Ayush Kadakia (ank102)
CS 440: Introduction to Artificial Intelligence Assignment 1

**Part 0:**

The gridworlds are generated by creating a Grid object and initializing its grid attribute with 30% of the tiles being blocked and the rest unblocked, with the top left and bottom right tiles being unblocked. In every game, the agent moves from the starting top left to the end bottom right tile.

Each of the created grids are then checked by the Breadth-First Search algorithm, which verifies that the path between the top left and bottom right tiles really does exist. If not, regenerate the grid. BFS works by first initializing an empty queue and a visited set, and, starting from the top left tile, continually exploring the neighboring unvisited tiles that aren't in the visited set, until the queue is empty (no path) or the bottom right tile is reached (path exists). The BFS search was chosen as it seemed to be the most straightforward to implement, despite not being the most efficient in terms of time complexity, as the specified problem did not require a more efficient solution. A more sound choice of search would have been, for example, the Depth-First search, as, unlike the BFS search which returns the shortest path, DFS simply returns any path that it has first reached. Since we only had to verify that A path exists, not that THE shortest path exists, DFS would have been a better choice.

After all 50 grids are created and verified, each of them is passed as a parameter to the ForwardA and BackwardA search functions, their performance (in terms of total time taken and number of cells expanded) is returned. The output 50 grids with the marked path are written to corresponding files in the form of (0 for unblocked, 1 for blocked, and " " spaces for marked taken path tiles). Finally, the average times and number of expanded cells are printed to the console.

**Part 1a:**

Given that the agent does not know which cells are initially blocked, the agent will move to the east first. This is because the agent will aim to take the shortest path to the target; the shortest path here when disregarding the blocked tiles is to move three spaces to the east. This is not possible as the tiles are blocked, however the agent not knowing this will initially try to move three spaces east. Therefore the first move the agent will take is a step east.

**Part 1b:**

An agent in a finite gridworld will indeed either reach the target or discover that it is impossible within a finite time. It is key for this example that the condition of a finite gridworld holds. It is also important to note that the agent only has four moves that it can make, which is to traverse a unit north, east, south, or west. Eventually, in a completely unblocked gridworld the agent will be able to explore all options in order to reach the target in finite time, given that it has a limited number of ways to move in a finite grid. If there are blocked cells, then the agent will

remember it cannot explore those cells and keep trying different paths to reach the target. Eventually, the agent will keep trying different paths to avoid blocked cells and amending its strategy until a viable path is found; this will once again occur within a finite time due to the finite nature of the gridworld, and limited options the agent has to move. In other instances, the agent may reach a point where it is in almost of a "stalemate"; the agent will keep trying to find paths only to realize that the path is hindered from blocked cells. Eventually, the agent will come to realize that all possible paths are not viable due to being blocked by cells which cannot be traversed. Eventually as it runs out of options, the agent can conclude no path to be found. In short, the agent will extensively try all potential paths and be able to determine within a finite time if there is a feasible path to the target, or if no potential path is viable due to blocked cells being in the way; this is all attributed to the key properties of the grid being finite, and the agent only being able to move north, east, south, or west. The number of moves of the agent until it reaches the target or discovers that this is impossible is bounded from above by the number of unblocked cells squared. This can be proved using a proof by contradiction. For the sake of the contradiction, let us say that the number of moves (represented by M) to reach the target or realize it's impossible is greater than the number of unblocked cells squared (represented by U); this can be represented by the inequality $M > U^2$. The maximum number of moves an agent can make per unblocked cell is 4, which is going through it up, down, left, and right, until all possibilities are exhausted; this logic leaves us with the inequality $M \leq 4U$. This contradicts the initial inequality we put forth, that $M > U^2$. For instance, in a 3x3 grid with all cells unblocked, these inequalities would say that $M \leq 4*3*3$ and $M > (3*3)^2$. The maximum number of moves cannot be less than or equal to 36 and greater than 81 at the same time. Therefore this is a contradiction in itself. Resultantly, we must deny the initial contradictory assertion that the number of moves to reach the target or realize it's impossible is greater than the number of unblocked cells squared. In the end, we have proved that the number of moves of the agent until it reaches the target or discovers that this is impossible is bounded from above by the number of unblocked cells squared.

**Part 2:**

Both the Repeated Forward A* with a favor for large g-values and the Repeated Forward A* with a favor for small g-values during tie-breaks, were implemented as a single function that takes a boolean favoSmallG parameter, that is used to determine the g-value tie strategy. In both cases, the f-value is compared first, and only afterwards, if f-values are equal, g-values are taken into consideration. The ForwardA function made sure to use the binary heap as the open, a set as a closed list, and the Manhattan distance as the heuristic function. Visited neighbor cells are skipped. When the agent reaches the end, the resulting path is reconstructed by following the chain of parent cells from end to start and reversing the direction. The path, together with the total number of expanded cells, is then returned.

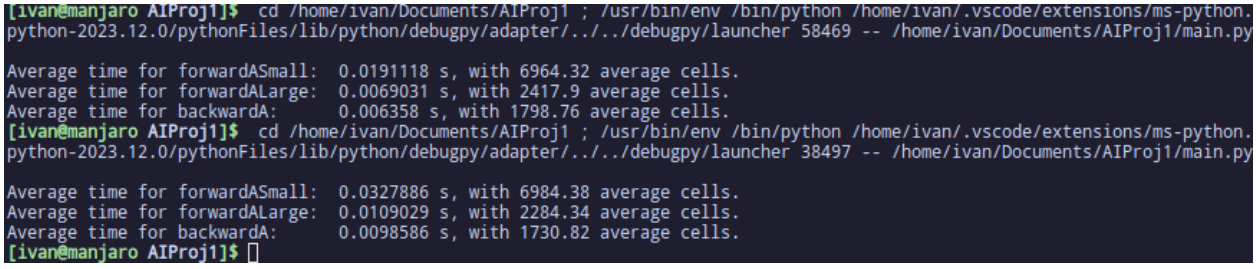
The two versions of the Repeated Forward A* searches had significant differences in performance. Over the course of several runs (each with 50 101x101 grids), the Forward A*

search with a favor towards small g-values showed itself as 3-4 times slower both in time and the number of expanded cells. For example, in one run, the average time for the small-g favored search was 0.0191118s with 6964 cells, while the average time for large-g favored search was 0.0069031s with 2417 cells (both searches performed on the same set of 50 grids).

The reason why favoring larger g-values is better is because, considering that f(s) = g(s) + h(s), choosing lower g(s) increases h(s), which is the Manhattan distance heuristic, basically, making the heuristic more larger. Higher g-values, which represent the distance between the start and the current state, are associated with those states that are closer to the end. Considering that the closer to the start the agent is, the more cells will have to be expanded in total, it makes sense to expand the least amount of cells near the start as possible, thus preferring higher g-values that are closer to the end target cell.

**Part 3:**

The Repeated Backward A* search was implemented with the consideration of a favor for large g-values during tie-breaks with same f-values. Several experiments, all with 50 101x101 grids, were run. This time, the difference between the Backward A* and Forward A* searches, both with a favor for larger g-values, was not as significant. Still, Backward A* showed itself as consistently 10-20% more efficient in the number of cells expanded and 0-10% more efficient in the average run time than the Forward A* search. For example, in one of these experiments, the average time for Backward A* was 0.0098586s with 1730 cells, while Forward A* ran for 0.0109029s with 2284 cells. The direction of the search influences the efficiency of the search because, in the reverse direction of the search, the cells towards the end target cell are expanded first before those towards the start cell. Considering the larger g-value tie-breaking some states wil g(s) + h(s) <= g(end) will end up not being explored at all. The attached screenshot below shows the console output for the described examples in Part 2 and Part3.




**Part 4:**

Manhattan distances are indeed consistent in gridworlds in which the agent can move only in the four main compass directions. This can be proved through the triangle inequality. The triangle inequality states that the sum of two sides is greater than or equal to the length of the remaining side. Manhattan distance is the sum of the difference between x and y coordinates of 2 points. With the definition of consistency, we would know that $h(n) \leq c(n,a,n')+h(n')$. Because

h(n') is already the target node, we can equate this zero which leaves us to prove $h(n) \leq c(n,a,n')$. This represents that the Manhattan Heuristic is less than or equal to the true cost of going from agent to target. As the Manhattan distance is the sum of difference of x & y coordinates, this would equate to the exact number of moves (up, down, left and right) in an unblocked grid. In a blocked grid, the heuristic's properties would still stand under the same principle. Because of the triangle inequality, consistency further stands as the remaining side, h(n) is less than or equal to the sum of the other 2 sides. Therefore, this function would not overestimate the cost of going from agent to target. As a result, on the basis of the triangle equality, the Manhattan distance is consistent.

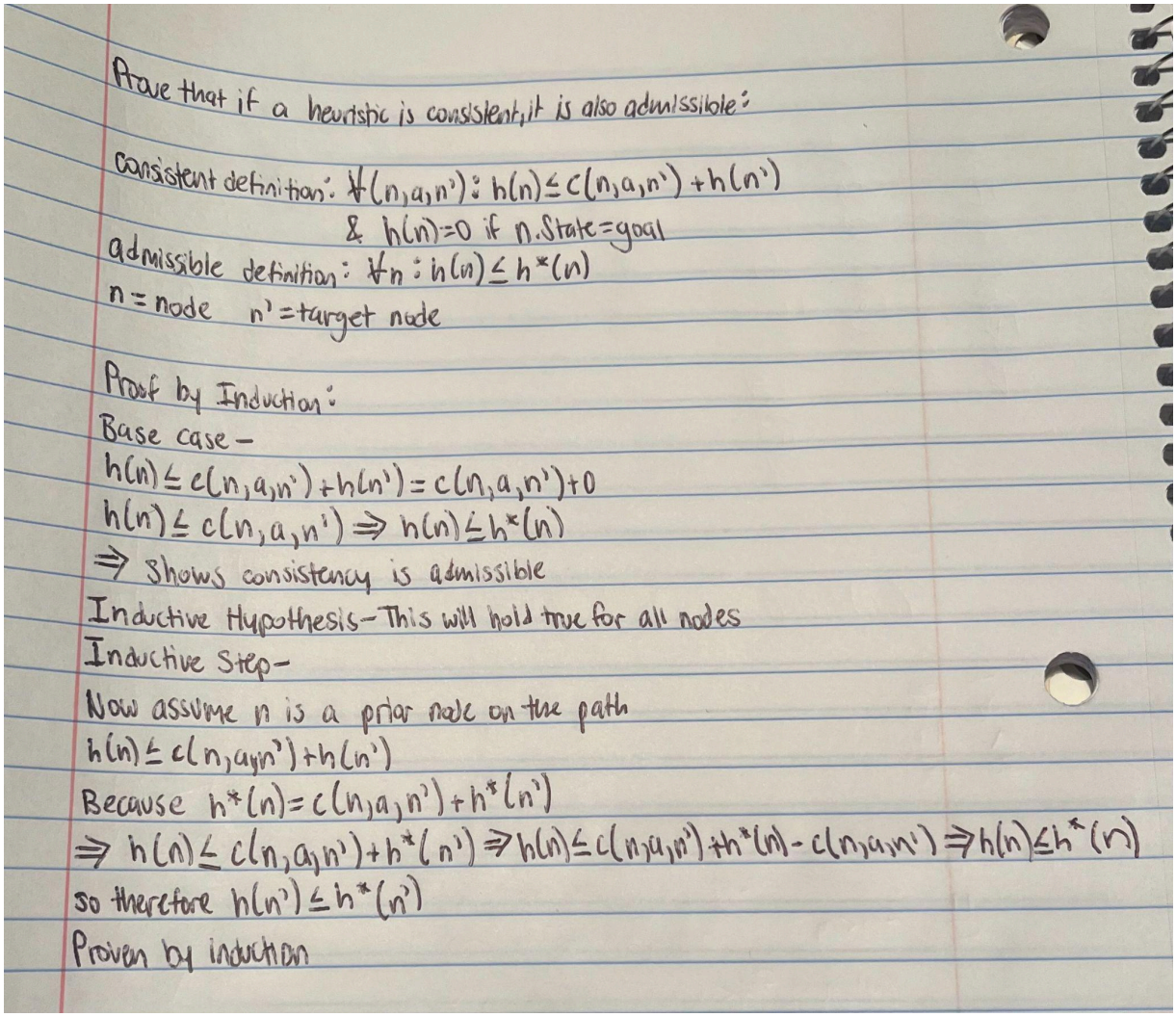
**Part 5:**

Adaptive A* search is anticipated to have a faster runtime than Repeated Forward A* search with the blocked cells in the gridworld. Adaptive A* search can adjust when blocked cells are found, which can result in a viable path to the goal being found faster. Repeated Forward A* search is less effective in this scenario, as many different runs of A* search may be necessary to find a path that works; this would result in a slower run time when compared to Adaptive A* search. As adaptive A* search uses its experience with earlier searches in the sequence to speed up the current A* search, it is better equipped to handle gridworld problems with blocked cells.

**Part 6:**

To compare the two algorithms, we can use a Statistical Hypothesis Test. In our test, we can set our null hypothesis by assuming that both algorithms have no significant difference. After that, we can select one statistical hypothesis test to prove that one of the algorithms works faster than the other.. In this case, a t-test can be used for the purpose of this project.. For the t-test, we will run both algorithms several times to collect a large amount of data to obtain enough samples of data, and get more accurate results. After collecting data, we can use the t-test to obtain the probability (or p-value) of this test. If the p-value obtained from testing both algorithms is less than 0.05, we will reject our initial hypothesis, and we will conclude that there is a significant difference between both algorithms. On the other hand, if the p value is greater than 0.05, we will conclude that there is no evidence to claim a significant difference.

**Extra Credit:**




An example of a heuristic that is admissible but not consistent includes the sliding block puzzle example we mentioned in class. When counting how many tiles are out of place from the goal or using Manhattan Distance as a heuristic, this can be seen to satisfy admissibility, but not be consistent through a violation of the triangle inequality.