Artificial Intelligence Homework

Gabriela-Loredana Dinu

June 4, 2018

Group CEN2.1 B

2nd Year

Problem statement

Map search problem for n cities. Having n cities and m roads, find the shortest path in order to reach city B from city A. Let us assume that each city is given by its coordinates.

Develop implementations and two heuristic functions for each A* and Recursive-Best-First-Search algorithms using a programming language of your choice.

Develop an application that allows you to test and evaluate your algorithm implementation.

Develop nontrivial data sets containing at least 10 test cases of various sizes: small, medium, large, and very large, such that overall the data files should fit in the 8MB archive that must be delivered.

Pseudocode of algorithms

Here are the important functions employed by the program:

```
The actions at a graph node are just reaching its neighbors

function actions(self, A) returns an action
1.       return list(self.graph.get(A).keys())
```

```
function path_cost(self, cost_so_far, A, action, B) returns the path
    cost computed from the start node to the current node
1.       return cost_so_far + (self.graph.get(A, B) or infinity)
```

```
function find_min_edge(self) returns the edge with the minimum cost
1.     m = infinity
2.     for d in self.graph.graph_dict.values():
3.          local_min <- min(d.values())
4.          m <- min(m, local_min)
```

```
The result of going to a neighbor is just that neighbor
function result(self, state, action) returns an action
1.       return action
```

Heuristics

- Manhattan heuristic which computes the Manhattan-distance between 2 states
- Euclidean heuristic which returns the Euclidean distance between 2 states

```
 function h(self,node)
1.       locs <-coordinates of city from graph
2.       return euclidian_distance between current node and goal
```

```
function h(self, node):
1.     locs <-coordinates of city from graph
2.     return euclidian_distance between current node and goal
```

```
function manhattan_distance(a, b) returns the manhattan distance between
    2 points represented with coordinates x and y
1.  x1, y1 <- a
2.  x2, y2 <- b
```

```
3.  return abs((x2-x1)) + abs((y2-y1))
```

```
function def euclidian_distance(a,b) returns the euclidian distance
    between 2 points represented with coordinates x and y
1.  x1, y1 <- a
2.  x2, y2 <- b
3.  return abs(sqrt((x2-x1)^2 + (y2-y1)^2)
```

Auxiliary functions

```
function
    random_string_generator(size=random.randint(1,10),chars=string.ascii_lowercase
    + string.digits) returns a string formed by concatenated random
    characters
1.    for iterator in range 0,size
2.     x<- join(random.choice(chars))
3.     return x
```

Functions used to make an undirected graph

```
function make_undirected(self) makes a digraph into an undirected graph
    by adding symmetric edges

1.       for a in list(graph_dict.keys())
2.           for (b, dist) in self.graph_dict[a].items()
3.               self.connect1(b, a, dist)
```

```
function connect1(self, A, B, distance) adds a link from A to B of given
    distance, in one direction only.
1.       self.graph_dict.setdefault(A, {}) [B] <- distance
```

```
function self.connect1(A, B, distance) adds a link from A and B of given
    distance, and also add the inverse link if the graph is undirected.
1.       if not self.directed:
2.           self.connect1(B, A, distance)
```

Function used to make a random graph

```
 Construct a random graph, with the specified nodes, and random links.
    The nodes are laid out randomly on a (width x height) rectangle.
    Then each node is connected to the min_links nearest neighbors.
    Because inverse links are added, some nodes will have more
        connections.
    The distance between nodes is the hypotenuse times curvature(),
    where curvature() defaults to a random number between 1.1 and 1.5.

function RandomGraph(nodes=list(range(10)), min_links=4, width=400,
    height=300,
              curvature=lambda: random.uniform(1.1, 1.5)) return a
                  random graph being given a list of strigs
                  (representing the cities) and a minimum number of
                  links (roads) to add for each city

1.   g <- UndirectedGraph()
2.   test2 <- open('random_graph.txt','w')
3.   g.locations <- {}
4.   # Build the cities
5.   for node in nodes
6.       g.locations[node] <- (random.randrange(width),
    random.randrange(height))
    # Build roads from each city to at least min_links nearest neighbors.
7.   for i in range(min_links)
8.       for node in nodes
9.           if len(g.get(node)) < min_links:
10.              here <- g.locations[node]

11.              def distance_to_node(n)
12.                  if n is node or g.get(node, n)
13.                      return infinity
14.                  return distance(g.locations[n], here)
15.              neighbor <- argmin(nodes, key=distance_to_node)
16.              d <- distance(g.locations[neighbor], here) *
    curvature()
17.              g.connect(node, neighbor, int(d))
write the random graph in a file
18.
    test2.write("{},{},{}\n".format(str((node)),str((neighbor)),
                    int(d)))
19.   return g
```

Application outline

```
The application contains the modules main.py, distances.py, graphs.py,
    graphs_heuristics.py, search.y and utils.py
```

The problem was implemented using AIMA Framework

The source file graphs.py contains:

- class Graph
  A graph connects nodes (vertices) by edges (links). Each edge can also have a length associated with it. The constructor call is something like: g = Graph('A': 'B': 1, 'C': 2) this makes a graph with 3 nodes, A, B, and C, with an edge of length 1 from A to B, and an edge of length 2 from A to C. You can also do: g = Graph('A': 'B': 1, 'C': 2, directed=False) This makes an undirected graph, so inverse links are also added. The graph stays undirected; if you add more links with g.connect('B', 'C', 3), then inverse link is also added. You can use g.nodes() to get a list of nodes, g.get('A') to get a dict of links out of A, and g.get('A', 'B') to get the length of the link from A to B. 'Lengths' can actually be any object at all, and nodes can be any hashable object.
  This class has methods such as:

  – def make undirected(self) used to make a digraph into an undirected graph by adding symmetric edges

  – def connect(self, A, B, distance=1) used to add a link from A and B of given distance, and also add the inverse link if the graph is undirected

  – def connect1(self, A, B, distance) used to add a link from A to B of given distance, in one direction only.

  – def get(self, a, b=None) used to return a link distance or a dict of node: distance entries.

  – def nodes(self) used to return a list of nodes in the graph.
- class Undirected Graph which builds a Graph where every edge (including future ones) goes both ways.
- class RandomGraph which constructs a random graph, with the specified nodes, and random links
- class GraphProblem which implements the problem of searching a graph from one node to another.

The source file distances.py contains helper functions used to comute the manhattan and euclidian distances between 2 points having x and y coordinates

The source file graphs$_{h}euristics.pycontains$:

- class GraphProblemEuclidian which extendes class GraphProblem, implementing the heuristic using the euclidian distance

- class GraphProblemManhattan which extendes class GraphProblem, implementing the heuristic using the manhattan distance

The source file search.py contains the class Problem from which the class GraphProblem is extended and it also contains the Informed (Heuristic) Search A* and the search Recursive Best First Search. It also contains a function used to compare 2 types of search by succesors, goal tests and states.

The source file utils.py contains utilities such as queues.

Input data format

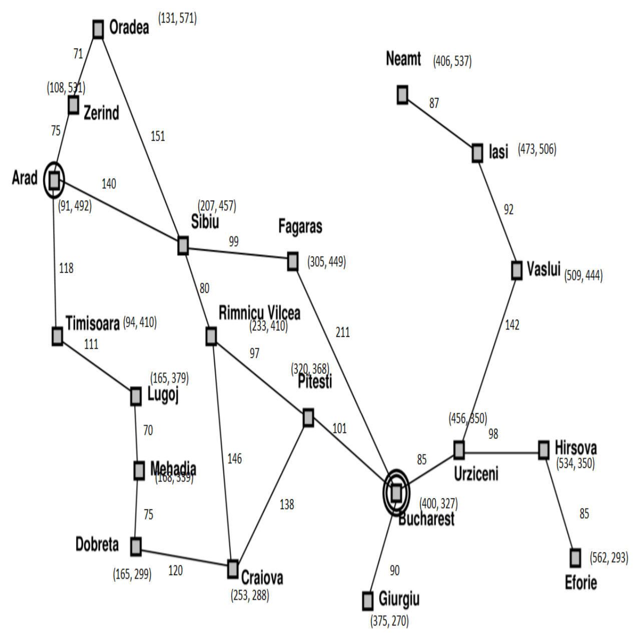
The input data is represented by initial state,goal state and the graph. The graph is represented as a dictionary Let us exemplify on the simplified version of Map of Romania:

Cities and distances between them

romania$_{m}ap = UndirectedGraph(dict(Arad = dict(Zerind = 75, Sibiu = 140, Timisoara = 118), Bucharest = dict(Urziceni = 85, Pitesti = 101, Giurgiu = 90, Fagaras = 211), Craiova = dict(Drobeta = 120, Rimnicu = 146, Pitesti = 138), Drobeta = dict(Mehadia = 75), Eforie = dict(Hirsova = 86), Fagaras = dict(Sibiu = 99), Hirsova = dict(Urziceni = 98), Iasi = dict(Vaslui = 92, Neamt = 87), Lugoj = dict(Timisoara = 111, Mehadia = 70), Oradea = dict(Zerind = 71, Sibiu = 151), Pitesti = dict(Rimnicu = 97), Rimnicu = dict(Sibiu = 80), Urziceni = dict(Vaslui = 142)))$

Locations of each city given by its coordinates romania$_{m}ap.locations = dict(Arad = (91, 492), Bucharest = (400, 327), Craiova = (253, 288), Drobeta = (165, 299), Eforie = (562, 293), Fagaras = (305, 449), Giurgiu = (375, 270), Hirsova = (534, 350), Iasi = (473, 506), Lugoj = (165, 379), Mehadia = (168, 339), Neamt = (406, 537), Oradea = (131, 571), Pitesti = (320, 368), Rimnicu = (233, 410), Sibiu = (207, 457), Timisoara = (94, 410), Urziceni = (456, 350), Vaslui = (509, 444), Zerind = (108, 531))$

The actual Map is:



Output data format

The output data is represented by a tuple of actions that are to be applied in order to reach the goal state. A* and RBFS will trace-back the solution from goal to the root, reversing the tuple, yielding the solution. In case of no solution, the algorithms will return none. The output also contains the time it took for each search with each heuristic to find the solution, and a comparison between searches with each heuristic

Experiments and results

In order to do the experiments it is used the function random_string_generator (size=random.randint(1,10),chars=string.ascii_lowercase + string.digits) in module Random_string_generator.py which concatenates a random number of characters(between 1 and 10) into a strings. The function is called in a for loop(in order to generate more strings)in the main.p module and each generated string is appended into a list. Then, in order to build the graph, it is called the function RandomGraph(nodes=list(range(10)), min_links=4, width=400, height=300, curvature=lambda: random.uniform(1.1, 1.5)) from the graphs.py module, which builds a graph with random links between the cities/strings previousley generated, and assigns to each city a random location.

One of the tests was:

```
for 50 cities connected by 111 edges
output:
path from t9 to 7j

astar_search cu euristica euclidiana {}
 0.014784336090087 89
['zj', 'k8', 'af', 'wq', '7j']
astar_search cu euristica manahttan {}
 0.0013577938079833984
['zj', 'k8', 'af', 'wq', '7j']
RBFS cu euristica euclidiana {}
 0.07838058471679688
['zj', 'k8', 'af', 'wq', '7j']
RBFS cu euristica manhattan {}
 0.0018944740295410156
['zj', 'k8', 'af', 'wq', '7j']
Searcher                    A* h1(n)           A* h2(n)
astar_search                <  20/  22/  84/7j>  <  11/  13/  46/7j>
recursive_best_first_search < 442/ 443/1917/7j> < 39/ 40/ 167/7j>



Input generated using the functions for randomly generating input:
In "random_graph_locations.txt", containing the city names and locations:
```

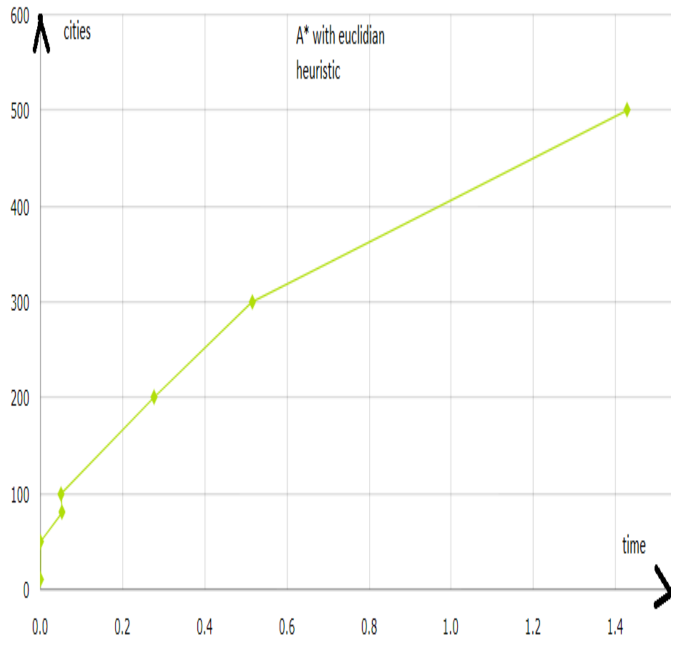
```
{'1h': (98, 171), 't9': (385, 192), 'x2': (263, 38), '3b': (43, 231),
    '25': (308, 220), 'xa': (348, 299), '83': (360, 296), 'wp':
(91, 234), 'gg': (177, 291), 'jn': (141, 32), 'q7': (184, 26), 'yk':
    (259, 211), 'kv': (82, 236), '4z': (50, 128), 'zl': (102, 3),
'7j': (176, 299), 'ci': (145, 278), 'rb': (13, 202), '28': (232, 150),
    'wc': (47, 198), 'af': (304, 285), '5x': (386, 288), '3c': (353,
    34),
 'vi': (239, 162), 'oh': (124, 24), 'pc': (79, 192), '46': (135, 116),
     '14': (92, 161), 'yg': (288, 4), '3m': (143, 204), '9q': (48, 30),
 'of': (302, 108), '72': (321, 26), 'jy': (349, 247), 'wq': (260, 286),
     'sl': (272, 146), '6r': (328, 140), 'ph': (340, 198), 'k2': (369,
     68),
 'zj': (328, 243), 'ul': (87, 259), 'dp': (119, 113), 'k8': (319, 241),
     'dj': (274, 108), 'qx': (104, 208), '6e': (69, 30), 'jb': (321,
     284),
 'ua': (372, 55), 'w9': (242, 168)}

In "random_graph.txt", containing the links betwwen cities and their
    euclidian distance:

1h,14,17
t9,ph,65
x2,yg,49
3b,wc,41
25,k8,33
xa,83,13
83,5x,36
wp,kv,10
gg,7j,9
jn,oh,26
q7,jn,53
yk,w9,56
kv,ul,34
4z,14,77
zl,oh,41
7j,ci,46
ci,gg,43
rb,wc,45
28,28,0
wc,pc,46
af,jb,20
5x,xa,59
3c,ua,42
vi,w9,7
oh,6e,70
pc,1h,40
46,dp,21
14,pc,47
yg,72,44
```

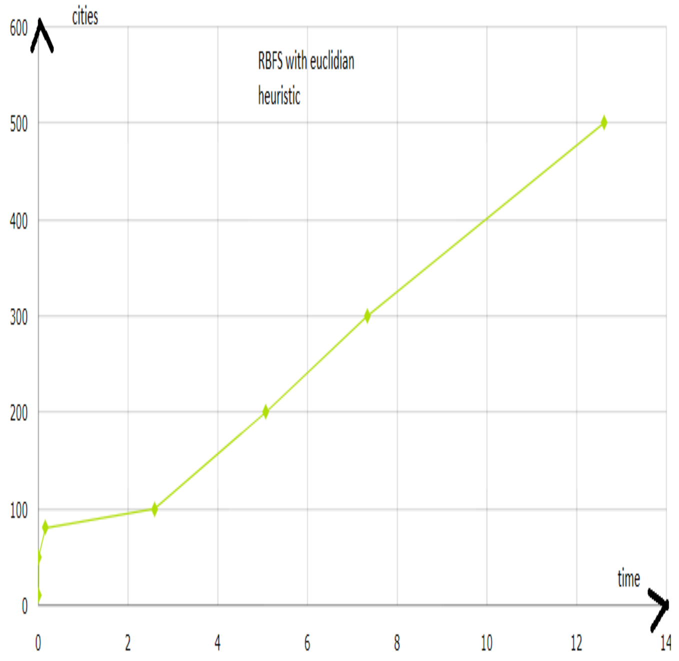
```
3m,qx,58
9q,6e,30
of,dj,37
72,3c,41
jy,zj,24
wq,af,53
sl,vi,41
6r,of,52
ph,25,57
k2,ua,15
zj,k8,13
ul,wp,35
dp,14,75
k8,jy,37
dj,sl,56
qx,wp,32
6e,zl,60
jb,xa,41
28,28,0
ua,72,87
w9,28,29
1h,qx,47
t9,jy,87
x2,72,74
3b,kv,47
25,zj,39
xa,af,53
83,jb,58
wp,pc,61
gg,wq,105
jn,zl,63
q7,oh,75
yk,25,58
kv,qx,44
4z,1h,81
zl,9q,79
7j,wq,106
ci,ul,90
rb,3b,56
28,28,0
wc,kv,76
af,k8,59
5x,jy,75
3c,k2,43
vi,28,18
46,14,81
yg,3c,91
3m,1h,72
9q,oh,88
of,sl,58
```

```
wq,jb,77
sl,w9,42
6r,sl,76
ph,zj,68
k2,72,71
ul,3b,66
dp,1h,73
dj,28,82
6e,jn,98
ua,of,109
t9,zj,102
x2,dj,79
83,jy,57
gg,3m,126
q7,x2,111
yk,vi,69
4z,wc,101
7j,ul,126
ci,wp,83
rb,pc,91
5x,jb,95
46,1h,74
yg,ua,124
3m,wp,88
9q,jn,120
6r,ph,81
k2,of,107
dp,4z,89
t9,6r,85
q7,zl,103
yk,sl,82
rb,kv,103
46,jn,119
```

Conclusions

Statistics





There are similarities between A* search with manhattan and euclidian heuristics.
There are similarities between RBFS search with manhattan and euclidian heuristics.

References

1)www.stackoverflow.com

2)https://github.com/aimacode/aima-python

3)http://www.sharelatex.com 4)https://en.wikipedia.org/wiki/A*_search_algorithm