project_notebook

April 25, 2018

1 Implementing a Route Planner

In this project you will use A* search to implement a "Google-maps" style route planning algorithm.

1.1 The Map

```
In [ ]: # Run this cell first!

        from helpers import Map, load_map_10, load_map_40, show_map
        import math

        %load_ext autoreload
        %autoreload 2
```

1.1.1 Map Basics

```
In [ ]: map_10 = load_map_10()
        show_map(map_10)
```

The map above (run the code cell if you don't see it) shows a disconnected network of 10 intersections. The two intersections on the left are connected to each other but they are not connected to the rest of the road network. This map is quite literal in its expression of distance and connectivity. On the graph above, the edge between 2 nodes(intersections) represents a literal straight road not just an abstract connection of 2 cities.

These `Map` objects have two properties you will want to use to implement A* search: `intersections` and `roads`

Intersections

The `intersections` are represented as a dictionary.

In this example, there are 10 intersections, each identified by an x,y coordinate. The coordinates are listed below. You can hover over each dot in the map above to see the intersection number.

```
In [ ]: map_10.intersections
```

Roads

The `roads` property is a list where `roads[i]` contains a list of the intersections that intersection `i` connects to.

```
In [ ]: # this shows that intersection 0 connects to intersections 7, 6, and 5
        map_10.roads[0]
```

```
In [ ]: # This shows the full connectivity of the map
        map_10.roads
```

```
In [ ]: # map_40 is a bigger map than map_10
        map_40 = load_map_40()
        show_map(map_40)
```

1.1.2 Advanced Visualizations

The map above shows a network of roads which spans 40 different intersections (labeled 0 through 39).

The `show_map` function which generated this map also takes a few optional parameters which might be useful for visualizaing the output of the search algorithm you will write.

- `start` - The "start" node for the search algorithm.
- `goal` - The "goal" node.
- `path` - An array of integers which corresponds to a valid sequence of intersection visits on the map.

```
In [ ]: # run this code, note the effect of including the optional
        # parameters in the function call.
        show_map(map_40, start=5, goal=34, path=[5,16,37,12,34])
```

1.2 The Algorithm

1.2.1 Writing your algorithm

The algorithm you write will be responsible for generating a `path` like the one passed into `show_map` above. In fact, when called with the same map, start and goal, as above you algorithm should produce the path `[5, 16, 37, 12, 34]`

```
> shortest_path(map_40, 5, 34)
[5, 16, 37, 12, 34]
```

```
In [ ]: # Do not change this cell
        # When you write your methods correctly this cell will execute
        # without problems
        class PathPlanner():
            """Construct a PathPlanner Object"""
            def __init__(self, M, start=None, goal=None):
                """ """
                self.map = M
                self.start= start
                self.goal = goal
                self.closedSet = self.create_closedSet() if goal != None and start != None else
                self.openSet = self.create_openSet() if goal != None and start != None else None
                self.cameFrom = self.create_cameFrom() if goal != None and start != None else No
        self.gScore = self.create_gScore() if goal != None and start != None else None
        self.fScore = self.create_fScore() if goal != None and start != None else None
        self.path = self.run_search() if self.map and self.start != None and self.goal !

    def get_path(self):
        """ Reconstructs path after search """
        if self.path:
            return self.path
        else :
            self.run_search()
            return self.path

    def reconstruct_path(self, current):
        """ Reconstructs path after search """
        total_path = [current]
        while current in self.cameFrom.keys():
            current = self.cameFrom[current]
            total_path.append(current)
        return total_path

    def run_search(self):
        """ """
        if self.map == None:
            raise(ValueError, "Must create map before running search. Try running PathPl
        if self.goal == None:
            raise(ValueError, "Must create goal node before running search. Try running
        if self.start == None:
            raise(ValueError, "Must create start node before running search. Try running

        self.closedSet = self.closedSet if self.closedSet != None else self.create_close
        self.openSet = self.openSet if self.openSet != None else  self.create_openSet()
        self.cameFrom = self.cameFrom if self.cameFrom != None else  self.create_cameFro
        self.gScore = self.gScore if self.gScore != None else  self.create_gScore()
        self.fScore = self.fScore if self.fScore != None else  self.create_fScore()

        while not self.is_open_empty():
            current = self.get_current_node()

            if current == self.goal:
                self.path = [x for x in reversed(self.reconstruct_path(current))]
                return self.path
            else:
                self.openSet.remove(current)
                self.closedSet.add(current)

            for neighbor in self.get_neighbors(current):
                if neighbor in self.closedSet:
                    continue    # Ignore the neighbor which is already evaluated.
                    if not neighbor in self.openSet:    # Discover a new node
                        self.openSet.add(neighbor)

                    # The distance from start to a neighbor
                    #the "dist_between" function may vary as per the solution requirements.
                    if self.get_tenative_gScore(current, neighbor) >= self.get_gScore(neighb
                        continue        # This is not a better path.

                    # This path is the best until now. Record it!
                    self.record_best_path_to(current, neighbor)
                print("No Path Found")
                self.path = None
                return False
```

Create the following methods:

```
In [ ]: def create_closedSet(self):
            """ Creates and returns a data structure suitable to hold the set of nodes already e
            # TODO: return a data structure suitable to hold the set of nodes already evaluated
            return set()

In [ ]: def create_openSet(self):
            """ Creates and returns a data structure suitable to hold the set of currently disco
            that are not evaluated yet. Initially, only the start node is known."""
            if self.start != None:
                # TODO: return a data structure suitable to hold the set of currently discovered
                # that are not evaluated yet. Make sure to include the start node.
                return

            raise(ValueError, "Must create start node before creating an open set. Try running P

In [ ]: def create_cameFrom(self):
            """Creates and returns a data structure that shows which node can most efficiently b
            for each node."""
            # TODO: return a data structure that shows which node can most efficiently be reache
            # for each node.

In [ ]: def create_gScore(self):
            """Creates and returns a data structure that holds the cost of getting from the star
            The cost of going from start to start is zero."""
            # TODO:  a data structure that holds the cost of getting from the start node to that
            # for each node. The cost of going from start to start is zero. The rest of the node


In [ ]: def create_fScore(self):
            """Creates and returns a data structure that holds the total cost of getting from th
            by passing by that node, for each node. That value is partly known, partly heuristic
            For the first node, that value is completely heuristic."""
            # TODO:  a data structure that holds the total cost of getting from the start node t
            # by passing by that node, for each node. That value is partly known, partly heurist
            # For the first node, that value is completely heuristic. The rest of the node's val
            # set to infinity.
```

```
In [ ]: def _reset(self):
            """Private method used to reset the closedSet, openSet, cameFrom, gScore, fScore, an
            self.closedSet = None
            self.openSet = None
            self.cameFrom = None
            self.gScore = None
            self.fScore = None
            self.path = self.run_search() if self.map and self.start and self.goal else None
```

```
In [ ]: def set_map(self, M):
            """Method used to set map attribute """
            self._reset(self)
            self.start = None
            self.goal = None
            # TODO: Set map to new value.
```

```
In [ ]: def set_start(self, start):
            """Method used to set start attribute """
            self._reset(self)
            # TODO: Set start value. Remember to remove goal, closedSet, openSet, cameFrom, gSco
            # and path attributes' values.
```

```
In [ ]: def set_goal(self, goal):
            """Method used to set goal attribute """
            self._reset(self)
            # TODO: Set goal value.
```

```
In [ ]: def get_current_node(self):
            """ Returns the node in the open set with the lowest value of f(node)."""
            # TODO: Return the node in the open set with the lowest value of f(node).
```

```
In [ ]: def get_neighbors(self, node):
            """Returns the neighbors of a node"""
            # TODO: Return the neighbors of a node
```

```
In [ ]: def get_gScore(self, node):
            """Returns the g Score of a node"""
            # TODO: Return the g Score of a node
```

```
In [ ]: def get_tenative_gScore(self, current, neighbor):
            """Returns the tenative g Score of a node"""
            # TODO: Return the tenative g Score of the current node
            # plus distance from the current node to it's neighbors
```

```
In [ ]: def is_open_empty(self):
            """returns True if the open set is empty. False otherwise. """
            # TODO: Return True if the open set is empty. False otherwise.
```

```
In [ ]: def distance(self, node_1, node_2):
            """ Computes the Euclidean L2 Distance"""
            # TODO: Compute and return the Euclidean L2 Distance
```

```
In [ ]: def heuristic_cost_estimate(self, node):
            """ Returns the heuristic cost estimate of a node """
            # TODO: Return the heuristic cost estimate of a node
```

```
In [ ]: def calculate_fscore(self, node):
            """Calculate the f score of a node. """
            # TODO: Calculate and returns the f score of a node.
            # REMEMBER F = G + H
```

```
In [ ]: def record_best_path_to(self, current, neighbor):
            """Record the best path to a node """
            # TODO: Record the best path to a node, by updating cameFrom, gScore, and fScore
```

```
In [ ]: PathPlanner.create_closedSet = create_closedSet
        PathPlanner.create_openSet = create_openSet
        PathPlanner.create_cameFrom = create_cameFrom
        PathPlanner.create_gScore = create_gScore
        PathPlanner.create_fScore = create_fScore
        PathPlanner._reset = _reset
        PathPlanner.set_map = set_map
        PathPlanner.set_start = set_start
        PathPlanner.set_goal = set_goal
        PathPlanner.get_current_node = get_current_node
        PathPlanner.get_neighbors = get_neighbors
        PathPlanner.get_gScore = get_gScore
        PathPlanner.get_tenative_gScore = get_tenative_gScore
        PathPlanner.is_open_empty = is_open_empty
        PathPlanner.distance = distance
        PathPlanner.heuristic_cost_estimate = heuristic_cost_estimate
        PathPlanner.calculate_fscore = calculate_fscore
        PathPlanner.record_best_path_to = record_best_path_to
```

```
In [ ]: planner = PathPlanner(map_40, 5, 34)
        path = planner.path
        if path == [5, 16, 37, 12, 34]:
            print("great! Your code works for these inputs!")
        else:
            print("something is off, your code produced the following:")
            print(path)
```

1.2.2 Testing your Code

If the code below produces no errors, your algorithm is behaving correctly. You are almost ready to submit! Before you submit, go through the following submission checklist:

Submission Checklist

1. Does my code pass all tests?
2. Does my code implement `A*` search and not some other search algorithm?
3. Do I use an **admissible heuristic** to direct search efforts towards the goal?
4. Do I use data structures which avoid unnecessarily slow lookups?

When you can answer "yes" to all of these questions, submit by pressing the Submit button in the lower right!

```
In [ ]: from test import test

        test(PathPlanner)
```

1.3 Questions

Instructions Answer the following questions in your own words. We do not you expect you to know all of this knowledge on the top of your head. We expect you to do research and ask question. However do not merely copy and paste the answer from a google or stackoverflow. Read the information and understand it first. Then use your own words to explain the answer.

- How would you explain A-Star to a family member(layman)?

** ANSWER **: A-start algorithm is a single source shortest path algorithm that returns the shortest path from a single source to a single goal

- How does A-Star search algorithm differ from Uniform cost search? What about Best First search?

** ANSWER **:A-star algorithm run faster than Uniform cost search because it considers heuristic cost that would make the search path always towards the direction of goal. However, uniform cost search always first find the shortest path from the source instead of the goal, therefore it can not always find the goal earlier than A-star algorithm. So does BFS.

- What is a heuristic?

** ANSWER **: we estimate the cost from current vertex to our goal

- What is a consistent heuristic?

** ANSWER **:A heuristic is consistent if the cost from the current node to a successor node, plus the estimated cost from the successor node to the goal is less than or equal to the estimated cost from the current node to the goal

- What is a admissible heuristic?

** ANSWER **:the heuristic is less than the real cost from current vertex to the goal

- _Some__ admissible heuristic are consistent. *CHOOSE ONE*
  - All
  - Some
  - None

** ANSWER **:

- _All__ Consistent heuristic are admissible. *CHOOSE ONE*
  - All
  - Some
  - None

** ANSWER **: