty simple. Let's suppose that we are at the n^{th} day where $n \in \{0, 1, ..., 80\}$. Also it is known that there are $m = \#roads$ predictions. We define the counters below:

- $p1\_counter = \sum_{i=0}^{n} \sum_{j=0}^{m} (pred[i][j] = act[i][j])$

- $p2\_counter = \sum_{i=0}^{n} \sum_{j=0}^{m} (p2\_a \cup p2\_b \cup p2\_c)$, where:
  - $p2\_a = (pred[i][j] = low \cap act[i][j] = normal)$
  - $p2\_b = (pred[i][j] = normal \cap act[i][j] = heavy)$
  - $p2\_c = (pred[i][j] = heavy \cap act[i][j] = normal)$
- $p3\_counter = \sum_{i=0}^{n} \sum_{j=0}^{m} (p3\_a \cup p3\_b \cup p3\_c)$, where:
  - $p3\_a = (pred[i][j] = low \cap act[i][j] = heavy)$
  - $p3\_b = (pred[i][j] = normal \cap act[i][j] = low)$
  - $p3\_c = (pred[i][j] = heavy \cap act[i][j] = low)$

where $pred[i][j]$ is the predicted traffic for the i^{th} day on the j^{th} road and $act[i][j]$ is the actual traffic for the i^{th} day on the j^{th} road.

To put it simply, we count the number of times that the prediction was correct up to this day (n^{th} day). p1 will be that number divided by the total number of predictions so far. Similarly for p2 and p3, we count the number of times their definitions "took place", and we divide that number with the total number of predictions so far. With this process we believe that we have optimal p1, p2 and p3 on a daily basis and they cannot be made any more accurate with the information that we have collected up to the n^{th} day.

Uninformed Search

For our uninformed search algorithm we chose **optimized Dijkstra** known as **Uniform Cost Search (UCS)**. The logic that this algorithm follows is well defined in a source that is linked in the bottom of this report. Having said that, we are not going to get too much in depth but rather explain a few key differences between Dijkstra & UCS. Dijkstra begins with a set of all unvisited nodes denoted as Q by initializing their distances from the source as infinity. Then the distance from the source to the source is initialized to 0 and the algorithm begins its well known process until Q is empty. It makes a total of V steps where V is the number of nodes of the graph and in each step the node with shortest path from the source is being "finalized" and removed from the set Q. In each one of those steps we need to make $log(V)$ operations in order to perform what is known as **"heapifyDown()"** and "re-balance" the minimum heap.

The optimization that happens to turn Dijkstra into UCS is the fact that the set Q works differently now. Instead of beginning by storing all of the nodes in the set, at each step the algorithm stores only the nodes that can be expanded in the next iteration. This might not be making any change to the asymptotic complexity of the algorithm (which remains $O(E+V*log(V))$) but it is obviously making the **"heapifyDown()"** function run faster since it will now perform less operations in every single step. Apart from the improvement on run-time, it is also clear that we have a better space complexity as well, not asymptotically of course but in practice UCS will always require less memory than Dijkstra.

Note: In the "worst" possible input that will practically never occur, both algorithms will require the same memory as well as time in order to run and find an optimal path.

Informed Search

As an informed search algorithm we were required to implement iterative deepening A-star (IDA*) search. IDA* is basically a slower version of A* that lowers the use of memory significantly, to be exact it reduces $O(b^d)$ to $O(d)$.

In order to get there we first had to wrap our minds around the concept of A*. For that purpose we visited some of the sources that are linked below and not only did we see how the A* is combining what it knows so far (g function) with the heuristic information that is provided (h function), but also comparisons on interactive simulations between this newest for us algorithm with algorithms we already knew, such as DFS, BFS, Dijkstra, etc. It was interesting for us to explore how this new aspect of "smartness" called a heuristic function is helping a search algorithm perform better, so after understanding what is an admissible and consistent heuristic it was very helpful to think about extreme cases. For example if the heuristic has a value of 0 for every single node, then to put it simply, the A* algorithm loses its "intelligence" and it becomes nothing more than UCS. On the other extreme case, if a heuristic is "the optimal" heuristic, and that is basically defined as having the maximum distance value for every single node (from the source) and still remain admissible, then the A* becomes so "smart" that it will only expand nodes that belong in an optimal path.

Having understood the above it was time to create our heuristic function. Our first thought was to visit every neighbor of the destination node, get the shortest distance and that would be our heuristic value for every single node in the graph except the destination. Obviously that's not a very helpful heuristic, is it? It's insignificantly better than starting with no heuristic at all. After implementing A* with this bad heuristic, we realized that LRTA* can be used multiple times in order to improve the heuristic function. In this paragraph we want to finish describing the heuristic that we used so we are going to take things in a non-chronological order. The heuristic that was finally used before running the IDA* algorithm (even though at this point we had only done the A*) was running LRTA* "$lrtaStarNo$" times on the graph and only then executing IDA* search, the improvement we noticed with that heuristic was extremely significant, in all 3 of the given input files the heuristic had converged into "the optimal" heuristic, as described earlier, in only "$lrtaStarNo = 50$" trials of LRTA*. In fact, somewhere between 10-20 trials were needed in all 3 given input files for the heuristic function to converge to the optimal one.

The last step to completing our informed search algorithm was to add iterative deepening to the A* search. That was quite straightforward considering the example we did in class as well as the implementation of IDA* on wikipedia linked below. We would like to note here that when the heuristic becomes optimal, then the IDA* "becomes" the A* because we basically know the exact path from the source to the destination from the heuristic function. Also each node contains its exact distance to get to the destination. Therefore the initial maximum depth to search, which is chosen to be $0+h(source)$ will never be exceeded until we get to the goal. In other words, for an optimal heuristic there will be no iterative deepening but IDA* will find the goal in its first try.

Online Search

For an online search algorithm we were required to implement LRTA*. The sources that were very useful in doing so were the lecture slides as well as a paper submitted in 2006 which we have linked below. The logic in LRTA* is that our algorithm realizes the environment by exploring it step by step. In each one of those steps, the algorithm is making the locally optimal move, in our example, it's taking the road that is more promising to lead to the goal state. That may be a little abstract so let's define it with more precision. In each step we "generate" all of the current state's children, and we go to the one that has the minimum f cost, where f is defined as the cost of getting to it + its heuristic cost to get to the goal state. One key point that makes this algorithm work, is that at the end of each such step we try to improve the current state's heuristic function before moving on to the next state. So if the f cost of going to the "cheapest" child is greater than the current heuristic estimate, we do $h[state] = f[newState]$. This algorithm is finding a path very quickly but generally speaking it's not an optimal path. What can be done to get a better path, is re-run the algorithm with the new and improved heuristic function that the previous trial has created. It can be proven that if the goal is reachable from any node, then the algorithm will find an optimal path after a finite number of trials. In our program that happened at a maximum of 20 trials depending on the input file.

Observations/Conclusion

If there is no way to find a heuristic function, then the best way to go about finding an optimal path is UCS. If the ability to find such function exists, then there are plenty of options to try and find an optimal solution. The best option that one has depends on the "structure" of his graph, for example the graph could be in a form of a grid, polygonal map, road map as in our case etc. If we were to compare IDA* with LRTA* in our example, we can safely say that running LRTA* until we converge to an optimal path is the fastest way to go. We can also say that running IDA* with a bad heuristic is by far the slowest algorithm of the ones we tried during this project. If we're trying to optimize running time and we don't care about the space complexity, we would either go with A* after computing a "solid" heuristic function or we would run LRTA* until convergence.

Additionally

For the purposes of experimenting, as well as understanding the capabilities as well as the flaws of our program, we created a program under the name "**randomInputGenerator.cpp**". By doing so, we were able to experiment with different inputs, by changing some key parameters such as the number of roads, the number of intersections (nodes), p1, p2, p3, etc. We will be submitting this program with the rest of our assignment.

Sources

- General understanding of search algorithms: Lecture slides of PLH311
- Adjacency matrix implementation: `https://www.programiz.com/dsa/graph-adjacency-matrix`
- Deepening our understanding on A* search: `https://www.redblobgames.com/pathfinding/a-star/introduction.html`
- Understanding heuristic functions: `http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#S7`

- Optimizing Dijkstra into UCS: http://www.aaai.org/ocs/index.php/SOCS/SOCS11/paper/viewFile/4017/4357
- IDA* search: https://en.wikipedia.org/wiki/Iterative_deepening_A*
- LRTA* search: https://arxiv.org/pdf/1110.4076.pdf