106119029

September 1, 2021

Program in Python to implement Heuristics and search strategy for Travelling salesperson problem.

Roll Number : 106119029

AI/ML Lab 3

Q - Write a program in Python to implement Heuristics and search strategy for Travelling salesperson problem

Google Collab Link

[25]:
```
import itertools
import random
```

[26]:
```
# class that represents a graph
class Graph:
        def __init__(self, amount_vertices):
                self.edges = {} # dictionary of edges
                self.vertices = set() # set of vertices
                self.V= amount_vertices # amount of vertices

        # adds a edge linking "src" in "dest" with a "cost"
        def addEdge(self, src, dest, cost = 0):
                # checks if the edge already exists
                if not self.existsEdge(src, dest):
                        self.edges[(src, dest)] = cost
                        self.vertices.add(src)
                        self.vertices.add(dest)

        # checks if exists a edge linking "src" in "dest"
        def existsEdge(self, src, dest):
                return (True if (src, dest) in self.edges else False)


        def showGraph(self):
                print('Showing the graph:\n')
                for edge in self.edges:
                        print('%d linked in %d with cost %d' % (edge[0],␣
 ↪edge[1], self.edges[edge]))

        # returns total cost of the path
```

```
        def getCostPath(self, path):
                total_cost = 0
                for i in range(self.V - 1):
                        total_cost += self.edges[(path[i], path[i+1])]
                # add cost of the last edge
                total_cost += self.edges[(path[self.V- 1], path[0])]
                return total_cost
```

[28]:
```
# Functions gives one random path in the graph
def get_one_random_path(g: Graph):
    lst =  list(range(g.V))
    rand_path =  []
    all_nodes = list(g.vertices)
    for i in range(g.V):
        random_node = all_nodes[random.randint(0,len(all_nodes)-1)]
        rand_path.append(random_node)
        all_nodes.remove(random_node)
    return rand_path
```

[29]:
```
# Generates a random complete graph,with random edge costs
def generate_random_complete_graph(n: int):
    graph = Graph(n)
    for i in range(graph.V):
        for j in range(graph.V):
            if i != j:
                weight = random.randint(1, 10)
                graph.addEdge(i, j, weight)
    return graph
```

We are going to be using Hill Climbing Method for finding the solution to this question.

[31]:
```
# Heuristic solution for TSP
def heuristic(g:Graph ):
    pth = get_one_random_path(g)
    l_pth = g.getCostPath(pth)
    neighbours = []
    for i in range(len(pth)):
        for j in range(i+1,len(pth)):
            nbr = pth.copy()
            nbr[i] = pth[j]
            nbr[j] = pth[i]
            neighbours.append(nbr)
    best_nbr,best_nbr_ln = closest_neighbour(g,neighbours)
    while best_nbr_ln < l_pth:
        pth = best_nbr
        l_pth = best_nbr_ln
        neighbours=[]

    for i in range(len(pth)):
```

```
        for j in range(i+1,len(pth)):
            nbr = pth.copy()
            nbr[i] = pth[j]
            nbr[j] = pth[i]
            neighbours.append(nbr)
        best_nbr,best_nbr_ln = closest_neighbour(g,neighbours)
    return pth + [pth[0]], l_pth
```

Hill climbing works by generating all neighbour solutions to the current one and then choosing the optimal one.

So for that we need to generate all the neigbouring solutions. Also Note that a neighbour has to be a correct solution too.

We can do this by taking the solution and swapping any two cities in it. We need to do this in a nested loop, for all cities.

Here is the code in the cell below:-

[30]:
```
# Gets the closest neighbour
def closest_neighbour(g:Graph, neighbours):
    best_ln = g.getCostPath(neighbours[0])
    best_nbr = neighbours[0]
    for nbr in neighbours:
        l = g.getCostPath(nbr)
        if( l < best_ln):
            best_nbr = nbr
            best_ln = l
    return best_nbr, best_ln
```

[32]:
```
# Bruteforce TSP
def brute_force_tsp(g: Graph):
    path = min(itertools.permutations(g.vertices), key=lambda path: g.
 ↪getCostPath(path))
    return (list( path  )+  [list( path )[0]] ,g.getCostPath(path))
```

[33]:
```
def main():
    # Generates a complete graph with 5 nodes and random cost between edges
    g = generate_random_complete_graph(5)
    g.showGraph()

    print("The best cost(bruteforce) is: ", brute_force_tsp(g))
    print("The best cost found with heuristic is: ", heuristic(g))
```

[35]:
```
main()
```

```
Showing the graph:

0 linked in 1 with cost 6
0 linked in 2 with cost 7
0 linked in 3 with cost 9
0 linked in 4 with cost 6
1 linked in 0 with cost 6
```

```
1 linked in 2 with cost 2
1 linked in 3 with cost 9
1 linked in 4 with cost 4
2 linked in 0 with cost 3
2 linked in 1 with cost 1
2 linked in 3 with cost 6
2 linked in 4 with cost 5
3 linked in 0 with cost 10
3 linked in 1 with cost 5
3 linked in 2 with cost 9
3 linked in 4 with cost 7
4 linked in 0 with cost 5
4 linked in 1 with cost 6
4 linked in 2 with cost 4
4 linked in 3 with cost 5
The best cost(bruteforce) is:  ([0, 4, 3, 1, 2, 0], 21)
The best cost found with heuristic is:  ([1, 2, 0, 4, 3, 1], 21)
```

[47]:

```
[NbConvertApp] Converting notebook Untitled0.ipynb to PDF
[NbConvertApp] Writing 35116 bytes to ./notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: [u'xelatex', u'./notebook.tex',
'-quiet']
[NbConvertApp] Running bibtex 1 time: [u'bibtex', u'./notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 37187 bytes to Untitled0.pdf
```

[ ]: