SPIKE REPORT SPIKE 12

Jake Lucic

2103435 AI for Games

Number: Spike 12
Spike Title: Graphs and Search
Personal: Jake Lucic (2103435)

Context:

Graph can be used to represent many aspects of a game, and selecting and using the right type of graph search is critical to finding solutions to problems in games.

Knowledge/Skill Gap:

The developer needs to be able to use appropriate graph search algorithms to solve problems for characters and controllers game situations.

Goals:
Clearly demonstrate the appropriate use of the Dijkstra's (search for item) and A* (search to position).
• 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:

In this task, the technologies used are listed below:

- Simple Code base, the code used in lab 10
- Python IDLE v3.6.4 / Python language / PyCharm
- Graphs, Paths and Search Lecture and Lab

Tasks undertaken:

- Read the code base and understand what each function does
- Study the different classes and variable names
- Approach Task using similar process below

Process:

Before approaching this task, I studied the lab 10 task and developed a good understanding of the different path searches. For this task, I would only be using A* and Dijkstras graph searches to locate an item or a target, however, it's crucial to have a good understanding on the fundamental search algorithms these are based on (BFS/DFS) are really important.

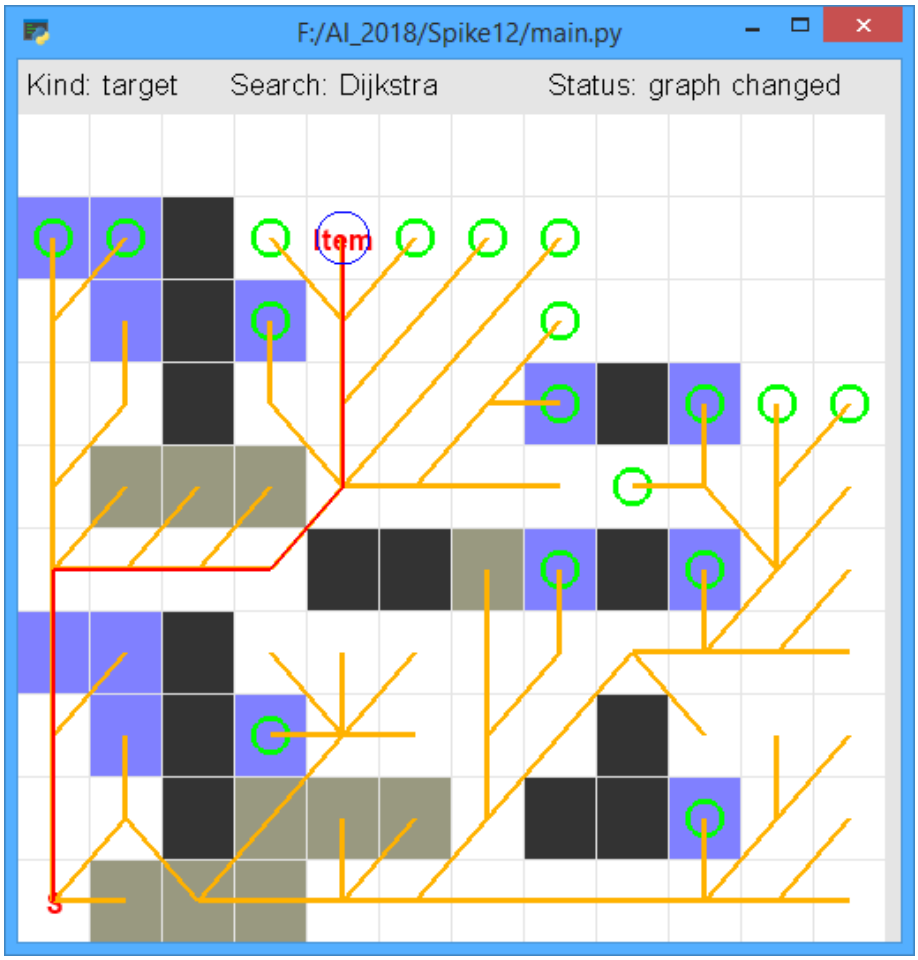
Firstly, I had to understand how the Start and End Markers were placed in the lab10 example, I then followed by adding an Agent class which is just a simple circle that starts his position at the start marker, then every update method it shortens its path by checking if the Agent is at its next destination point, until the length of its path is 0. This happens every time a new path is made.

```
from point2d import Point2D




class Agent(object):
    def __init__(self, x, y, boxes):

        self.pos = Point2D(x, y)
        self.boxes = boxes
        self.path = []
        self.finished = False

    def update(self):
        if len(self.path) > 0:
            src = self.pos
            dest = self.boxes[self.path[0]].vc

            scale = 0.85
            self.pos.x = (src.x + (dest.x - src.x) * scale)
            self.pos.y = (src.y + (dest.y - src.y) * scale)

            if self.pos.x == dest.x and self.pos.y == dest.y:
                self.path = self.path[1:]

            if len(self.path) == 0:
                self.finished = True


    def render(self):
        egi.blue_pen()
        egi.circle(self.pos, 15)
```

I then followed by adjusting the end markers display depending on what mode it was on. If it was on A* I made the target look for a target, and if the mode was on Dijkstras, It would look for an Item.

This is purely so I could explain which mode is more effective for either task.

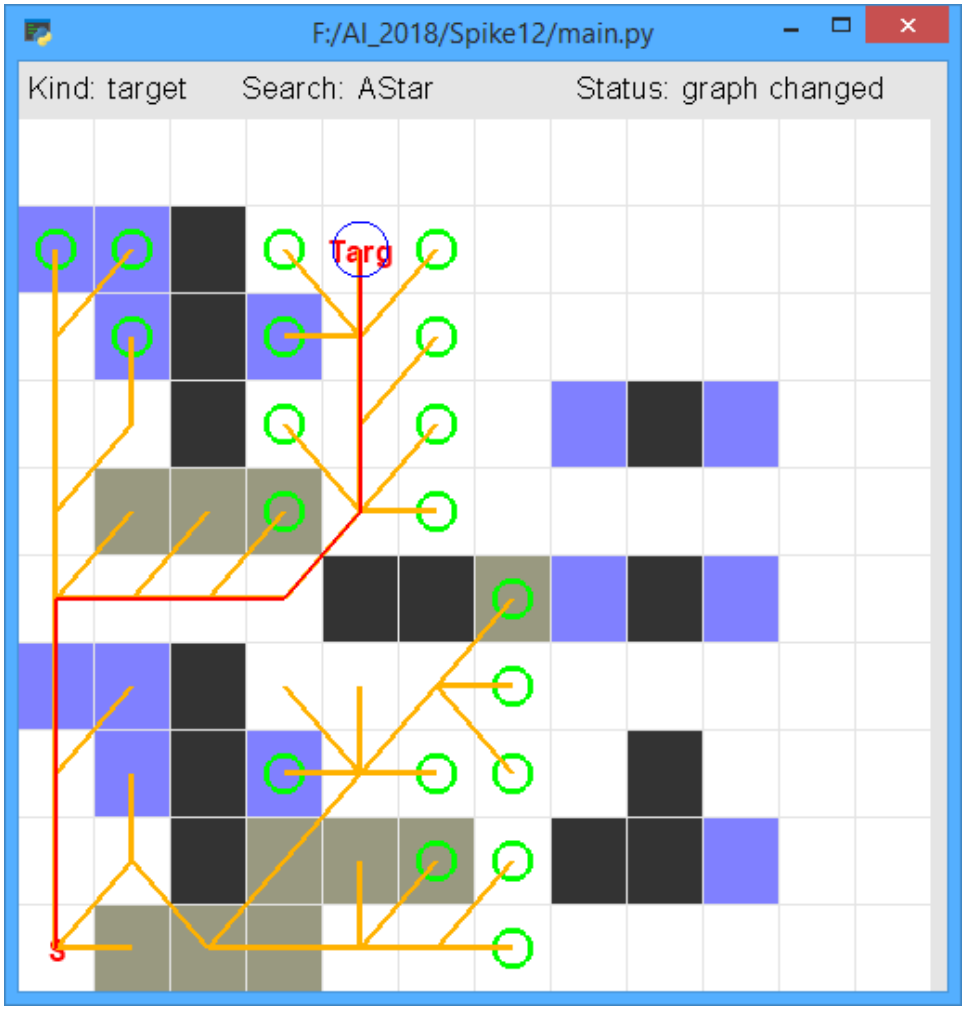
Dijstra's Method:




```
Success! Done! Steps: 70 Cost: 19.4142
Path (12)=[0, 12, 24, 36, 48, 49, 50, 51, 64, 76, 88, 100]
Open (17)=pq: [(17.727800000000002, 32, 27), (19.828400000000002, 95, 99), (19.828400
Closed (70)={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 22, 23, 24
Route (87)={0: 0, 1: 0, 12: 0, 13: 0, 24: 12, 25: 13, 2: 13, 36: 24, 37: 24, 3: 2, 15
```

Dijkstras graph search is similar to Breadth first search that it expands a certain path however Dijkstras is more efficient that it calculates its cost as it develops a path. It then chooses to develop one of those paths until a certain cost has been achieved, and switches to develop another path if no "item" has been found yet. This is an effective search for item, because it is not looking for a specific Target. Its job is to find the first entity of what its looking for and then completes its search. It does not make use of an estimate of the position of the target item, which is beneficial for when you are looking for an item. However, A* (look for specific target), makes use of an estimate, as shown on the next page. It completed its search with a cost of 19.4 and used 70 steps to complete its task.

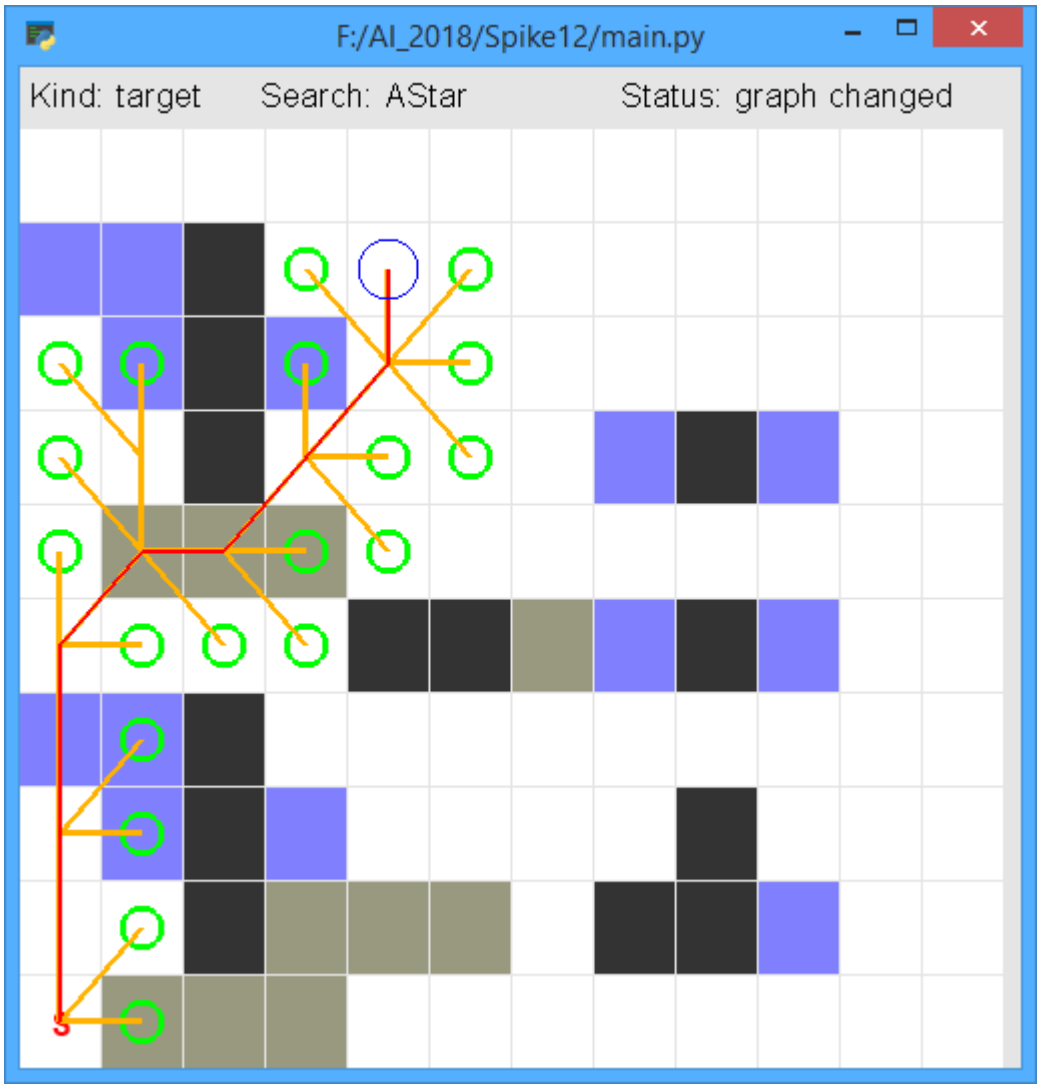
Astar




```
Success! Done! Steps: 32 Cost: 19.4142
Path (12)=[0, 12, 24, 36, 48, 49, 50, 51, 64, 76, 88, 100]
Open (19)=pq: [(19.656799999999997, 49, 18), (19.727800000000002, 37, 29), (19.8284, 31, 63), (19.82
Closed (32)=[0, 1, 2, 3, 4, 5, 12, 13, 15, 16, 24, 25, 28, 36, 37, 39, 40, 41, 48, 49, 50, 51, 60, 6
Route (51)={0: 0, 1: 0, 12: 0, 13: 0, 24: 12, 25: 13, 36: 24, 37: 24, 2: 13, 48: 36, 49: 48, 3: 2, 1
```

Astar is like Dijkstra's search, in that it expands its path based on cost and switches, however, it has an estimate of where the target is. It does this by using a heuristic function, such as the manhattan distance. It knows where it needs to go, but it wants to find the most cost efficient path. You can see here that it only chooses 3 main branches, it does not bother to develop the lower paths further, because it knows the target position is not further in that direction. This is more efficient for search for target. It completed its search for target with a cost of 19.4, in 32 steps, which is a lot shorter than Dijkstras search for Item. However, if I had multiple items in the Dijkstras search method, it would have found an item with a shorter cost. By having a minimum edge cost of 1, you can see it decides to take the route of the clear path, rather than the muddy path. You can see the difference on the next page of a mininmum edge cost of 10.

Astar cont




```
Success! Done! Steps: 11 Cost: 24.071
Path (10)=[0, 12, 24, 36, 48, 61, 62, 75, 88, 100]
Open (19)=pq: [(34.071000000000005, 28, 89), (34.485200000000006, 29, 99), (34.4852000
Closed (11)={0, 36, 100, 73, 75, 12, 48, 24, 88, 61, 62}
Route (30)={0: 0, 1: 0, 12: 0, 13: 0, 24: 12, 25: 24, 36: 24, 37: 24, 48: 36, 49: 48,
```

Here you can see that with a minimum edge cost of 10, it considers the types of terrain it is going through and plans its path accordingly, in the previous example, we had a minimum edge cost of 1. The cost is increased slightly, and the steps taken is shorter, this is because it decided to make its way through the mud.