|  |
| --- |
|  |
| **DEPARTMENT OF COMPUTER ENGINEERING** |
| **CSL604 Artificial Intelligence Lab** |
| **Sixth Semester, 2021-2022 (Even Semester)** |

|  |  |  |
| --- | --- | --- |
| **Name of Student** | **:** |  |
| **Roll No.** | **:** |  |
| **Class** | **:** | TE-CMPN |
| **Day** | **:** |  |
| **Session** | **:** |  |
| **Venue** | **:** | Lab No. 307 |
| **Experiment No.** | **:** |  |
| **Title of Experiment** | **:** | Study and write programs on informed search methods. |
| **Date of Conduction** | **:** |  |
| **Date of Submission** | **:** |  |

|  |  |  |
| --- | --- | --- |
| **Particulars** | **Max. Marks** | **Marks Obtained** |
| Preparedness and Efforts (PE) | 3 |  |
| Knowledge of Tools (KT) | 3 |  |
| Debugging and Results (DR) | 3 |  |
| Documentation (DN) | 3 |  |
| Punctuality and Lab Ethics (PL) | 3 |  |
| **Total** | **15** |  |
| **Grades:** Meet Expectations (3 Marks), Moderate Expectations (2 Marks), Below Expectations (1 Mark) | | |

**Checked and verified by,**

|  |  |  |
| --- | --- | --- |
| **Name of Faculty** | **:** | Asst. Prof. Savyasaachi Pandit |
| **Signature** | **:** |  |
| **Date** | **:** |  |

**Experiment No 4**

**Aim:** To study and write Python programs on informed search methods.

**Theory:**

Informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node. The algorithm is more useful for large search space. It uses the idea of heuristic, so it is also called as Heuristic search.

**Heuristics Function:**

Heuristic is a function which is used in informed search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by , and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Here is heuristic cost, and is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

In informed search, there are two main algorithms which are given below:

* Best-first Search (Greedy search)
* A\* Search

**Best-first Search (Greedy Search):**

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

where, is estimated cost from node to the goal.

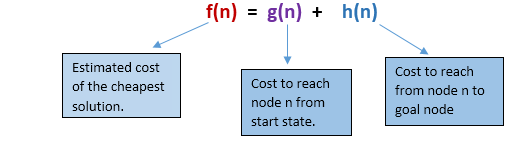
**Best-first Search Algorithm:**

1. Place the starting node into the OPEN list.
2. If the OPEN list is empty, stop and return failure.
3. Remove the node n from the OPEN list which has the lowest value of , and place it in the CLOSED list.
4. Expand the node , and generate the successors of node .
5. Check each successor of node , and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
6. For each successor node, algorithm checks for evaluation function , and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both lists, then add it to the OPEN list.
7. Return to Step 2.

**A\* Search:**

A\* search is the most commonly known form of best-first search. It uses heuristic function , and cost to reach the node from the start state . It has combined features of UCS and greedy best-first search, by which it solves the problem efficiently. A\* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A\* algorithm is similar to UCS except that it uses instead of .

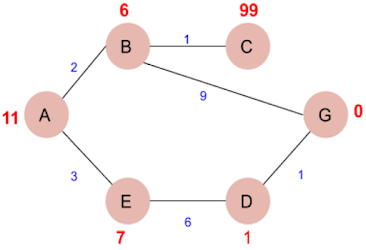
In A\* search algorithm, we use search heuristic as well as the cost to reach the node. Hence, we can combine both costs as following, and this sum is called as a fitness number.



**A\* Search Algorithm:**

1. Place the starting node in the OPEN list.
2. Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
3. Select the node from the OPEN list which has the smallest value of evaluation function , if node is goal node, then return success and stop.
4. Expand node and generate all of its successors, and put into the closed list. For each successor , check whether is already in the OPEN or CLOSED list, if not then compute evaluation function for and place into Open list.
5. Else if node is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest value.
6. Return to Step 2.

**A\* Search Example:**



Numbers written on edges represent the distance between nodes. Numbers written on nodes represent the heuristic value.

We have to the find the cost-effective path from to . That is is the source node and is the goal node.

Now from , we can go to point or , so we compute for each of them,

Since the cost for is less, we move forward with this path and compute the for the children nodes of .

Now from , we can go to point or , so we compute for each of them,

Here the path has the least cost but it is still more than the cost of , thus we explore this path further.

Now from , we can go to point the , so we compute ,

Comparing the cost of with all the paths we got so far and as this cost is least of all we move forward with this path.

Now compute the for the children of .

Now comparing all the paths that lead us to the goal, we conclude that is the most cost-effective path to get from to .

**A\* Search Algorithm Implementation in Python:**

def aStarAlgo(start\_node, stop\_node):

    open\_set = set(start\_node)

    closed\_set = set()

    # store distance from starting node

    g = {}

    # parents contains an adjacency map of all nodes

    parents = {}

    # distance of starting node from itself is zero

    g[start\_node] = 0

    # start\_node is root node i.e it has no parent nodes

    # so start\_node is set to its own parent node

    parents[start\_node] = start\_node

    while len(open\_set) > 0:

        n = None

        # node with lowest f() is found

        for v in open\_set:

            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

                n = v

        if n == stop\_node or Graph\_nodes[n] == None:

            pass

        else:

            for (m, weight) in get\_neighbors(n):

                # nodes 'm' not in first and last set are added to first

                # n is set its parent

                if m not in open\_set and m not in closed\_set:

                    open\_set.add(m)

                    parents[m] = n

                    g[m] = g[n] + weight

                # for each node m,compare its distance from start i.e g(m) to the

                # from start through n node

                else:

                    if g[m] > g[n] + weight:

                        # update g(m)

                        g[m] = g[n] + weight

                        # change parent of m to n

                        parents[m] = n

                        # if m in closed set,remove and add to open

                        if m in closed\_set:

                            closed\_set.remove(m)

                            open\_set.add(m)

        if n == None:

            print('Path does not exist!')

            return None

        # if the current node is the stop\_node

        # then we begin reconstructin the path from it to the start\_node

        if n == stop\_node:

            path = []

            while parents[n] != n:

                path.append(n)

                n = parents[n]

            path.append(start\_node)

            path.reverse()

            print('Path found: {}'.format(path))

            return path

        # remove n from the open\_list, and add it to closed\_list

        # because all of his neighbors were inspected

        open\_set.remove(n)

        closed\_set.add(n)

    print('Path does not exist!')

    return None

# define fuction to return neighbor and its distance

# from the passed node

def get\_neighbors(v):

    if v in Graph\_nodes:

        return Graph\_nodes[v]

    else:

        return None

# A star Example-1

# for simplicity we'll consider heuristic distances given

# and this function returns heuristic distance for all nodes

def heuristic(n):

    H\_dist = {

        'A': 11,

        'B': 6,

        'C': 5,

        'D': 7,

        'E': 3,

        'F': 6,

        'G': 5,

        'H': 3,

        'I': 1,

        'J': 0

    }

    return H\_dist[n]

# graph

Graph\_nodes = {

    'A': [('B', 6), ('F', 3)],

    'B': [('A', 6), ('C', 3), ('D', 2)],

    'C': [('B', 3), ('D', 1), ('E', 5)],

    'D': [('B', 2), ('C', 1), ('E', 8)],

    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],

    'F': [('A', 3), ('G', 1), ('H', 7)],

    'G': [('F', 1), ('I', 3)],

    'H': [('F', 7), ('I', 2)],

    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],

}

aStarAlgo('A', 'J')

# for simplicity we'll consider heuristic distances given

# and this function returns heuristic distance for all nodes

def heuristic(n):

    H\_dist = {

        'A': 11,

        'B': 6,

        'C': 99,

        'D': 1,

        'E': 7,

        'G': 0,

    }

    return H\_dist[n]

# graph

Graph\_nodes = {

    'A': [('B', 2), ('E', 3)],

    'B': [('A', 2), ('C', 1), ('G', 9)],

    'C': [('B', 1)],

    'D': [('E', 6), ('G', 1)],

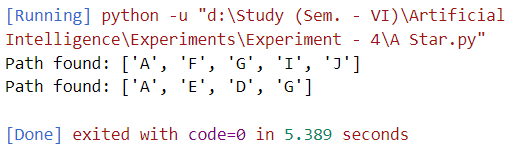
    'E': [('A', 3), ('D', 6)],

    'G': [('B', 9), ('D', 1)]

}

aStarAlgo('A', 'G')

**Output:**



**Hill-Climbing Search:**

In hill climbing, the basic idea is to always head towards a state which is better than the current one.

So, if we are in town A and we can get to town B and town C (and our target is town D) then we should make a move if town B or C appear nearer to town D than town A does.

**Hill-Climbing Search Algorithm:**

1. Evaluate the initial state.
   1. If it is also goal state then return it, otherwise continue with the initial state as the current state.
2. Loop until the solution is found or until there are no new operators to be applied in the current state.
   1. Select an operator that has not yet been applied to the current state and apply it to produce new state.
   2. Evaluate the new state.
      1. If it is a goal state then return it and quit.
      2. If it is not a goal state but it is better than the current state, then make it as current state.
      3. If it is not better than the current state, then continue in loop.

**Travelling Salesman Problem:**

First, let’s code an instantiation of the travelling salesman problem. If we think about it, such an instantiation should be a list of cities, where each one has information about the distances from there to the other cities. Of course, the distance from each city to itself is zero, and the distance from city A to city B is the same as the distance from city B to city A. That gives us a list, containing lists of size (where in this case equals to 4).

tsp = [

[0, 400, 500, 300],

[400, 0, 300, 500],

[500, 300, 0, 400],

[300, 500, 400, 0]

]

**Travelling Salesman Problem Implementation in Python:**

import random

def randomSolution(tsp):

    cities = list(range(len(tsp)))

    solution = []

    for i in range(len(tsp)):

        randomCity = cities[random.randint(0, len(cities) - 1)]

        solution.append(randomCity)

        cities.remove(randomCity)

    return solution

def routeLength(tsp, solution):

    routeLength = 0

    for i in range(len(solution)):

        routeLength += tsp[solution[i - 1]][solution[i]]

    return routeLength

def getNeighbours(solution):

    neighbours = []

    for i in range(len(solution)):

        for j in range(i + 1, len(solution)):

            neighbour = solution.copy()

            neighbour[i] = solution[j]

            neighbour[j] = solution[i]

            neighbours.append(neighbour)

    return neighbours

def getBestNeighbour(tsp, neighbours):

    bestRouteLength = routeLength(tsp, neighbours[0])

    bestNeighbour = neighbours[0]

    for neighbour in neighbours:

        currentRouteLength = routeLength(tsp, neighbour)

        if currentRouteLength < bestRouteLength:

            bestRouteLength = currentRouteLength

            bestNeighbour = neighbour

    return bestNeighbour, bestRouteLength

def hillClimbing(tsp):

    currentSolution = randomSolution(tsp)

    currentRouteLength = routeLength(tsp, currentSolution)

    neighbours = getNeighbours(currentSolution)

    bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

    while bestNeighbourRouteLength < currentRouteLength:

        currentSolution = bestNeighbour

        currentRouteLength = bestNeighbourRouteLength

        neighbours = getNeighbours(currentSolution)

        bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

    return currentSolution, currentRouteLength

def main():

    tsp = [

        [0, 400, 500, 300],

        [400, 0, 300, 500],

        [500, 300, 0, 400],

        [300, 500, 400, 0]

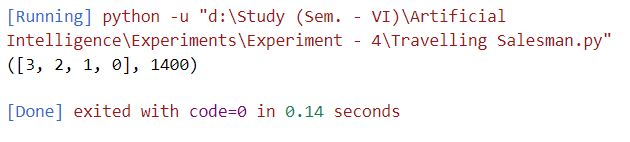
    ]

    print(hillClimbing(tsp))

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Output:**



**Conclusion:**

* Informed search refers to search algorithms which help in navigating large databases with certain available information about the end goal in search and most widely used in large databases where uninformed search algorithms can’t accurately curate precise results.
* Best-first search can switch between BFS and DFS by gaining the advantages of both the algorithms. It is more efficient than BFS and DFS algorithms.
* Best-first search can behave as an unguided depth-first search in the worst-case scenario. It can get stuck in a loop as DFS. This algorithm is not optimal.
* A\* search algorithm is the best algorithm than other search algorithms. It is optimal and complete. This algorithm can solve very complex problems.
* A\* search algorithm does not always produce the shortest path as it mostly based on heuristics and approximation. It has some complexity issues. The main drawback of A\* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.