Spike Outcome Report

Number: 12

Spike Title: Graphs and Search

Personal: Michael Williams (7668481)

**Goals:**

* Modify the graph search lab code or create your own simulation.
* Add in a simple moving agent to moves to each way-point in a successful graph search result.
* Be able to demonstrate either search-for-item or search-to-point examples.
* Make sure your agents (or their graph searching algorithms) correctly consider wall, mud or water tiles in the map.
* Display path cost for comparison
* Clearly demonstrate the need for different search algorithms

**Technologies, Tools, and Resources used:**

* Python IDE
* Sample Lab Code
* Lecture Material

**Tasks undertaken:**

1. The first thing we did was to make sure we had a thorough understanding of the different search methods in question (Dijkstra and A Star) before we did anything. In our case, we had already done this in another lab, so we were ready to move on.
2. We then decided that, contrary to the suggestion of using the Planet Wars lab, we would adapt the path finding lab to have an agent that can follow the path. We made some major modifications to the agent class to simplify the movement dramatically. We removed the whole notion of force and mass and everything and stripped out down to be this:

**class** Agent(object):  
  
 **def** \_\_init\_\_(self):  
 self.pos = Vector2D(10, 10)  
self.color = **'GREEN'**self.path = Path()  
  
 **def** update(self):  
 self.follow\_path()  
  
 **def** render(self, color=**None**):  
 *# draw the ship* egi.set\_pen\_color(name=self.color)  
  
 egi.cross(self.pos,10)  
 egi.circle(self.pos,10)

We chose to use the agent class as we were already very familiar with it and what each part of it did, making the time taken to refactor it for this purpose minimal.

1. The next step was to make it so that the world map would “tick”, to do this, we just had to enable the framerate:

self.fps\_display = clock.ClockDisplay()

1. Now that we had the map updating every second we had to add the agent, we modified the code that drew the path once the path was found so that it would also calculate the path to give to the agent and create it.

**if not** self.agent:  
 path\_pts = []  
 **for** i **in** range(0,len(self.path.path)):  
 path\_pts.append(PointToVector2D(self.boxes[self.path.path[i]].\_vc))  
  
 self.agent = Agent()  
 self.agent.path.set\_pts(path\_pts)  
  
self.agent.update()  
self.agent.render()

1. Now we had the agent drawing and moving! But it was not very good… So we made some major simplifications to the path following so that it did this:

**def** follow\_path(self):  
 to\_target = self.path.current\_pt() - self.pos  
 dist = to\_target.length()  
  
 threshold = 5  
  
 *# if we are close enough to the current point, go to the next one* **if** dist < threshold **and not** self.path.is\_finished():  
 self.path.inc\_current\_pt()  
  
 **if** self.pos.x > self.path.current\_pt().x:  
 self.pos.x -= 5  
 **if** self.pos.x < self.path.current\_pt().x:  
 self.pos.x += 5  
 **if** self.pos.y > self.path.current\_pt().y:  
 self.pos.y -= 5  
 **if** self.pos.y < self.path.current\_pt().y:  
 self.pos.y += 5

Instead of doing anything complex, on each tick we just bring ourselves 5 units closer to the goal position in each direction. Now the agent was following the path much better (and faster).

1. Now we just needed to make the agent be able to perform a search using Dijkstra or A Star, we chose to just alternate using a key press for this. We modified the code when the map loaded to not add the target position, we also cut out all the key mappings so we could add our own. We made it so when you press “Q” the target would be loaded (hard coded number) and the A Star search would happen. Similarly we did this for “W” and Dijkstra. To reset, we set the “R” key mapping.

**What we found out:**

We found out more in depth how each of the search algorithms work and when to use them. After learning about the algorithms and seeing them work it is very clear why you need each of these searches. The A Star search can only be ideal when you actually know when the end goal is, allowing you to calculate the distance to the goal. When doing an item search you do not know where the item is, so the next best thing you can do is simply use the Dijkstra algorithm.

See below some screenshots of both of these scenarios in action:



