project_notebook

May 20, 2020

1 Implementing a Route Planner

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

```
In [10]: # Run this cell first!

         from helpers import Map, load_map, show_map
         from student_code import shortest_path

         %load_ext autoreload
         %autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1.0.1 Map Basics

```
In [11]: map_10 = load_map('map-10.pickle')
         show_map(map_10)
         map_10.intersections
```

```
Out[11]: {0: [0.7798606835438107, 0.6922727646627362],
          1: [0.7647837074641568, 0.3252670836724646],
          2: [0.7155217893995438, 0.20026498027300055],
          3: [0.7076566826610747, 0.3278339270610988],
          4: [0.8325506249953353, 0.02310946309985762],
          5: [0.49016747075266875, 0.5464878695400415],
          6: [0.8820353070895344, 0.6791919587749445],
          7: [0.46247219371675075, 0.6258061621642713],
          8: [0.11622158839385677, 0.11236327488812581],
          9: [0.1285377678230034, 0.3285840695698353]}
```

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. 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 [12]: map_10.roads

Out[12]: [[7, 6, 5],
          [4, 3, 2],
          [4, 3, 1],
          [5, 4, 1, 2],
          [1, 2, 3],
          [7, 0, 3],
          [0],
          [0, 5],
          [9],
          [8]]
```

Roads

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

```
In [13]: # this shows that intersection 0 connects to intersections 7, 6, and 5
         map_10.roads[0]

Out[13]: [7, 6, 5]

In [14]: # This shows the full connectivity of the map
         map_10.roads

Out[14]: [[7, 6, 5],
          [4, 3, 2],
          [4, 3, 1],
          [5, 4, 1, 2],
          [1, 2, 3],
          [7, 0, 3],
          [0],
          [0, 5],
          [9],
          [8]]

In [17]: # map_40 is a bigger map than map_10
         map_40 = load_map('map-40.pickle')
         show_map(map_40)
         map_40.intersections
```

```
Out[17]: {0: [0.7801603911549438, 0.49474860768712914],
          1: [0.5249831588690298, 0.14953665513987202],
          2: [0.8085335344099086, 0.7696330846542071],
          3: [0.2599134798656856, 0.14485659826020547],
          4: [0.7353838928272886, 0.8089961609345658],
          5: [0.09088671576431506, 0.7222846879290787],
          6: [0.313999018186756, 0.01876171413125327],
          7: [0.6824813442515916, 0.8016111783687677],
          8: [0.20128789391122526, 0.43196344222361227],
          9: [0.8551947714242674, 0.9011339078096633],
          10: [0.7581736589784409, 0.24026772497187532],
          11: [0.25311953895059136, 0.10321622277398101],
          12: [0.4813859169876731, 0.5006237737207431],
          13: [0.9112422509614865, 0.1839028760606296],
          14: [0.04580558670435442, 0.5886703168399895],
          15: [0.4582523173083307, 0.1735506267461867],
          16: [0.12939557977525573, 0.690016328140396],
          17: [0.607698913404794, 0.362322730884702],
          18: [0.719569201584275, 0.13985272363426526],
          19: [0.8860336256842246, 0.891868301175821],
          20: [0.4238357358399233, 0.026771817842421997],
          21: [0.8252497121120052, 0.9532681441921305],
          22: [0.47415009287034726, 0.7353428557575755],
          23: [0.26253385360950576, 0.9768234503830939],
          24: [0.9363713903322148, 0.13022993020357043],
          25: [0.6243437191127235, 0.21665962402659544],
          26: [0.5572917679006295, 0.2083567880838434],
          27: [0.7482655725962591, 0.12631654071213483],
          28: [0.6435799740880603, 0.5488515965193208],
          29: [0.34509802713919313, 0.8800306496459869],
          30: [0.021423673670808885, 0.4666482714834408],
          31: [0.640952694324525, 0.3232711412508066],
          32: [0.17440205342790494, 0.9528527425842739],
          33: [0.1332965908314021, 0.3996510641743197],
          34: [0.583993110207876, 0.42704536740474663],
          35: [0.3073865727705063, 0.09186645974288632],
          36: [0.740625863119245, 0.68128520136847],
          37: [0.3345284735051981, 0.6569436279895382],
          38: [0.17972981733780147, 0.999395685828547],
          39: [0.6315322816286787, 0.7311657634689946]}
```

1.0.2 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 [18]: # run this code, note the effect of including the optional
         # parameters in the function call.
         show_map(map_40, start=8, goal=24, path=[8, 14, 16, 37, 12, 17, 10, 24])
```

1.0.3 Writing your algorithm

You should open the file `student_code.py` in another tab and work on your algorithm there. Do that by selecting `File > Open` and then selecting the appropriate file.

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 [21]: import math

         def insert(heap, node, element):
             n = len(heap)
             if n == 0:
                 heap.append((node, element))
                 return heap
             heap.append((node, element))
             n = len(heap)-1
             return siftUp(heap, n, node)



         def swap(heap, first, last):
             tmp = heap[last]
             heap[last] = heap[first]
             heap[first] = tmp
             return



         def siftUp(heap, position, node):
             parent = (position - 1) // 2
             if position <= 0 or heap[position][1] >= heap[parent][1]:
                 return heap
             swap(heap, parent, position)
             return siftUp(heap, parent, node)
```

```
def extract(heap):
    n = len(heap)
    if n == 0:
        return "Heap is empty"
    heap[0] = heap[n-1]
    heap.pop()
    n = len(heap) - 1
    return siftDown(heap, n - n)

def heappopmin(heap):
    val = heap[0]
    new_top = heap.pop()
    if len(heap) == 0:
        return val
    heap[0] = new_top
    siftDown(heap, 0)
    return val[0], val[1]

def siftDown(heap, position):
    left = (2 * position) + 1
    right = (2 * position) + 2
    if left > len(heap)-1 or right > len(heap)-1:
        if left <= len(heap)-1 and heap[left][1] < heap[position][1]:
            return swap(heap, position, left)
        if right <= len(heap)-1 and heap[right][1] < heap[position][1]:
            return swap(heap, position, right)
        return heap
    if heap[left][1] < heap[right][1] and heap[position][1] > heap[left][1]:
        swap(heap, position, left)
        return siftDown(heap, left)
    if heap[right][1] < heap[left][1] and heap[position][1] > heap[right][1]:
        swap(heap, position, right)
        return siftDown(heap, right)
    return heap



def euclidean(intersections, start, goal):
        return math.sqrt(((intersections[start][0] - intersections[goal][0]) ** 2) + ((


def shortest_path(G, start_node, end_node):
    heap = [(start_node, 0)]
    dist_so_far = {start_node:0}
    visited = {}
    # path = {start_node: None}
    path_to_node = {start_node:[start_node]}
    while len(visited) < len(G.roads):
```

```
                current_node, current_score = heappopmin(heap)
                visited[current_node] = current_score

                if len(dist_so_far) == 0:
                    return path_to_nodes

                del dist_so_far[current_node]

                node = G.roads[current_node]
                i = 0
                while i < len(node):
                    if node[i] not in visited:
                        heuristic_dist = current_score + euclidean(G.intersections, current_nod
                        if node[i] not in dist_so_far:
                            insert(heap, node[i], heuristic_dist)
                            dist_so_far[node[i]] = heuristic_dist
                            # path[node[i]] = current_node
                            path_to_node[node[i]] = path_to_node[current_node] + [node[i]]

                        elif heuristic_dist < dist_so_far[node[i]]:
                            index = heap.index((node[i], dist_so_far.get(node[i])))
                            heap[index] = (node[i], heuristic_dist)
                            siftUp(heap, index, heap[index][0])
                            dist_so_far[node[i]] = heuristic_dist
                            # path[node[i]] = current_node
                            path_to_node[node[i]] = path_to_node[current_node] + [node[i]]
                    i += 1
            return path_to_node[end_node]

        path = shortest_path(map_40, 5, 34)
        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)
```

```
great! Your code works for these inputs!
```

1.0.4 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 [22]: from test import test

         test(shortest_path)
```

```
All tests pass! Congratulations!
```