John Hall

RBE550 – HW2

Game Behavior:

I created the game in Python using PyGame to manage game logic and events. To begin, the map is randomly generated as both a PyGame display to draw sprites and as a 64x64 matrix for path planning. A goal location and hero are then randomly placed on the map where there is no obstacle. Lastly, 10 enemies are placed on the map. Once everything has been added to the field, the hero generates an initial path and the game begins.

Figure 1. Game field randomly generated with placed goal, hero, and enemies.

At each time step the hero will try to move to the next tile in its preplanned path. Before it does this, the hero checks how close enemies are to it. If any enemies are closer than 10 tiles away, the hero will replan a path to see if there is a way to avoid any enemies in the way. This preserves teleportations for dire situations and works surprisingly well to shiftily move the hero so that enemies collide with walls and become junk. If any enemy is within 3 tiles, the hero will use one of its five teleports and path plan from that new location. Teleporting acts as a move, so it won't move along the new path until the next turn. If the hero cannot find a path to the goal at its current location it will teleport until it has a path. If there are no teleports remaining, the hero has to make do and try and avoid the enemies with a new path. If there are no teleports remaining and it cannot find a path, the game ends. If the hero arrives at the same location as the goal, the hero wins!

Enemies are much simpler in that they always travel one tile towards the hero in the direction of greatest distance. If the step it was going to take is a wall or another piece of junk, the enemy will turn itself into immovable junk the hero must avoid. To detect if any of the enemies have come in contact with the hero, I used PyGames built in sprite collision detection to end the game.

Path Planner:

The hero utilizes A* to plan a path from its location to the goal, while avoiding obstacles and enemies. Similar to Dijkstra's algorithm, A* uses a node graph with weighted edges and a priority queue to search nodes with lesser weights first. In this case, I assigned each edge between two free nodes a weight of one to traverse between. When enemies are spawned onto the field, they create a 5x5 weighted kernel around them to dissuade the hero from traveling in their direction as seen in figure 2.

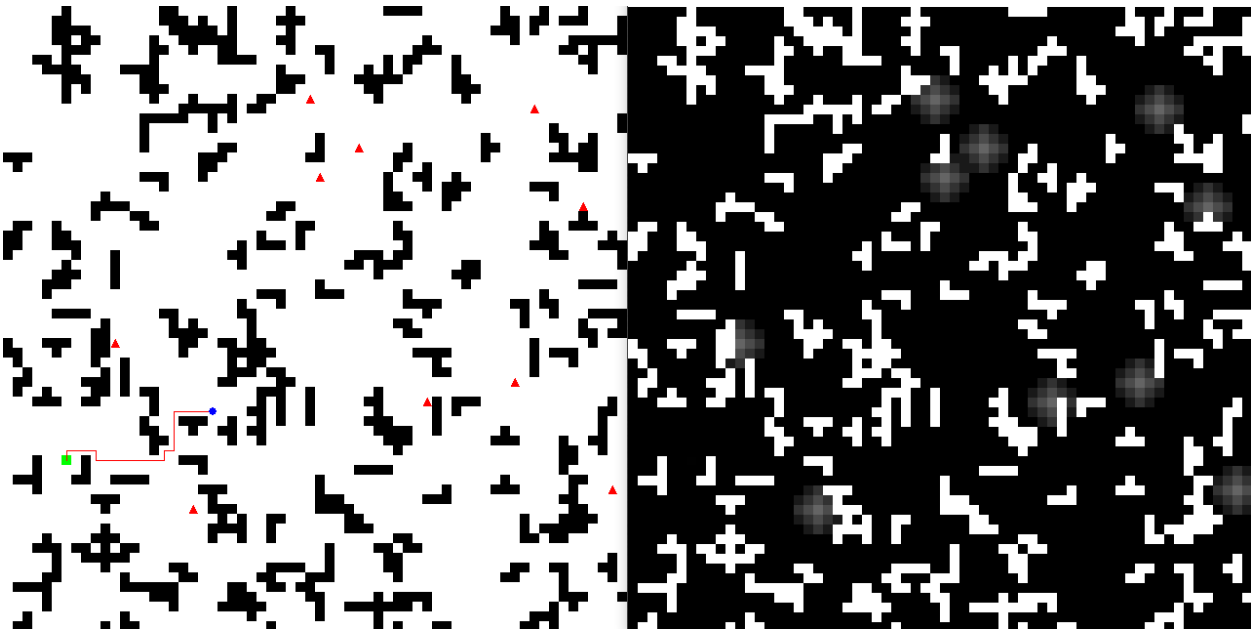

Figure 2. Displayed field (left) versus field used by the A* path planner (right).

What differentiates A* from Dijkstra's, is that our hero is omnipresent and knows where the goal location is. This allows the path planner to prioritize nodes in the direction of the goal using a heuristic. I decided to use Manhattan distance as the heuristic because the hero cannot move diagonally in my simulator. This meant that nodes closer to the goal were given higher priority in the queue. My A* implementation follows RedBlob Games pseudocode, but is modified to check for node validity (within the playing field and not a wall) along with returning the path as a list once completed. Figure 4 shows that the path planner prioritizes the direction of the goal, while still avoiding high weighted edges where enemies are close by.

```python
# Path Planner:
def Path_Planner(goal, start_loc):
    frontier = PriorityQueue()
    frontier.put((0,start_loc))
    came_from = {}
    cost_so_far = {}
    came_from[start_loc] = None
    cost_so_far[start_loc] = 0

    while not frontier.empty():
        current_node = frontier.get()
        if current_node == goal:
            return formated_path()

        for next_node in current_node.neighbors(): # +/- (1,1)
            if valid(next_node): # Within field and not a wall
                new_cost = cost_so_far[current_node] + next_node.cost()
                if next_node not in cost_so_far or new_cost < cost_so_far[next_node]:
                    cost_so_far[next_node] = new_cost
                    priority = new_cost + heuristic(next_node)
                    frontier.put((priority,next_node))
                    came_from[next_node] = current_node
```

Figure 3. Pseudocode describing A* algorithm structure.

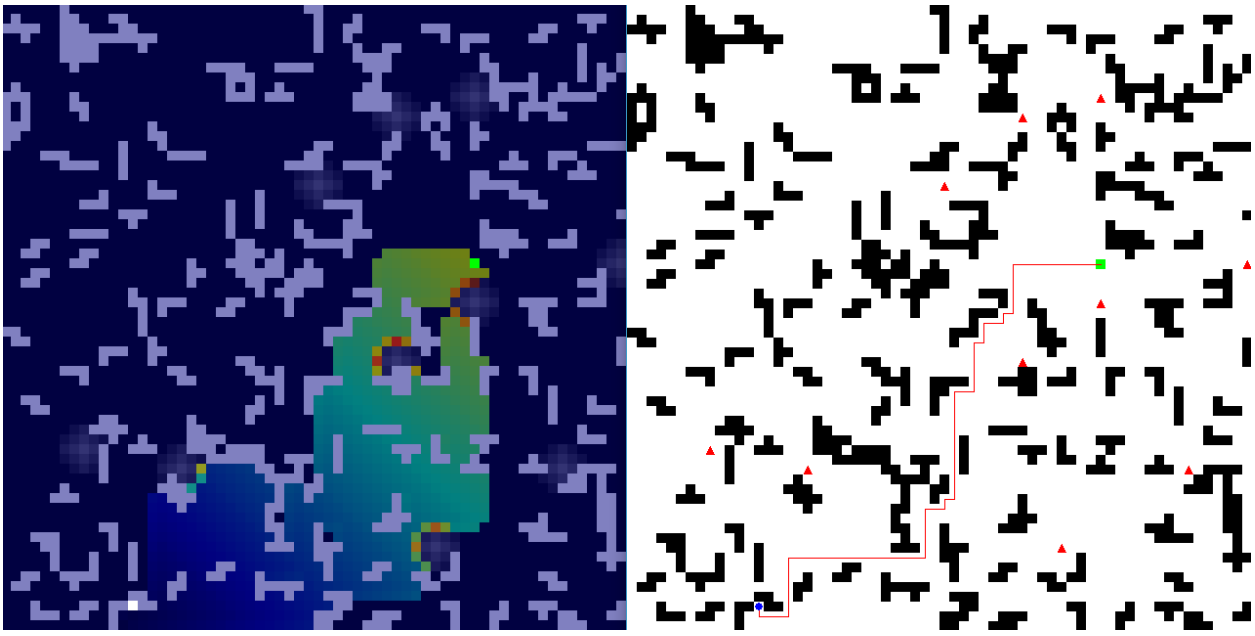

Figure 4. Color map of the A* algorithm based on cost to arrive at each node searched.

Results:

With five teleports, the hero can pretty consistently win the game. However, as mentioned in the path planning section, most games it can maneuver around enemies very well and make them crash into walls on their own. There were a couple of runs where the hero danced behind an

obstacle until the enemy on the other side finally became junk. It's very fun to restrict teleportations and see if the hero can survive.

I ran into a couple one-of-a-kind situations while testing the program. One time, an enemy ran into a wall while on top of the goal and proceeded to block the hero. I debated about making a fix for that, but I believe it should be a game over scenario and left it. The other interesting case was where the hero was being chased by an enemy a tile or two away and teleported either to the same spot, or to a tile right next to an enemy and was killed. Really unfortunate luck.

Resources:

https://www.redblobgames.com/pathfinding/a-star/introduction.html

https://www.geeksforgeeks.org/python/pygame-tutorial/

https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Sprite