Horia “Ryder” Stancescu
Hstances - 0721385

CIS*3700 Assignment 2

Part A:

1. *Design a program that used a brute force algorithm to solve puzzles in a minimum number of steps. Describe how your program operates.*
   bruteforce.py
   I have constructed the brute force algorithm in as efficient a manner as I could come up with. The program uses an iterative approach to the algorithm utilizing two lists of nodes, the first list is used as the current list of nodes to traverse, the second list is a loop optimization list which keeps track of all Unique states visited. The program works by constantly taking the first node in the list regardless which node is best thus this uninformed node selection is what gives this greedy algorithm its brute force properties. When a node is picked we check to see if it’s the solution state if it is, and we are told not to continue than return the path and end the search otherwise before any operation is performed the node is cross-examined with the list of unique nodes to see if we have already visited a state with this configuration, if we have skip this node, if we have not acquire all of this nodes neighbours and stack them at the end of the list storing current nodes to visit. By taking note, since the neighbouring nodes are always tacked on at the end this Bruteforce algorithm functions almost exactly like a breadth first search algorithm thus guaranteeing the minimum number of steps, the loop removal also aids in this fact.

2. *Design a program that uses a best-first-search algorithm to solve the puzzles in a minimum number of steps.*
   bestfirst.py

   This program functions exactly the same as my bruteforce algorithm with 1 change needed to the core algorithm. Instead of always popping the 1st node, instead perform a heuristic analysis on all the nodes that need to be visited and choose the one with the best heuristic best in this programs case means lowest. By popping from a heuristic defined index, the algorithms performance does not degrade on its own but the heuristics applied are what determine whether or not it’s performance will be improved or not.

3-5. *Implement two separate heuristics for your best-first-search algorithm, provide a convincing argument that each heuristic is admissible, and provide a convincing argument that one of you heuristic gives higher admissible estimates than the other.*

For a heuristic to be admissible it must always estimate a cost that is lower than or equal to the actual cost of reaching the goal state [1]. As such I have constructed my first heuristic to be extremely simple an optimistic, the way the heuristic works is given a node it will analyze it’s state and judge how far the red cars current position is from its goal position, the farther the red car is the worse the heuristic, the closer the better. If a board is created in such a manner that a red car is the only car placed, the heuristic will always determine a cost equal to moving the car directly to the goal, if there’s an obstruction the same metric is determined thus although in accurate it underestimates the cost every time thus proving it’s admissibility. This heuristic was created with the rationale that the red car is the only car who's position is known in the goal state, thus by referencing this car a heuristic can be determined, however in cases where the red car must backtrack such as to allow a gap to be created, these moves will be checked last, there will also be numerous cases where large numbers of states have the same heuristic value due to only the red car having weight in the calculations. These pros and cons show that although the algorithm is effective in its simplicity, there is much room for improvement.

The second heuristic builds upon the first, it functions by attempting to reposition all the cars in a "possible" goal state. This is achieved by taking the start state the cars are in, and sliding them around until a possible solution is achieved, this does not give an answer as it does not care about cars passing through each other, if a possible state is invalid it continues to check until a valid one is achieved or no such answer exists, a value of 0 is returned in this case. Due to the requirement of needing this "possible" solution state, it requires that it is calculated ahead of time, in my algorithm rushhour.py two fields are used to tell the program you wish to setup heuristic 2 and use it via the "resetH2" flag for initializing the required setup, and when running the bestfirst search, passing in the argument of '2' into the search as 1 is default. Now the difference between this state and the above simpler version is instead of calculating the distance of the red car to its solution state, every car has its distance from its original to its possible answer state calculated to provide a heuristic value which takes the entire board state into the calculation. This heuristic is also admissible as it will always estimate based upon closeness the current state is to its counterpart answer state, meaning that the search algorithm will never be able to complete the game in less moves than the algorithm prescribes as the only event this could occur in is if a car was positioned upward instead of downward, but due to the algorithm taking this exact path based on the heuristic itself, a contradiction such as this will not occur and the answer state will always be either less than or equal to the number of moves needed to reach the goal state.

Due to the complexity of the second heuristic as it takes in more values, it is quite evident that this heuristic results in higher admissible estimates than the other, as heuristic one can only be perfectly accurate in the event of a board with only the red car, whereas this improved version takes in the entire board and thus results in higher results due to less states sharing the same heuristic values. Of course this heuristic's actual performance is subject to change based upon the provided "possible" goal state, however improving this algorithm to do such that outweighs the benefit of this assignment, and its current performance is enough at assessing its solutions. Although the results are a bit off on this heuristic I do believe with a few modifications and performance tweaks it will outperform heuristic 1 easily.

Note: The pre-setup stage prints text corresponding to the combinations it has attempted, this was left in as an indicator that the program is running and not halting during this recursive step this is created on line 447 and 71 should you wish to comment it out.

6. Comparing Search Space for all 3 algorithms.

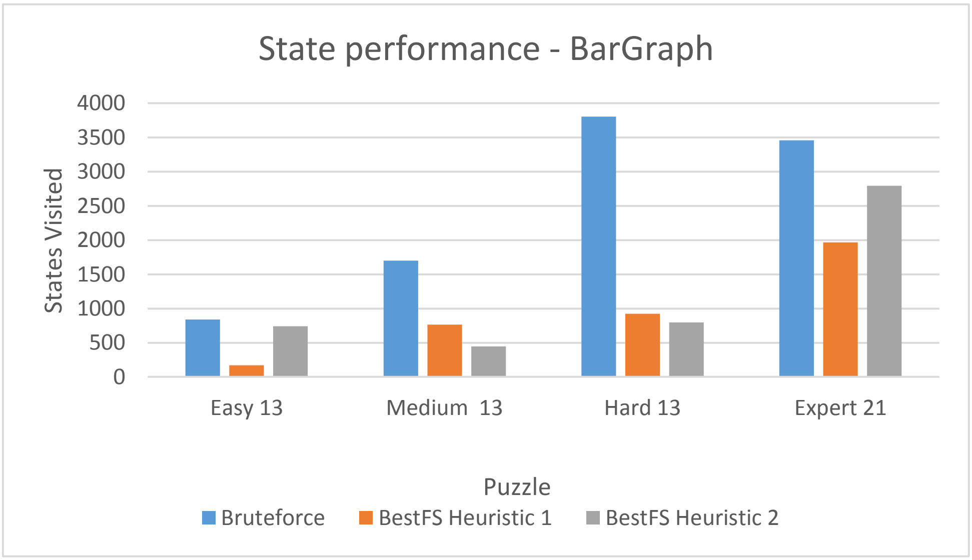
To test this, puzzles of varying difficulty have been chosen from http://www.thinkfun.com/mathcounts/play-rush-hour to test the search space of each algorithm.

Table for Each puzzle difficulty type and each algorithms performance on said type are shown below:

Puzzles vs Algorithm

Puzzle	Bruteforce-States	Bruteforce-Depth	Bestfirst_H1-States	Bestfirst_H1-Depth	Bestfirst_H2-States	Bestfirst_H2-Depth
Easy 13	839	18	170	18	741	20
Medium 13	1699	29	766	29	447	31
Hard 13	3803	40	921	40	799	57
Expert 21	3460	62	1966	62	2795	69

As mentioned above the Bestfirst only performs as well as the heuristic calculation it’s associated with. For heuristic two interesting results occur, it would appear that the fewer cars that are on the board the easier it is to create a potential board answer state, and as such the longer it will take for the heuristic to reach that answer(as mentioned in the Goal state vs actual answer state above).



Here we can observe quite clearly that Heuristic 1 and 2 always outperform the bruteforce algorithm in regards to state visits, however in this sample size heuristic one visits less states than heuristic 2 even though heuristic 2 takes more information to process its heuristic score.

Another observation would be the time complexity required for the BestF_H2 to solve a given grid/board. Although the algorithm checks fewer states on most occasions, due to the distance calculations performed per node, it is noticeably slower than the other implementations (Time estimates shown below provided in log.txt). Another observation would be how the Second heuristic takes unnecessary extra steps to reach a solution state, this is due to it trying to match its target state as opposed to the closest answer so depth levels have a tendency of differentiating, note that the unnecessary steps are not caused by looping as repeat states are not allowed in the algorithm. Time Stamps for the Algorithms Plus setup will be posted below to give a better understanding as to the performance per runtime.

Runtime Algorithms vs Puzzles

Puzzle	Bruteforce	Bestfirst Heuristic 1	Bestfirst Heuristic 2
Easy 13	1 Seconds	1 seconds	1min 13seconds
Medium 13	5 Seconds	8 seconds	42 seconds
Hard 13	19 seconds	7 seconds	1min 2 second
Expert 21	16 seconds	33 seconds	10min 19 seconds

As shown above the computational task performed by each heuristic is reflected here in the runtime of the algorithm. It would appear in order for the calculation to pay off in regards to speed, either a larger search space is required to filter through or a slower node traversal navigation.

<u>Part B:</u>

1-2. Use your code from PART A, above to develop a program that evaluates the difficulty of a RushHour puzzle.

For this I have basically taken the output of the bruteforce algorithm as it performs at a drastically faster speed (less calculations due to no heuristic) since all search space will need to be analyzed prior to attaining a puzzles difficulty. The difficulty of a puzzle is determined based on the Search Space, the amount of total solutions within that search space, and the depth length of the shortest solution. The logic behind this is to use the shortest depth length as a base estimate of the problems difficulty, the assumption being in order for a human to solve a puzzle the puzzle in a minimum amount of steps is typically dependant on the number of moves the individual can predict ahead, the larger the number the less likely an average human can find the task trivial. As an average person can remember on average up to 7 items in working memory the path is divided into groups of 7, every two groups of seven will act as a difficulty from easy to expert. Since there is the possibility of multiple paths to an answer state this will allow individuals who’ve made a mistake in one non-looping path to still find a solution to another. Since there is no accurate way to decide on a reasonable value for answer ranges, the above pre-set difficulties used at testing the algorithms will be used to determine reasonable answer ranges per difficulty.

The way to accurately determine the difficulty, utilizing the above ranges of: minimum depth Difficulty, number of Solutions Difficulty, and Search space difficulty is quite simple, calculate the likelihood of a puzzle being solved via chance by taking the number of solutions and divide by a subsection of the search space. Since it is impossible to have 100% solvability via randomness even if the red car is positioned exactly on the answer due to a possible branch of moving away from the answer resulting in a 50/100 result. The search space is first divided by 2, and this value is then divided into 3 areas to determine the difficulty via arriving at the solution from pure chance. Therefore here are the proposed ranges of difficulty where ‘x’ represents the calculated randomness value:

3 Hard: 0 > x <= 16.6%

2 Moderate: 16.6% > x <=33.26%

1 Easy : 33.26% > x <=50

Once this value is calculated simply average between this resultant difficulty score (Easy=1…Expert=4) for the randomness value and the shortest depth, to achieve the puzzles estimated difficulty.

Example Results from the above trials:

Easy 13:

States Visited : 1083
Total Solutions: 213
Minimum Depth: 18

Randomness Calculation = 213/1083 = ~19% = Moderate
Depth Calculation = 18/7(7 items in 1 chunk of working memory)
= 2 / 2 (2 chunks per difficulty)= 1 = Easy
(1 + 2)/2 = 1 Therefore this puzzle would be categorized as Easy.

Medium 13:

States Visited : 1798
Total Solutions: 4
Minimum Depth: 18

Randomness Calculation = 213/1083 = ~002% = Hard
Depth Calculation = 31 / 7 = 4/2 = 2 = Moderate

(3 + 2)/2 = 2 = Puzzle Difficulty is Moderate

Hard 13:

States Visited : 7336
Total Solutions: 2016
Minimum Depth: 41

Randomness Calculation = 4/1798 = ~27% = Moderate
Depth Calculation = 41 / 7 = 5/2 = 2 = Moderate

(2 + 2)/2 = 2 = 2 Puzzle Difficulty is Moderate.

Expert 21:

States Visited : 6299
Total Solutions: 1698
Minimum Depth: 62

Randomness Calculation = 1698/6299 = ~26% = Moderate
Depth Calculation = 62 / 7 = 8/2 = 4 = Hard

(4 + 2)/2 = 2.5 = 3 Puzzle Difficulty is Hard

Utilizing this approach it is clear that in order for a puzzle to be easy it must both be solvable in a small number of steps and have multiple solutions, otherwise it's of moderate difficulty at best. These calculations can be shown and recalculated in Difficulty Analyzer.py, simply alter the puzzle configuration to receive the results. As an aside in the future I believe more accurate results can be achieved if a larger difficulty spectrum were given to fill in the grey areas between Easy-Moderate, and Moderate-Hard, as puzzle medium 13 would be calculated as Moderate-Hard in this regard, and easy 13 would be Easy-Moderate. This problem will naturally exist at any gradient used for a difficulty system however in my opinion covering these in between ranges would have given more precise results at informing the player the degree of difficulty used thus a range segmentation of 5 instead of 3 might be preferable, but that is solely my opinion after performing this analysis.

3. Is the difficulty of a puzzle only related to the number of steps that it takes to reach a solution? Is it possible to make an "Easy" puzzle that takes many steps, or a "Hard" puzzle that does not take as many?

As explained above there are more factors than just "number of steps" to accurately determine the difficulty of a puzzle. For example given a puzzle that has a depth of 31, and a grid that is setup so the player only ever has 1 decision to make and it leads to a solution state, is the puzzle difficult? Given the depth of only 31/7 = 4/2=2 would equate to a difficulty of Medium as it requires 3x the average person's working memory to chunk the moves (at least in theory), however the puzzle itself is trivial. It's also just as feasible to derive a puzzle that has a small depth but a large search space, resulting in a potentially complex puzzle meaning a potentially Hard difficulty. Finally given a puzzle of any difficulty it's also equally likely that it is easy to solve if there are numerous ways to reach a solution state, meaning if the puzzle were given to a child tasked to move pieces in random order, it is possible the child could solve a previously deemed "Expert" puzzle in a small manner of time on luck alone due to the probability of having a large pool of solution states to choose from. That is why all these 3 components are used in the above algorithm for deciding difficulty.

References.

1) http://en.wikipedia.org/wiki/Admissible_heuristic