Spike: Spike_12
Title: **Graphs and Search**

Author: Daniel Newman, 6449921

Goals / deliverables:
The goal of this task is to utilise graph traversal techniques, including Dijkstra & A* techniques to demonstrate pathfinding behaviour using a simple bot.

Technologies, Tools, and Resources used:
List of information needed by someone trying to reproduce this work

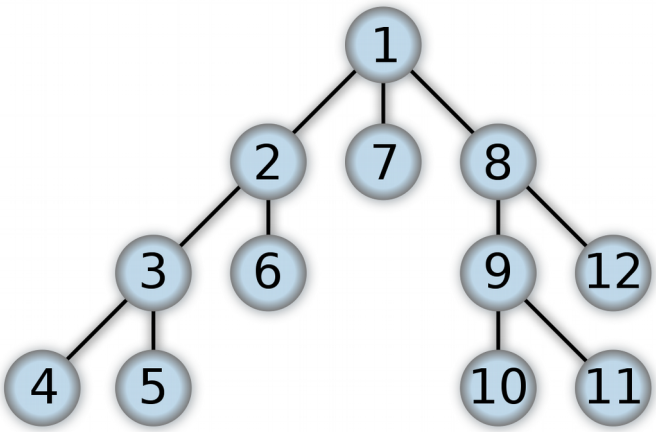
- Python 3.6.4
- Python compatible IDE

Tasks undertaken:
Utilising the graph search algorithms, we wanted to demonstrate how the use of these techniques influences a bot moving around a tilemap to collect particular items or reach a particular position.

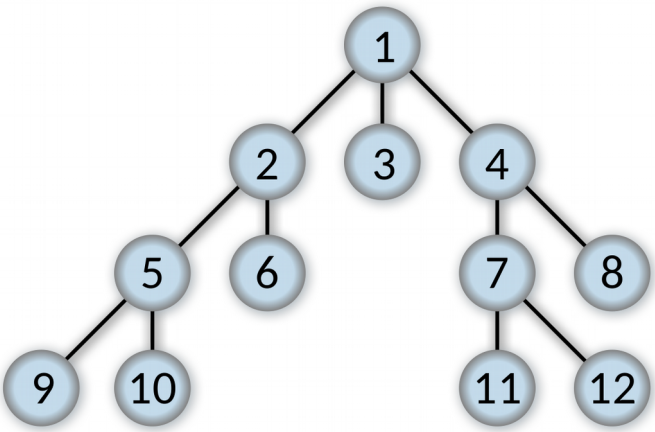
This was done by creating a path plan for the bot to follow using one of the available graph search techniques. Once the path plan was complete, we created a follower bot which took each of the node's position on the path as the breadcrumbs to follow to reach its target. In doing so we could reliably path-find around different objects and terrains to reach the selected point using the best path the graph search method provided.

```
if self.world.path:
    route = self.world.path.path
    for box in route:
        fuzzFactor = 4
        pos = self.world.get_box_by_iterable(box)._vc
        while(abs(math.hypot(self.world.follower.x - pos.x, self.world.follower.y - pos.y))) > f
            if self.world.follower.x > pos.x + fuzzFactor:
                self.world.follower.x -= fuzzFactor #self adjustment
            elif self.world.follower.x < pos.x - fuzzFactor:
                self.world.follower.x += fuzzFactor  # self adjustment
            if self.world.follower.y > pos.y + fuzzFactor:
                self.world.follower.y -= fuzzFactor  # self adjustment
            elif self.world.follower.y < pos.y - fuzzFactor:
                self.world.follower.y += fuzzFactor  # self adjustment
            self.clear()
            self.on_draw()
            self.flip()
            time.sleep(0.01)
```

However there is a cost to using each different search algorithm.
BFS & DFS
These techniques are similar in code structure but differ in terms of their traversal method. Depth-first searches to the bottom leaf of the branch before coming up a level and checking all available sub-branches, repeating up each level until it reaches the trunk again.
Breadth-first checks each branch at the current level before moving down from the top layer to the next branch, where it checks every sub-branch at that level before moving up. This is memory intensive as more branches are added.



DFS



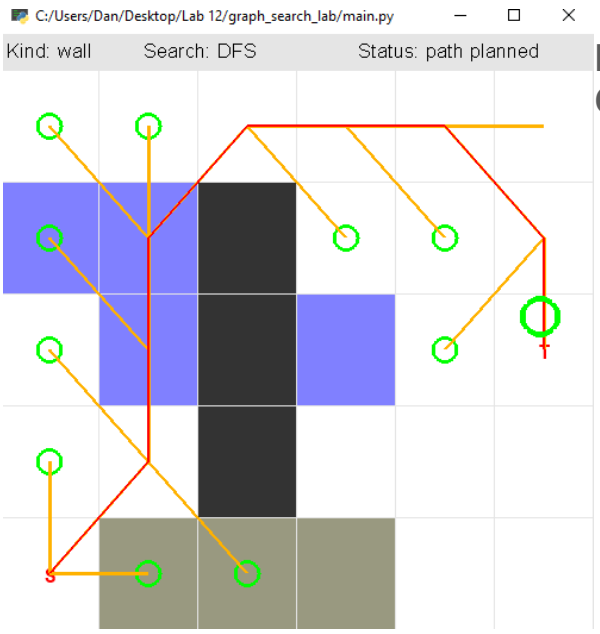
BFS

There are benefits to both, If you are required to search fro something that is likely to be at the lowest depth possible, then BFS is more likely to give an optimal solution, whereas DFS would reach a solution quicker but it may not be optimal if at the lowest depth possible, but the benefit of DFS is that it requires less memory to complete its task.

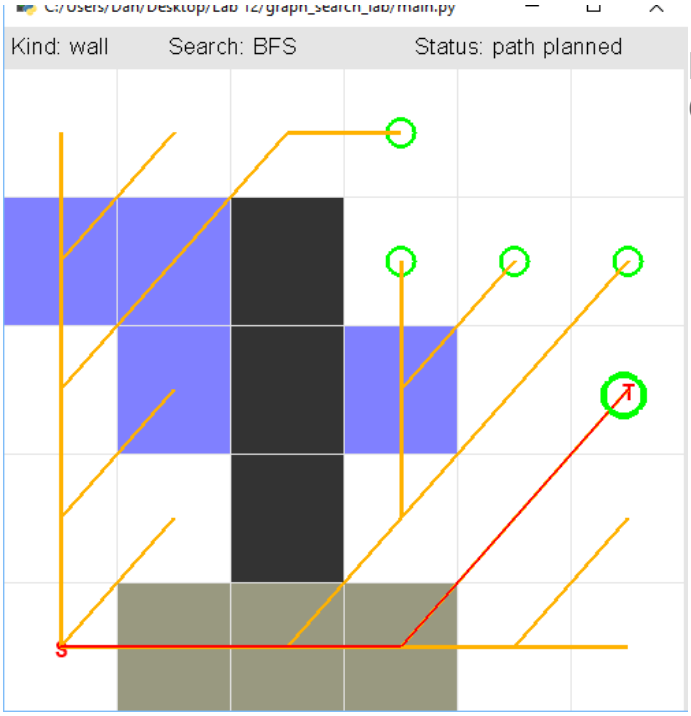
Using Dijkstra factors in a cost value for traversal between each node, and bases the best path decision upon the minimal cost value between each choice at each level. This allows for more effective pathfinding compared to the previous methods, but fails to take into account a running total so fails to predict what the minimum cost should be for the whole path. A* works similiarly to Dijkstra but includes the running total and a heuristic that counts for the minimum cost a traversal could have and uses that to attempt to find the best path.

What we found out:

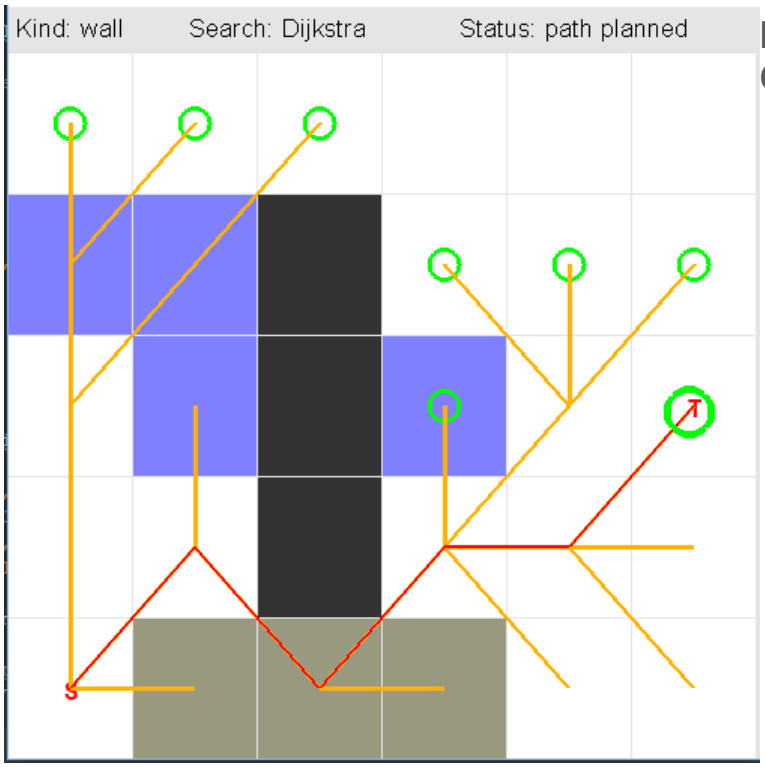
We found out that there is a remarkable difference between each path-finding algorithm, with each method iterating on the earlier methods to improve upon them and provide lesser cost paths that an agent can utilise to get to and from certain locations.



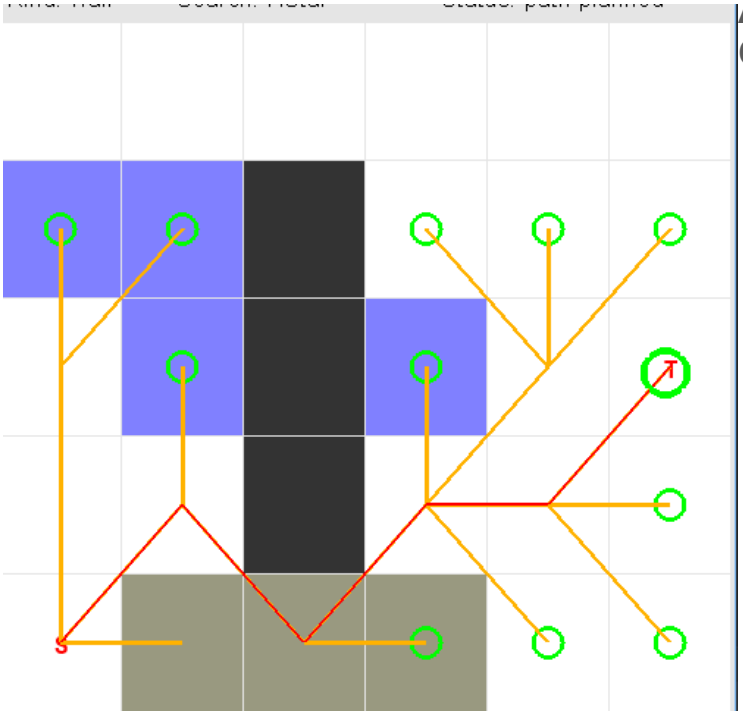
DFS
Cost 27.8994



BFS
Cost 14.2426



Dikjstra
Cost: 9.485



A*
Cost: 8.54