Part 1: Understanding the Methods [10 points]: Read the chapter in your textbook on uninformed and informed (heuristic) search, and then read the project description again. Make sure that you understand A* and the concepts of admissible and consistent h-values.

a) Explain in your report why the first move of the agent for the example search problem from Figure 8 is to the east rather than the north, given that the agent does not know initially which cells are blocked.

When the program first starts, all the information given to the computer is that in Figure 8, the agent is in the E2 box and the target is in the E5 box. When creating the algorithm, the computer is going to initially pick the direction with the lowest h-value and then calculate each direction's h-value before moving. If the agent wanted to move west, the h-value would be 4 (4 blocks away from T); if the agent wanted to move north, then the h-value would also be 4, but if the agent moved east, the h-value would be 2, which is why the program's first move is east.

b) This project argues that the agent is guaranteed to reach the target if it is not separated from it by blocked cells. Give a convincing argument that the agent of infinite grid worlds indeed either reaches the target or discovers that this is impossible in finite time. Prove 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.

If the agent of an infinite grid world were to figure out a path to its target with a variety of blocked cells, the agent either finds the path or discovers it is impossible in finite time. By utilizing the A* algorithm, we are exploring potential paths within a finite search space, which will either lead to a path or the impossibility of one. The number of moves necessary for the agent to reach the target is capped by the square of the number of unblocked cells, which would be the upper limit of exploring the grid through the x and y axes. Regardless of the infinite-size grid, the number of unblocked cells guarantees the agent's path within a finite time. It will also provide a lower estimate of the goal of the search method in an infinite environment.

Part 2: The Effects of Ties [15 points] Repeated Forward A* needs to break ties to decide which cell to expand next if several cells have the same smallest f-value. It can either break ties in favor of cells

with smaller g-values or in favor of cells with larger g-values. Implement and compare both versions of Repeated Forward A* with respect to their runtime or, equivalently, the number of expanded cells. Explain your observations in detail, that is, explain what you observed and give a reason for the observation.

With the program I have created in Python, to be consistent when switching between the smallest f-value and larger g-value, I would have to import time to get the relative runtime for each method of favoring different g-values. Originally, I hypothesized that favoring smaller g-values would be more efficient and take less time than favoring larger g-values. I thought this was because, by expanding the larger g values, you are essentially spending more time searching for paths that are unnecessary while finding the shortest path. However, after implementing both algorithms and the runtime calculator, favoring larger values will result in a faster runtime. For example, for the first time, I ran the program while finding no path to the Agent to Target; the smaller g-values runtime ran at 0.07793211936950684, while the larger g-values runtime ran at 0.010941028594970703. After running it again with a more complicated path, smaller g-values concluded at 4.220008850097656e-05, while larger g-values ended with a time of 3.695487976074219e-05, which is over.1 of a difference. I believe that there are many factors that led to larger g-values having a shorter runtime, such as pruning suboptimal paths and efficient exploration. By prioritizing larger g-values, the computer is able to eliminate paths early in the search process, leaving a more direct exploration for the agent to target. If the program favored smaller g-values, the computer is not going to hold information about the suboptimal paths, so it would most likely explore the paths that are not needed, creating a longer time complexity. Another reason would be that larger g-values will guide the algorithm to explore paths that are more promising first. If the program favors the smaller g-values, the direction the algorithm is going most likely has a blockage, preventing the path towards the target and making it backtrack and waste time. By utilizing the larger g-value favors, it is most likely to discover the optimal path faster while limiting the chance to bump into a blocked cell.

Part 3: Forward vs. Backward [20 points] Implement and compare Repeated Forward A* and Repeated Backward A* with respect to their runtime or, equivalently, the number of expanded cells. Explain your observations in detail, that is, explain what you observed and give a reason for the observation. Both versions of Repeated A* should break ties among cells with the same f-value in favor of cells with larger g-values and remaining ties in an identical way, for example, randomly.

While implementing both Repeated Forward A* and Repeated Backward A*, I realized that runtime was not greatly affected; however, it depended on the blocked cells to guide the target from agent to agent and target to agent. Although both A*s will result in the same smallest path from agent to target, the actual performance and efficiency of each algorithm depend on how far the target is from the agent. If the target is far from the agent, there could be more of a variation in the path the algorithm takes due to the difference in blocked cells. By running the algorithms multiple times with different grid configurations and agent and target positions, I can conclude that the number of expanded cells varies greatly towards longer paths than shorter paths in my program. For example, by importing time, creating variables before and after the call to each method, and subtracting the end time from the start time, I can get the general basis for both runtimes, forward and backward. Running the methods one time shows that the repeated forward A* runtime is 7.605552673339844e-05, while the backwards A* runtime is 3.504753112792969e-05. This means that backward A* is a little faster than forward A* due to the barriers each algorithm has to go around and the path it decides to take. However, when I run the program again, forward A*'s runtime is 0.00011992454528808594, while repeated backward A* is 4.792213439941406e-05, concluding that forward A* is faster. The direction at which Repeated A* starts from does not affect the runtime as greatly; it relies on the obstacles in the way and the perspective from which the computer sees it to determine how long it will take to reach the target or agent.

Part 4: Heuristics in the Adaptive A* [20 points]: The project argues that "the Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions." Prove that this is indeed the case.

Furthermore, it is argued that "the h-values hnew(s) are not only admissible but also consistent." Prove that adaptive A* leaves initially consistent h-values, even if action costs can increase.

In a gridworld where the agent can only move in the four main compass directions, the calculated heuristic from each cell to the target calculates the shortest path without overestimating which diagonal movements would do. Furthermore, the actual cost to reach the target cannot be less than the Manhattan distance. Therefore, h(s) will always be less than or equal to the true cost to reach the target, making the restriction of only 4 directions admissible. To prove that adaptive A* leaves initially consistent h-values, even if the action cost increases, we have to look at how A* adapts to its h-values. The adaptive A* algorithm updates its h-values only when the node's actual cost is less than the estimated cost. That said, the heuristic function ensures that the estimated cost will never decrease as the agent progresses to other nodes. Adaptive A* corrects the inconsistency by updating the h-values when necessary to ensure consistency through the algorithm. For example, let's take into account the Manhattan distance. As the agent explores the gridworld, the cost of moving from cells to the target would be higher than expected due to the obstacles labeled "X" in the way, creating inconsistent h-values. Whenever there is a blockage from the shortest h-value from A to T, the adaptive A* detects the violation and increases the cost by going around the X. This ensures that the heuristics remain consistent even if action costs increase or if the environment creates blockages.

Part 5: Heuristics in the Adaptive A* [15 points]: Implement and compare Repeated Forward A* and Adaptive A* with respect to their runtime. Explain your observations in detail, that is, explain what you observed and give a reason for the observation. Both search algorithms should break ties among cells with the same f-value in favor of cells with larger g-values and remaining ties in an identical way, for example, randomly.

After implementing Repeated Forward A* and Adaptive A* in my program, I am able to conclude that Adaptive A* has a faster runtime. In my Python program, I am able to import time, which can help determine what the runtime for each method actually is, and I utilized it by creating a variable that would get the time before and after the A* method and subtract the finish to start to get the initial runtime of each runtime. After running the program with “No path found," the Repeated Forward A* runtime evaluates to 0.026778936, while the Adaptive A* runtime evaluates to 0.0002532005. After running the program again for a path that actually exists, the Repeated Forward A* runtime concluded at 7.605552673339844e-05, while the Adaptive A* runtime was 4.8160552978515625e-05. The repeated forward A* is not as fast as the adaptive A* algorithm because the adaptive A* adapts its heuristic functions based on the information gathered during the search, which helps with the decision-making of the program and creates a faster way to determine the shortest path. With more information that the computer holds, it can reduce the search space and eliminate paths that the Repeated Forward A* might go through.

Part 6: Statistical Significance [10 points] Performance differences between two search algorithms can be systematic in nature or only due to sampling noise (the bias exhibited by the selected test cases since the number of test cases is always limited). One can use statistical hypothesis tests to determine whether they are systematic in nature. Read up on statistical hypothesis tests (for example, in Cohen, Empirical Methods for Artificial Intelligence, MIT Press, 1995) and then describe for one of the experimental questions above exactly how a statistical hypothesis test could be performed. You do not need to implement anything for this question; you should only precisely describe how such a test could be performed.

One statistical hypothesis test that determines whether they are systematic in nature is Cohen’s h. This test measures the difference between two probabilities. An example would be measuring a value to see if it is small, medium, or large. If h = 0.2, it is a small difference; if it is 0.5, it is a medium difference; and if it is 0.8, it is a large difference. Cohen’s h finds the average of all the test statistics of two different test cases, subtracks the average, and divides it by the population standard deviations. One way we can implement this method in the code is through repeated forward A* and adaptive A* with respect to their runtime. First, we should calculate the mean of the runtimes of Repeated Forward A* and Adaptive A* after 10 runs. Then we would have to find the standard deviation in multiple steps. First, find the mean of both Repeated Forward and Adaptive, subtract the mean from each runtime, square each deviation, add the squared deviations, and divide the sum by the value 10 since we ran it 10 times. Finally, we take the square root of the result of the previous step. After finding the means of forward A* and adaptive A* and the standard deviation, you need to subtract the means and divide by the standard deviation. The number you get from that equation needs to be compared to the h value that I referenced before, where 0.2 is small, around 0.5 is medium, and 0.8 is large. This value is going to show the difference in the runtimes and greatly determine which algorithm is faster for the program. Let's say the value comes out to -0.98, which would conclude that forward A* is greatly faster than adaptive A*. With a value of 0.98, we can determine that adaptive A* is much faster than forward A*. The Cohen method is a great way to predict which algorithm should be used in the program to ensure the run-time is more efficient.