Spike: Spike_08
Title: Navigation with Graphs
Author: Adnan Zafar, 103169535

Goals / deliverables:

The goal of this spike is to modify and extend the existing BoxWorld codebase to implement a simulation where agents plan paths using heuristic search algorithms and then navigate the environment. All steering forces were removed and only velocity was updated for the agents to follow the path properly.
.

Technologies, Tools, and Resources used:

- Visual Studio Code
- Python 3.0
- Pyglet 1.5.27
- Lab14 and Lab09 code

Tasks undertaken:

1. Created the Agent class:

The Agent class is designed to represent moving agents in the simulation. Here's an explanation of the methods added to this class and their purpose:

- **__init__(self, start_pos, speed, color='GREEN'):** The constructor initializes an agent with a starting position (start_pos), a speed (speed), and an optional color (default is 'GREEN'). It also sets the agent's path as not set initially (self.path_set = False).
- **update(self):** This method updates the agent by calling the follow_path() method, which makes the agent follow its assigned path.
- **set_new_target(self, new_target):** This method sets a new target for the agent by updating its path with the new target points. It also resets the current point index of the agent's path so that the agent starts navigating from the beginning of the new path.
- **render(self, color=None):** This method renders the agent on the screen using the provided color or the agent's default color. It draws a cross and a circle at the agent's current position.
- **follow_path(self):** This method makes the agent follow its assigned path. It calculates the distance to the current target point and checks if the agent is within a certain threshold. If the agent is close enough to the target point, it increments the current point index to navigate to the next point on the path. It then calculates the direction vector to the target point and updates the agent's position based on its speed and direction.

These methods enable the agent to navigate through the environment by following a path while updating its position based on its speed. This results in a moving agent that can adapt its movement according to the assigned path and target points.

```python
class Agent(object):
    def __init__(self, start_pos, speed, color='GREEN'):
        self.pos = start_pos
        self.path_set = False
        self.color = color
        self.path = Path()
        self.path_pts = []
        self.speed = speed

    def update(self):
        self.follow_path()

    def set_new_target(self, new_target):
        self.path.set_pts(new_target)
        self.path.current = 0  # Reset the current point
```

```python
def render(self, color=None):
    # draw
    egi.set_pen_color(name=self.color)
    egi.cross(self.pos,10)
    egi.circle(self.pos,10)

def follow_path(self):
    to_target = self.path.current_pt() - self.pos
    dist = to_target.length()
    threshold = 5
    if dist < threshold and not self.path.is_finished():
        self.path.inc_current_pt()
    # Calculate the direction vector
    direction = (self.path.current_pt() - self.pos).get_normalised()
    # Move agent in the direction of the target by its speed
    self.pos += direction * self.speed
```

2. Modified the BoxWorld class:

The BoxWorld class has been modified to implement moving agents in the simulation. Here's an explanation of the added or modified methods to handle moving agents:

- **add_agent(self, start_idx):** This method adds an agent to the world by calling the set_agent_start() method, which sets the agent's starting position based on the provided index.
- **set_agent_start(self, agent_idx, idx):** This method sets the starting position for the agent with the given index (agent_idx). If the agent index is within the current agent list, it updates the agent's starting position. Otherwise, it adds a new agent to the list. The agent's starting position is set using the box index (idx).
- **set_target(self, idx):** This method sets the target box based on the provided index (idx). It updates the agents' starting positions to the previous target position and sets the new target node. It then updates the agents' paths and targets based on the new target position.
- **plan_path(self, search, limit, agent_idx, speed=None):** This method is modified to handle the path planning for a specific agent, given by agent_idx. It initializes the agent with a start position and speed if it doesn't exist. It then sets the agent's path if it hasn't been set before.

These modifications to the BoxWorld class allow the simulation to handle multiple moving agents, update their starting positions and target positions dynamically, and plan paths for each agent individually.

```python
def set_target(self, idx):
    '''Set the target box based on its index idx value. '''
    # remove any existing target node, set new target node
    if self.start == self.boxes[idx]:
        print("Can't have the same start and end boxes!")
        return
    # Update agents' start positions to the previous target posit
    if self.target is not None:
        prev_target_idx = self.target.idx
        for agent_idx, agent in enumerate(self.agent):
            if agent is not None:
                self.set_agent_start(agent_idx, prev_target_idx)
        self.target.marker = None
    # Set the new target node
    self.target = self.boxes[idx]
    self.target.marker = 'T'
    # Update agents' paths and targets
    if self.path is not None:
        for agent_idx, agent in enumerate(self.agent):
            if agent is not None:
                self.plan_path(self.current_search, self.limit, agent_idx)
                new_path_pts = [PointToVector2D(self.boxes[p]._vc) for p in self.path.path]
                agent.set_new_target(new_path_pts)
```

```python
def add_agent(self, start_idx):
    self.set_agent_start(len(self.agent), start_idx)

# New method for setting agent starting points
def set_agent_start(self, agent_idx, idx):
    if agent_idx < len(self.start):
        self.start[agent_idx].marker = None
    else:
        self.start.append(None)
        self.agent.append(None)
    self.start[agent_idx] = self.boxes[idx]
    self.start[agent_idx].marker = 'S' + str(agent_idx + 1)
```

```python
def plan_path(self, search, limit, agent_idx, speed=None):
    self.current_search = search
    self.limit = limit
    cls = SEARCHES[search]
    start = self.start[agent_idx]
    self.path = cls(self.graph, start.idx, self.target.idx, limit)
    path_pts = [PointToVector2D(self.boxes[p]._vc) for p in self.path.path]
    if self.agent[agent_idx] is None:
        start_pos = PointToVector2D(self.boxes[start.idx]._vc)
        self.agent[agent_idx] = Agent(start_pos, speed=speed if speed is not None else 5)
    if not self.agent[agent_idx].path_set:
        self.agent[agent_idx].path_pts = path_pts
        self.agent[agent_idx].path.set_pts(path_pts)
        self.agent[agent_idx].path_set = True
```

3. Modified the Main class:

In the main file, several changes have been made to implement agent movement in the BoxWorld simulation. Here's an explanation of the modifications:

- In the BoxWorldWindow class, the __init__() method has been updated to include a clock.schedule_interval() function call that schedules the update_agent() method to be called 20 times per second (1/20).
- The update_agent(self, dt) method is added to update the agents in the world. It iterates through the agents and calls the update() method for each agent to update their position according to their path.
- In the on_mouse_press() event handler, when setting the start node (agent position) or target node, the method now calls self.plan_path() with the agent's speed as an argument to plan paths for all agents based on their speeds.
- The on_key_press() event handler has been updated to handle agent speed changes. When the user presses 'A' or 'D', the method sets the search mode to A* or Dijkstra respectively, and calls self.plan_path() with a specified speed for the agent.
- The plan_path(self, speed=None) method has been modified to plan paths for all agents. It iterates through the self.world.start list and calls the self.world.plan_path() method with the agent index and speed as arguments. This ensures that each agent's path is planned individually based on their speed.
- The on_draw() method remains unchanged, as it already takes care of drawing agents through the self.world.draw() method.

These modifications in the main file allow the simulation to handle multiple moving agents, update their positions based on their paths, and plan paths individually based on each agent's speed.

```python
def __init__(self, filename, **kwargs):
    kwargs.update({
        'width': 500,
        'height': 500,
        'vsync':True,
        'resizable':True,
    })
    super(BoxWorldWindow, self).__init__(**kwargs)
    clock.schedule_interval(self.update_agent, 1/20)
    self.searching = False;
```

```python
def update_agent(self, dt):
    if self.world.agent:
        for agent in self.world.agent:
            if agent is not None:
                agent.update()
```

```python
@self.event
def on_key_press(symbol, modifiers):
    if symbol in self.mouse_modes:
        self.mouse_mode = self.mouse_modes[symbol]
        self._update_label('mouse')

    if symbol == key.A:
        if self.searching:
            return
        else:
            self.searching = True
            print("Searching with A*")
            self.search_mode = 3
            self.plan_path(speed=10)
            self._update_label('search')
    elif symbol == key.D:
        if self.searching:
            return
        else:
            self.searching = True
            print("Searching with Dijkstra")
            self.search_mode = 2
            self.plan_path(speed=3)
            self._update_label('search')

    elif symbol == key.R:
        print("reset")
        self.searching = False
        self.world.path = None
        self.world.agent = None
```

```python
@self.event
def on_mouse_press(x, y, button, modifiers):
    if button == 1:  # left
        box = self.world.get_box_by_pos(x, y)
        if box:
            if self.mouse_mode == "start":
                self.world.add_agent(box.node.idx)
                # Assign a random speed to the agent from the options 2 and 10
                agent_speed = random.choice([2, 10])
                self.plan_path(speed=agent_speed)
            elif self.mouse_mode == "target":
                self.world.set_target(box.node.idx)
            else:
                box.set_kind(self.mouse_mode)
            self.world.reset_navgraph()
            self.plan_path()
            self._update_label("status", "graph changed")
```

```python
def plan_path(self, speed=None):
    for agent_idx in range(len(self.world.start)):
        self.world.plan_path(search_modes[self.search_mode], self.limit, agent_idx, speed=speed)
        self._update_label('status', 'path planned')
        print(self.world.path.report(verbose=3))
```

4. <u>Reused the existing codes to build the box world and path planning:</u>

The original BoxWorld codebase provides a solid foundation for path planning and navigation, with several essential components already in place. We have reused the following files from the codebase for this spike:

i) **graph.py:** Contains the Graph class and associated methods for creating and managing the navigation graph.

ii) **graphics.py:** Provides the EGI (Extended Graphics Interface) for drawing primitives and text on the screen.

iii) **path.py:** Contains the Path class and associated methods for managing agent paths.

iv) **matrix33.py:** Provides Matrix33 class for 3x3 matrix operations (used for calculations within the path and graph classes).

v) **point2d.py**: Contains the Point2D class for handling 2D points and their operations.

vi) **searches.py:** Contains different search algorithms (BFS, DFS, Dijkstra, A*) for path planning.

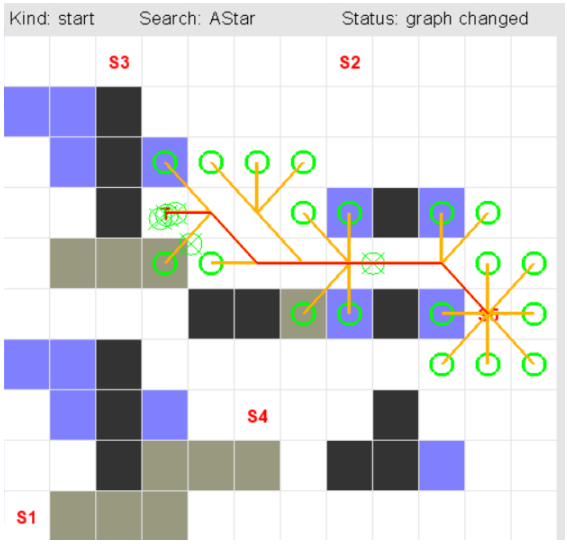
vii) **vector2d.py:** Provides the Vector2D class for 2D vector operations (used for agent movement and calculations).

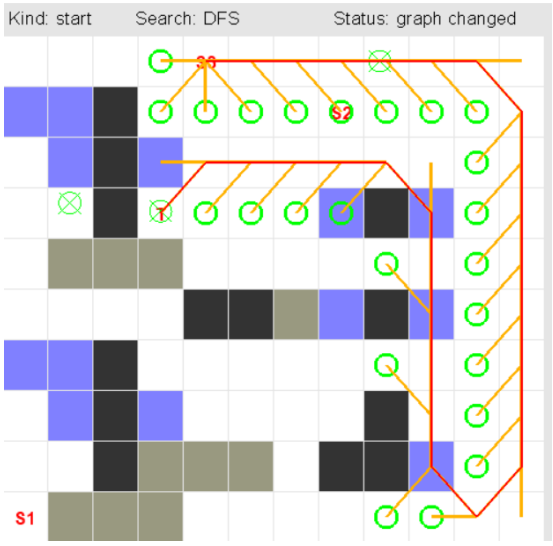
What we found out:

As a result of these modifications and extensions, the BoxWorld simulation now supports multiple agents that can plan paths using heuristic search algorithms and navigate the environment using steering-based movement. The simulation can handle different agent speeds and allows users to interact with the agents and the environment for an engaging experience.

Flexibility of Heuristic Search Algorithms: The use of heuristic search algorithms, such as A* and Dijkstra, proved to be highly adaptable for path planning in a dynamic environment. These algorithms were able to generate efficient paths for the agents despite changes in the environment, such as the addition or removal of obstacles.

Importance of Agent Speed Variation: Allowing agents to have different speeds added an interesting layer of complexity to the simulation. Agents with different speeds could reach their targets at varying times, which could be useful for modelling real-world scenarios where agents have different capabilities or priorities.





Open Issues and Recommendations:

The current implementation directly updates the agent's velocity to follow the path. Although this approach works for the given simulation, utilizing more advanced steering behaviors like the "seek" function could have potentially led to smoother agent movement and better overall performance.