(a) Implementing and comparing the Repeated Forward A* and Repeated Backward A* search with respect to runtime was an interesting process. In the gridworlds that we tested both of them on, or in our code, the runtime for Repeated Backward A* search was constantly shorter but not by a lot. We measured in terms of nanoseconds and both algorithms are ran and timed for a 100 times total. Repeated Forward A* Search would usually come up with something like 136501731 nanoseconds and Repeated Backward A* search would finish after 1771580 nanoseconds. (Could be target started with a better path, will explain later)

Not a very noticeable difference, also, this allowed for us to observe what would happen if we changed how the gridworld was set-up. I personally asked a game-developer I know that has used these two algorithms for some of his games for "pathfinding". He told me and showed me that Forward A* and Backward A* does not differ much in terms of runtime or the number of expanded cells, but rather the runtime is dependant on the scenario of our gridworld. So, the more important function in terms of speed or runtime is actually the heuristics function that we use. A* goes with a mantra of taking the better or "closest" path with the least amount of cost to get there. However, there are cases where some video games or even our gridworld sets up a "trap" of some sort. Since it is done randomly, it is not every single time we run the search that there are gridworlds with "traps" in it. (Such as a 'corner' of walls, shaped like a reverse L.) The agent can end up hitting the middle of that reverse L, totally depending on how the A* search algorithm starts, and then hitting every single wall until it tries every path in that reverse L and then goes around it, when the agent gets there. (Assuming agent has no knowledge of gridworld).

So, to summarize, the algorithms, repeated forward A* and repeated backward A* cannot be compared in terms of time by themselves. Because of the heuristic algorithm we use, how A* search starts for either algorithm is more important. Again, if it leads the agent into a "trap" first, then it could be pretty screwed for time, however, if the agent gets sent in a "better" direction (no knowledge of grid, so it is random), the timing is a lot better.

© There are many, many examples of where the path found by Repeated A* is a shortest path, but not the optimal path from start to goal. In general, the argument for finding the optimal path is very controversial because in real life circumstances there is a variable that we cannot test in our code, which is travel time. (Delays, closed ways, etc.) An example for our Repeated A* is a graph or gridworld where there are weights to each path taken. A* algorithm will look for the next “closest” node to the target node (example A -4-> B -10-> D -11-> G), only 3 paths to G, however there could be a path (A-1->C-3->E-4->D-1->F-1->G), 5 paths to G. Depending on how the A* search sends out the agent to go search. Also, we have to consider sometimes weights on a path or edge are not equal to distance. They may be costs to go a certain way, so that means that the shortest path is not necessarily the optimal path. Another thing that we can consider are negative weights, however in A* search we would not get stuck in a cycle. And one last thing that I can think of where the shortest path may not be the optimal path is when self-loops and parallel edges are present inside a gridworld or graph.