SWINBURNE UNIVERSITY OF TECHNOLOGY

COS30002

Navigation with Graphs

Task 18, Week 10
Spike: Spike_No18

Matthew Coulter
S102573957

21/05/2020

Tutor: Tien Pham

Table of Contents

Goals / Deliverables

The aim of this task is to apply the searching functions we have learned from task 17 into our agent world. Here, the agent will follow this path on graph, because we will be using a graph search for each agent to make its way from point a to point b.. We will also use the seek code and remove steering forces, simply updating the velocity. This way, the agents will will follow the path exactly.

Technologies, Tools, and Resources used:

If you wish to reproduce my work, here is a list of all the required tools and resources that you will need:

- Visual Studio
- Python language pack
- This spike report
- Git bash or sourcetree (if you wish to access my code)
    - repo: 'git clone https://mattcoulter7@bitbucket.org/mattcoulter7/cos30002-102573957.git'

Tasks undertaken:

Basic understanding

The way that this solution will work is through a graph. The graph represents a grid which is initialised as a 2D list. All of the values in the list are going to be 0, unless it represents a node that cannot be travelled through, hence that would become a 1. This is important because our selected graph search will ignore taking paths through nodes that aren't available.

The way that the path is followed is simply by reusing opur path finding code from lab 9. Instead of using the arrive function, we will use the seek instead because we are removing steering forces.

That is pretty much it! Below will explain the code for the implementation of the above.

implementing the graph

We are going to create a new class called Graph. This wil contain the 2D list Grid, but also the width and height for the amount of nodes. Add this initialisation function to your new Graph class:

```
def __init__(self,world):
        self.world = world
        # Grid
        self.scale = 100
        self.grid_size = world.cx/world.cy * self.scale
        self.height = round(world.cx / self.grid_size)
        self.width = round(world.cy / self.grid_size)
        self.grid = [[0 for x in range(self.width)] for y in 
range(self.height)]
```

We also want to draw our grid to the screen. Add this render function to do so:

```
   def render(self):
        ''' Draws the grid to the screen'''
        grid = self.grid_size
        egi.white_pen()
        for i in range(len(self.grid)):
            for j in range(len(self.grid[i])):
                x = i*grid
                y = j*grid
                pt1 = Point2D(i*grid,j*grid)
                pt2 = Point2D(pt1.x + grid,pt1.y)
                pt3 = Point2D(pt1.x,pt1.y + grid)
                egi.line_by_pos(pt1,pt2)
                egi.line_by_pos(pt1,pt3)
```

The rest of the functions will be used by other classes for handling the difference between position in pixels and position by node coordinates. Add these functions for the conversions:

```
  def get_node(self,pt):
    ''' Returns the node of a given coordinate '''
    for i in range(len(self.grid)):
      for j in range(len(self.grid[i])):
        if floor(pt.x/self.grid_size) == i and
floor(pt.y/self.grid_size) == j:
          return Vector2D(i,j)

  def fit_pos(self,pt,type):


    ''' Takes a coord and fits it to a given place in a square '''
    for i in range(len(self.grid)):
      for j in range(len(self.grid[i])):
        if floor(pt.x/self.grid_size) == i and
floor(pt.y/self.grid_size) == j:
          if type == 'center':
            return Vector2D(i*self.grid_size +
self.grid_size/2,j*self.grid_size + self.grid_size/2)
          elif type == 'corner':
            return Vector2D(i*self.grid_size,j*self.grid_size)

  def get_pos(self,pt,type):
    ''' Returns the position of a node, fitted to the square '''
    return self.fit_pos(Point2D(pt.x * self.grid_size,pt.y *
self.grid_size),type)

  def node_available(self,node):
    ''' returns true if node is 0 '''
    return not self.grid[node.x][node.y]

  def node_exists(self,pt):
    ''' returns true if a node is valid '''
    return pt.x in range(0,self.grid_count) and pt.y in
range(0,self.grid_count)

  def update_grid(self,node):
    ''' 0 becomes 1 '''
    self.grid[node.x][node.y] = 1

  def rand_node(self):
    ''' returns an random node that is still available '''
    return Point2D(randrange(0,self.height),randrange(0,self.width))
```

It simply allows a random position to return a node that is corresponds to. Likewise, a node can return a pixel position of either the center or the corner.

implementing blocks on the graph

We want to have blocks that agent wil have to travel around. Add this class to achieve that:

```python
from graphics import egi
from point2d import Point2D
from random import randrange

class Block(object):
    def __init__(self, world):
        self.world = world
        self.size = world.graph.grid_size
        node = self.world.graph.rand_node()
        self.pos = self.world.graph.get_pos(node,'corner')
        self.world.graph.update_grid(node)
        self.shape = [
            Point2D(self.pos.x,self.pos.y),
            Point2D(self.pos.x + self.size,self.pos.y),
            Point2D(self.pos.x + self.size,self.pos.y + self.size),
            Point2D(self.pos.x,self.pos.y + self.size)
        ]

    def render(self):
        egi.white_pen()
        # draw it!
        egi.closed_shape(self.shape,filled=True)
```

The key part of this class is this line: *self.world.graph.update_grid(node)*. What this does is update the grid so when performing a search, it knows to avoid that particular node.

applying the search function

After removing all force variables and updating position based on velocity only, we will have our astar search function live within a py file called search_functions.py. You can add more functions to this file for different searching algorithms, but in this case we will use the astar searchin method as it is the most efficient algorithm. Here is the code for that file:

```python
from vector2d import Vector2D
from node import Node

def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the 
given end in the given maze"""
    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0
```

```
    # Initialize both open and closed list
    open_list = []
    closed_list = []
    # Add the start node
    open_list.append(start_node)
    # Loop until you find the end
    while len(open_list) > 0:
        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index
        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)
        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1] # Return reversed path
        # Generate children
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1),
(-1, 1), (1, -1), (1, 1)]: # Adjacent squares
            # Get node position
            node_position = Vector2D(current_node.position.x +
new_position[0], current_node.position.y + new_position[1])
            # Make sure within range
            if node_position.x > (len(maze) - 1) or node_position.x < 0 or
node_position.y > (len(maze[len(maze)-1]) -1) or node_position.y < 0:
                continue
            # Make sure walkable terrain
            if maze[node_position.x][node_position.y] != 0:
                continue
            # Create new node
            new_node = Node(current_node, node_position)
            # Append
            children.append(new_node)
        # Loop through children
        for child in children:
```

```
            # Child is on the closed list
            for closed_child in closed_list:
                if child == closed_child:
                    continue
            # Create the f, g, and h values
            child.g = current_node.g + 1
            child.h = ((child.position.x - end_node.position.x) ** 2) +
((child.position.y - end_node.position.y) ** 2)
            child.f = child.g + child.h
            # Child is already in the open list
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue
            # Add the child to the open list
            open_list.append(child)
```

Now in our agent class we need to have the path and a function that will assign the path values to the astar result. This function will be called everytime the mouse is clicked because that is when the world.target is moved.

Here is that function:

```
   def update_path(self):
        ''' Reassigns the points of path to head towards new
destination'''
        maze = self.world.graph.grid
        start = self.world.graph.get_node(self.pos)
        end = self.world.graph.get_node(self.world.target)
   def calculate(self):
        # calculate the current steering force
        mode = self.mode
        vel = Vector2D(0,0)
        if self.mode == 'follow_path':
            vel = self.follow_path()
        return vel
```

There are some other functions with slight changes in agent that should be noted:

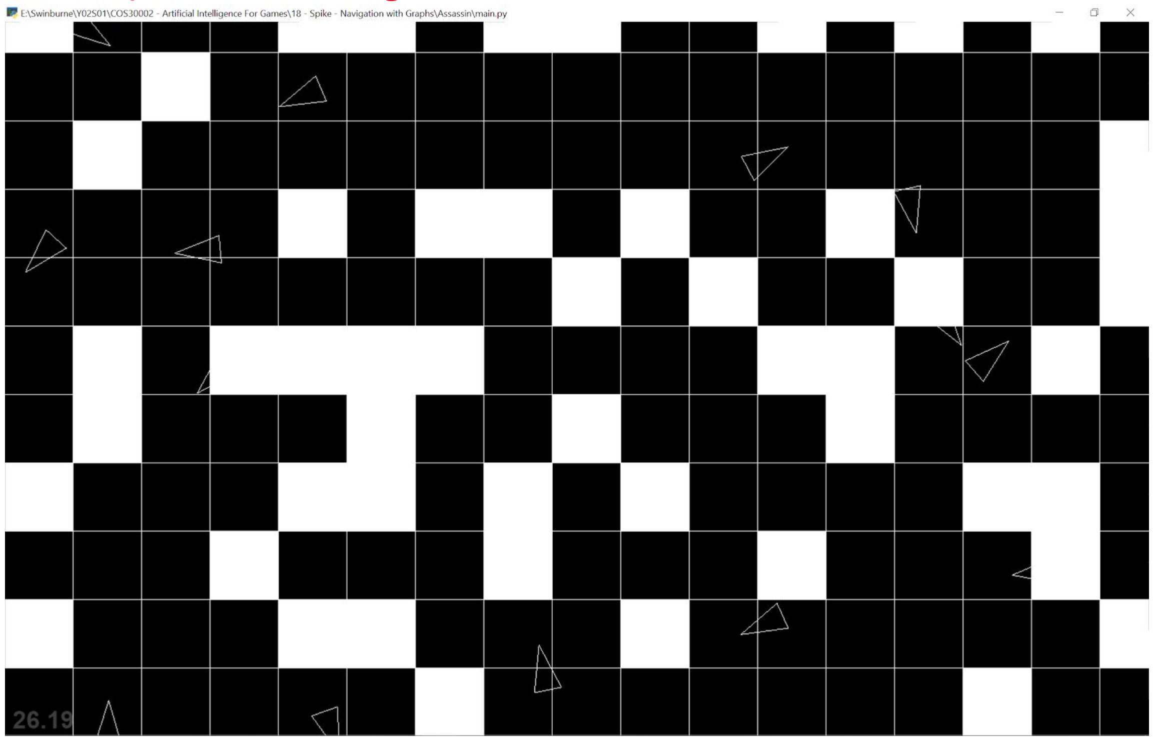
```
    def update(self, delta):
        ''' update vehicle position and orientation '''
        # update mode if necessary
        self.update_mode()
        # new velocity
        self.vel = self.calculate()
        # limit velocity
        self.vel.truncate(self.max_speed)
        # update position
```

```
        self.pos += self.vel * delta
        # update heading is non-zero velocity (moving)
        if self.vel.length_sq() > 0.00000001:
            self.heading = self.vel.get_normalised()
            self.side = self.heading.perp()
        if self.world.target:
            if self.intersect_pos(self.world.target):
                self.path.clear()
        if self.world.graph.node_available(end):
            pts = astar(maze,start,end)
            for pt in pts:
                pt.x = pt.x * self.world.graph.grid_size +
self.world.graph.grid_size/2
                pt.y = pt.y * self.world.graph.grid_size +
self.world.graph.grid_size/2
            self.path.set_pts(pts)

   def seek(self, target_pos):
        ''' move towards target position '''
        desired_vel = (target_pos - self.pos).normalise() * self.max_speed
        return desired_vel

    def follow_path(self):
        if (self.path.is_finished()):
            # Arrives at the final waypoint
            return self.seek(self.path._pts[-1])
        else:
            # Goes to the current waypoint and increments on arrival
            if self.intersect_pos(self.path.current_pt()):
                self.path.inc_current_pt()
            return self.seek(self.path.current_pt())

    def intersect_pos(self,pos):
        ''' Returns true if assassin intersects a particular position'''
        return self.world.graph.get_node(pos) ==
self.world.graph.get_node(self.pos)

   def update_mode(self):
        ''' Updates state according to different variables '''
        if self.path._pts:
            self.mode = 'follow_path'
        else:
            self.mode = None
```

What we found out:

The result works quite well. Here are some screenshots to demonstrate:

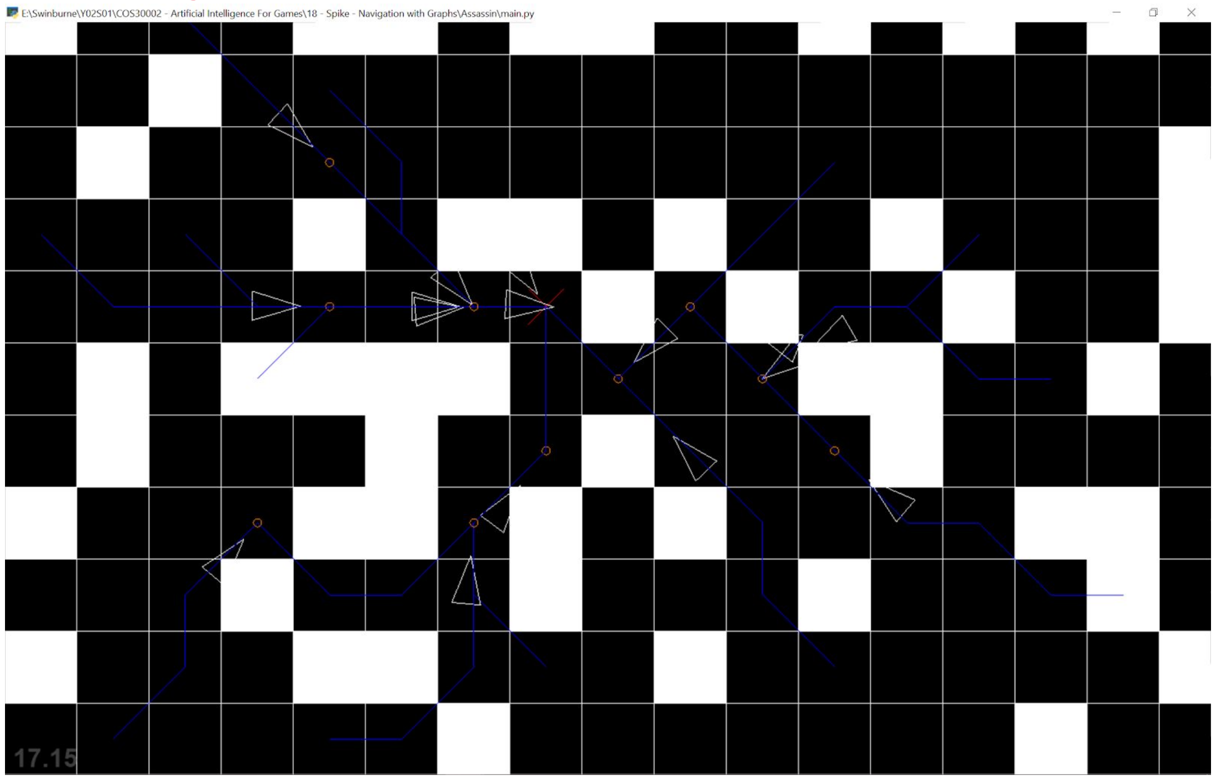
Ready to start moving:



E:\Swinburne\Y02S01\COS30002 - Artificial Intelligence For Games\18 - Spike - Navigation with Graphs\Assassin\main.py

26.19

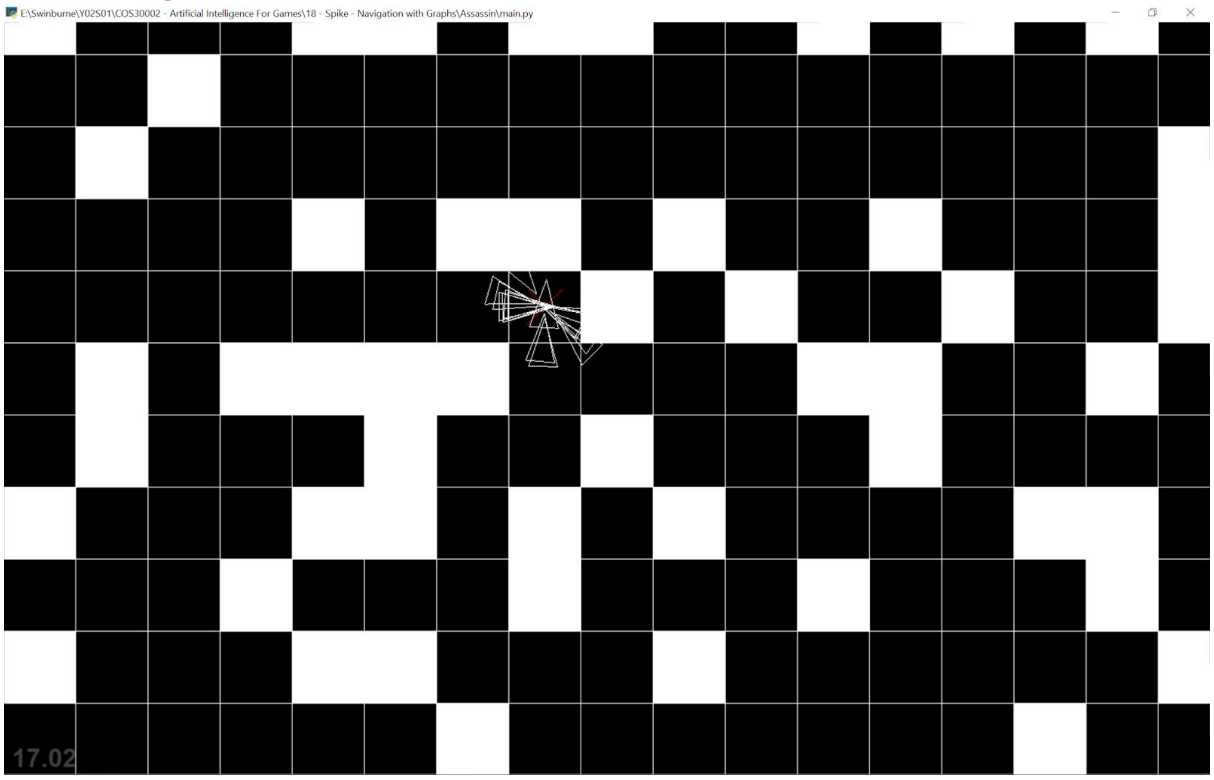
following path:



E:\Swinburne\Y02S01\COS30002 - Artificial Intelligence For Games\18 - Spike - Navigation with Graphs\Assassin\main.py

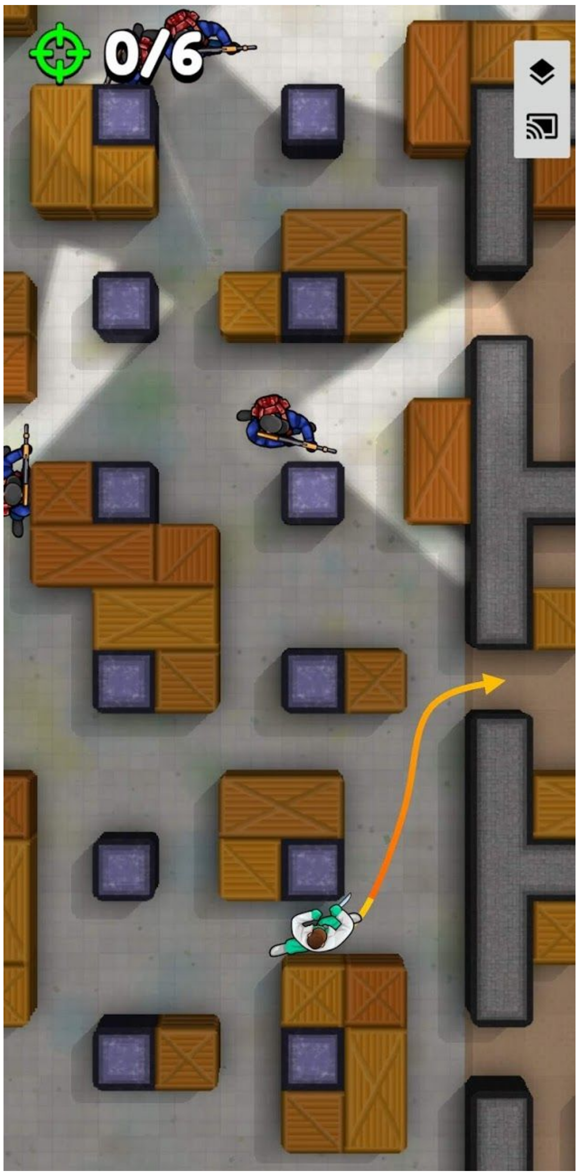
17.15

arriving at destination:


E:\Swinburne\Y02S01\COS30002 - Artificial Intelligence For Games\18 - Spike - Navigation with Graphs\Assassin\main.py
17.02

Quite clearly, they all avoid they blocks really well by following their own individual calculated path.

This could potentially serve as a nice introduction to making this mobile game called Assassin:



The primary movement of this game is driven by clicking on parts of the map and your assassin travelling to those places. The calculation would most likely be an astar calculation.