Wyatt Plaga

For this assignment, I wanted to explore the differences between A* and Dijkstra's algorithm, as well as their applicability to pathfinding. In order to do this, I created both a small and a large graph on which to practice and test the implementation of the two algorithms. The small graph (Figure 1) contained 31 nodes and 120 edges. I created this graph by plotting the locations of my childhood friend's houses from growing up in Raleigh. I choose this as a basis for my graph two reasons. First, having a basis in my personal experience meant that the graph would reflect realistic scenarios in which an informed search would actually be useful, such as when using a GPS to go from one friend's house to another. Secondly, Raleigh has the interesting feature that the beltline divides the city into areas outside the beltline (OTB) and inside the beltline (ITB). This meant that there was only one edge connecting my OTB and ITB friends. Because this feature is challenging to a search algorithm, I thought it would be valuable in testing the quality of my implementations. As can be seen in Figure 2 and Figure 3, both the A* search and the greedy search explore to the right side of the graph before realizing that there is only one crossing to the left side. Greedy search is more susceptible to this issue then A*, as it will continue pursuing a path its begun even when it begins to venture far away from the goal node.

For the second graph, I attempt to randomly generate 10,000 nodes by placing them in a random location on a 300 by 300 map. If a node is already placed in the chosen random location, the node is not created. This results in the creation of 9400 - 9500 nodes. After node generation, each node picks a random point within a slowly growing radius. If there is a node on that point, the node forms an edge with it. This occurs with each node until that node has 5 edges connecting it to other nodes. This results in between 47,000 – 48,000 edges in the graph as a whole. This strategy causes nodes to tend to be connected to their closest neighbors, but allows some amount of randomness with which nodes they form edges. In this way, a unique and extremely large randomly generated map is achieved each time the large map is created. Figures 4-6 show three possible large maps that might be generated. Because of the computational limits, the nodes and edges themselves are only drawn once the path has been completed. During the pathfinding process, both the closed and open set are rendered so that the progress of the search can be visually observed.

Using these two graphs, I compared the number of nodes visited and runtime for Dijkstra's Algorithm and A*. As expected, A* was able to complete the search much more quickly while visiting much fewer nodes. For example, Figure 7 shows the search pattern of A* on a large graph, while Figure 8 shows the search patterns of Dijkstra's on the same graph. While A* directs its search towards the goal, Dijkstra's searches in a uniform area outwards. On the graph from Figures 7 and 8, for example, A* was able to find the goal node with 252 nodes in the open set and 1024 nodes in the closed set, while Dijkstra's had 195 nodes in the open set and 6050 nodes in the closed set. As the distance between the start node and the goal node increase, the number of nodes visited by Dijkstra's search will with the area of the circle it has to search, thus meaning the number of nodes visited increases with distance squared. It is important to note, however, that because the search radius often reaches the edge of the graph, this growth is often reduced. A*, meanwhile, will grow much more slowly, with its exact growth dependent on the quality of its heuristic. Experimental data using a linear distance heuristic produced the results listed in Figure 9.

This data suggests that, at least for the distances possible in the large graph, A* grows approximately linearly. Clearly, this is an enormous improvement over the time taken by Dijkstra's. A full comparison of the two algorithms is shown in Figure 10. In order to explore the impact that different heuristics had on the performance of A*, I created three different heuristics.

Wyatt Plaga

The first, admissible heuristic I created was simply the Euclidian distance from the node to the goal node. This is admissible since the distance from a node to the goal node will always be as long or longer than a straight line distance. The second, inadmissible heuristic I created was the Manhattan distance between the node and the goal node. While often admissible, it is possible for the Manhattan distance to overestimate the cost to reach the goal, such as a node directly connected to the goal node at an angle. The Manhattan heuristic would return a value of the x offset plus the y offset, which is more than the actual cost of the hypotenuse formed by those two offsets. The behavior of this heuristic was very similar to the first, although it sometimes explored fewer nodes than the first. While this increased the speed of the algorithm, such a search would have missed the optimal path had it gone through one of the unexplored nodes.

The third, inadmissible heuristic I designed returned the square of the linear distance. This heuristic nearly universally overestimates the actual cost to reach the goal, which creates behavior similar to greedy search. Because of this continuous overestimation, the A* algorithm was driven even more strongly than the greedy algorithm to seek whichever node was closest to the goal node, completely drowning out the consideration of the distance traveled from the start node. This resulted in consistently suboptimal paths, but allowed the algorithm to find a path in extremely short time periods. An example comparison of these three heuristics is shown in Figures 11-13. Figure 14 compares the time to complete the same graph for each heuristic, and demonstrates that the Manhattan heuristic visits less than half the nodes of the linear distance, while the distance squared visits hardly any.

For the path following demo, I created a four roomed area (Figure 14) with three obstacles (a square, a diamond, and a triangle) in order to demonstrate the ability of the boid to navigate the area. Nodes were placed densely at navigable points within in the environment to allow the boid to efficiently path find without complex obstacle avoidance behaviors. When a mouse press is registered, the node closest to the boid is determined, as well as the node closest to the mouse click. A* then computes a path between these nodes, and the boid uses the pathfinding algorithm discussed in class to navigate the path determined by A*. Figures 15-17 demonstrate this ability. Interestingly, using greedy search to plot the course for the boid did not result in unusual or inefficient looking behavior. In more complex environments, this might allow for faster movement calculations when optimality is not as important.

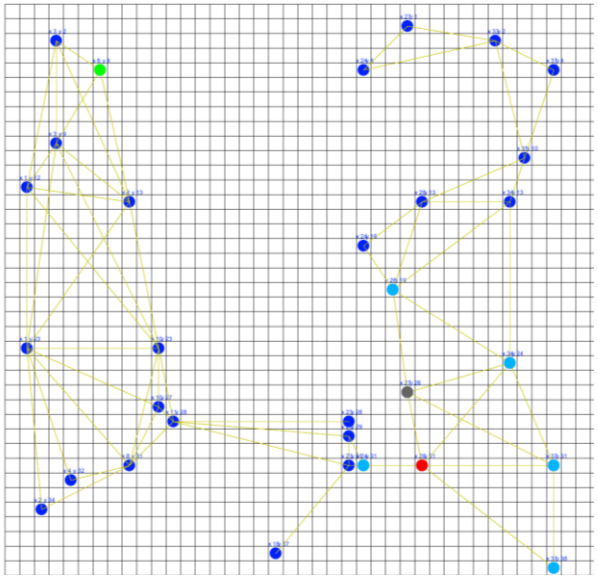
One interesting and unexpected side effect of using the Manhattan heuristic for pathfinding is that it will plot courses through the environment which involve several sharp turns. This is undesirable both because it takes longer to traverse the environment and because it causes an unsightly jittering of the boid. While, in this instance, efficiency and aesthetic quality are both improved by using the linear distance heuristic, this raises questions about whether a video game should consider other heuristics that promote aesthetics over efficiency. For instance, a better algorithm might be one in which the angle between the incoming and outgoing edge is considered, such that paths that require many sharp turns are discouraged. Because this could not be easily implemented simply by adding a new heuristic, an alternative strategy would be to come up with a list of several possible paths and then subsequently analyze them for sharp turns, returning the path with the least number of sharp turns. This would require several searches, however, and would prohibitively computationally intensive. One alternative method would be to set a threshold for the maximum amount of sharp turns that would be allowed to occur and cut off the search once a path that satisfies the cutoff if found. Another alternative would be to use greedy search instead of A* since it computes so much more quickly. While this would surely

produce many suboptimal paths, its significantly reduced computing time would make it a viable option for evaluating many paths to determine minimal turning.

Wyatt Plaga

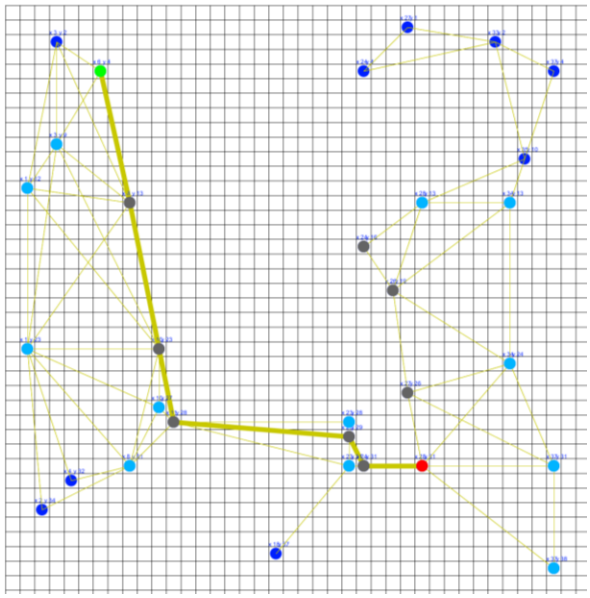
Appendix

Figure 1 (Small Map):



Light blue nodes are in the open set, grey nodes are in the closed set.

Figure 2 (A* Search):



Wyatt Plaga

Figure 3 (Greedy Search):

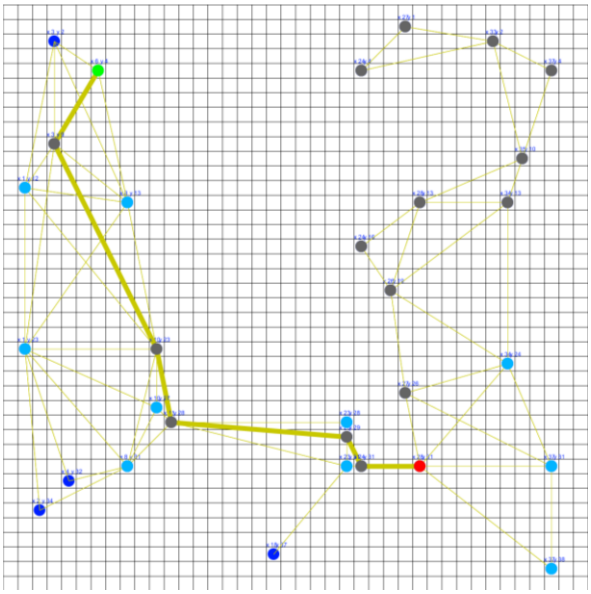
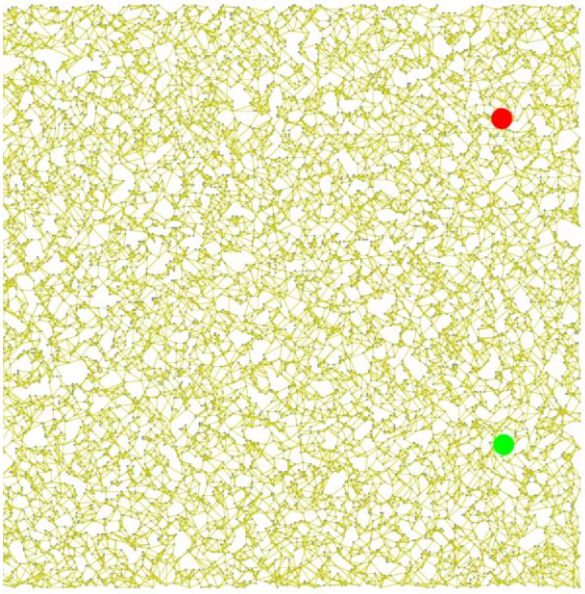


Figure 4 (Example Large Map 1):



Wyatt Plaga

Figure 5 (Example Large Map 2):

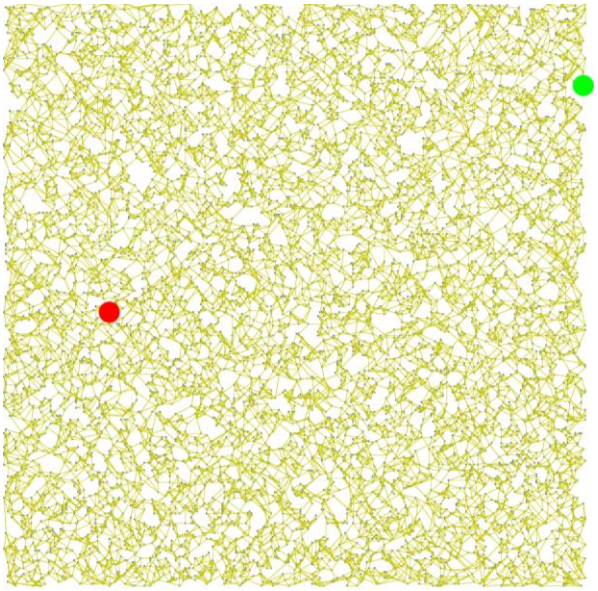


Figure 6 (Example Large Map 3):

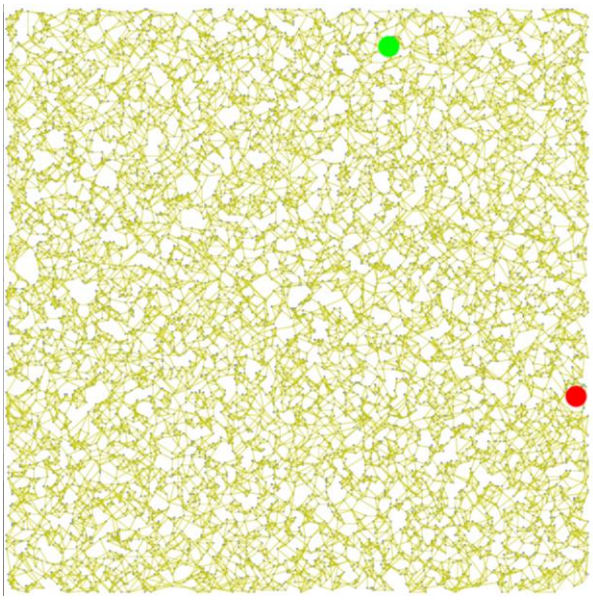
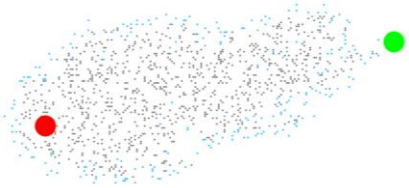


Figure 7 (Large Map A* Search):



Wyatt Plaga

Figure 8 (Large Map Djikstra Search):

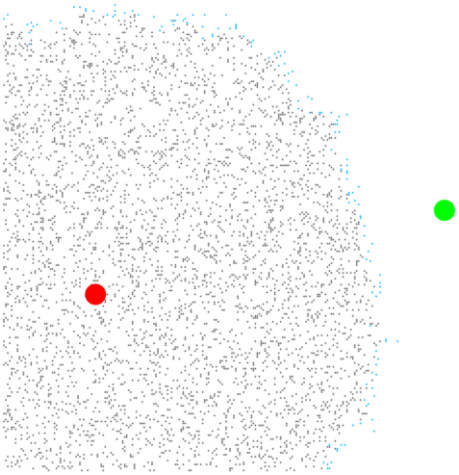
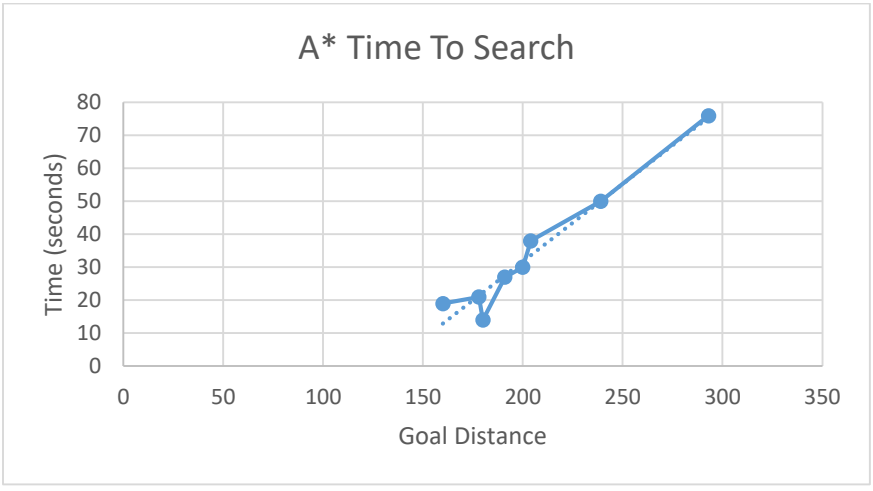


Figure 9 (Analysis of A* Search Time by Distance):

Distance	Time (seconds)
160	19
178	21
180	14
191	27
200	30
204	38
239	50
293	76



Wyatt Plaga

Figure 10 (Dijkstra's vs A* on Large Ma):

Distance	Time (Dijkstra's)	Time (A*)	Closed Nodes (Dijkstra's)	Closed Nodes (A*)
180	242	14	5177	622
191	361	27	6889	1148
200	388	30	6667	1199
208	429	43	6494	1378
242	452	29	7633	1157

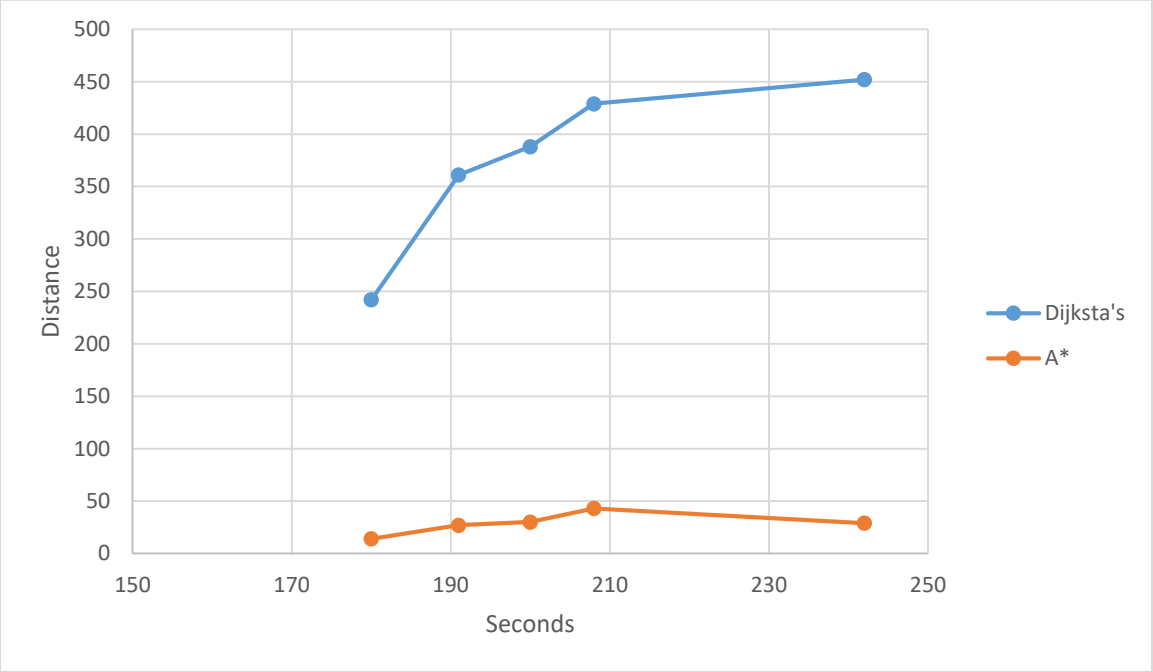
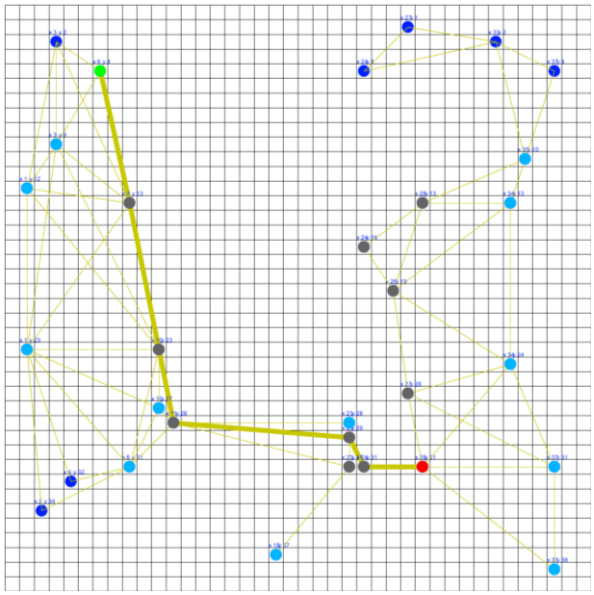


Figure 11 (Linear Distance on Small Graph):



Wyatt Plaga

Figure 12 (Manhattan Distance on Small Graph):

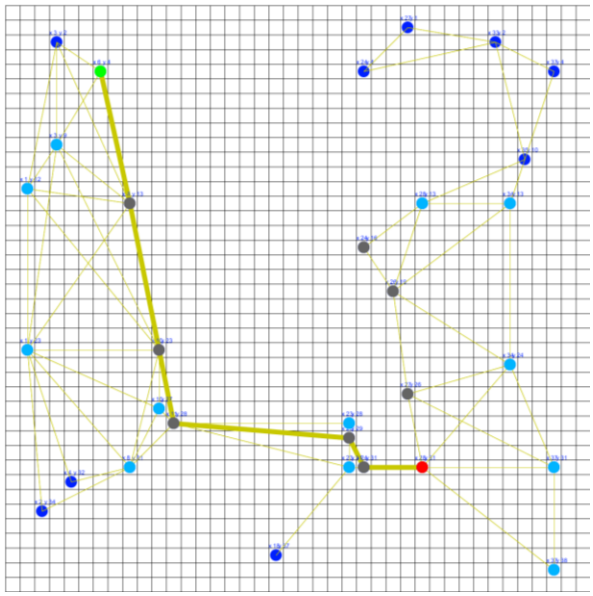


Figure 13 (Double Linear Distance on Small Graph):

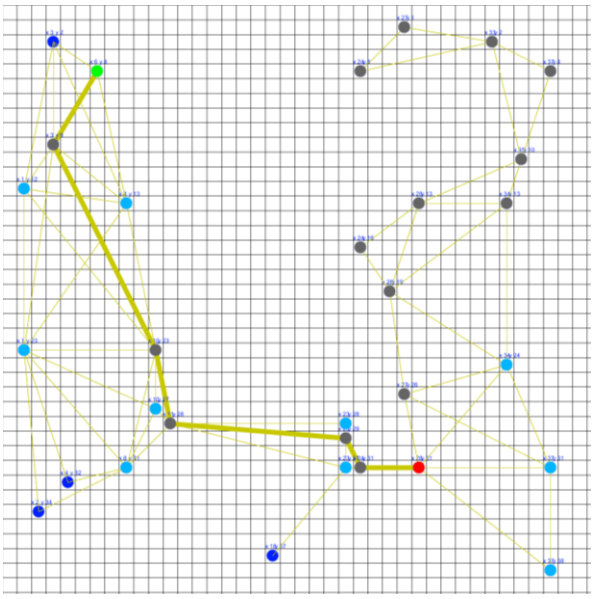


Figure 14:

Heuristic	Nodes Visited	Time To Completion
Linear Distance	2970	102
Mannhattan	1170	40
Distance Squared	97	3

Wyatt Plaga

Figure 15 (Pathfinding Environment):

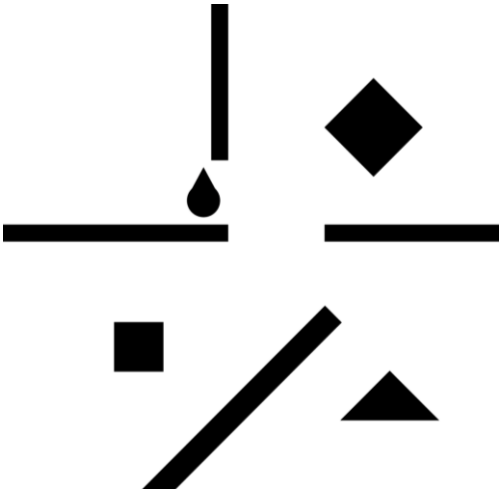


Figure 16 (Pathfinding Example 1):

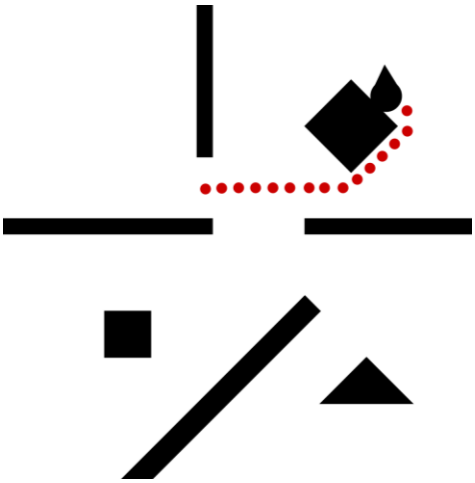
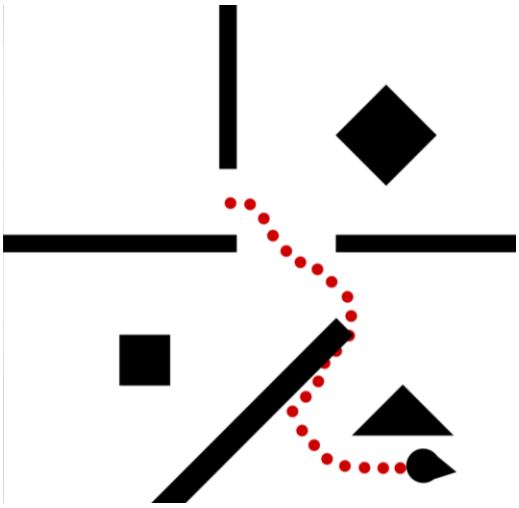


Figure 17 (Pathfinding Example 2):



Wyatt Plaga

Figure 18 (Pathfinding Example 3):