|  |  |  |
| --- | --- | --- |
| **Name :**  Dyotak Kachare | **Class/Roll No. :**  D11AD/26 | **Grade:** |

**Title of Experiment :**

Implement any one of the Informed search techniques E.g. A-Star algorithm for 8 puzzle problem.

**Objective of Experiment :**

To design a Problem-Solving Agent

**Outcome of Experiment :**

Implement various search techniques for a Problem-Solving Agent.

**Problem Statement :**

Implement any one of the Informed search techniques E.g. A-Star algorithm for 8 puzzle problem.**Description / Theory :**

Informed (Heuristic) Search Strategies

An informed search strategy—one that uses domain-specific hints about the location of goals—can find solutions more efficiently than an uninformed strategy. The hints come in the form of a heuristic function denoted by h(n).

*h(n) = estimated cost of the cheapest path from the state at node n to a goal state.*

* Heuristic is estimate of cost from node n to goal
* Defined only using the state of node n
* h(n) = 0 if n is the goal node

A\* search

The most common informed search algorithm is A\* search (pronounced “A-star search”), a best-first search that uses the evaluation function.

*f(n) = g(n) + h(n)*

where g(n) is the path cost from the initial state to node and is the estimated cost of the shortest path from to a goal state, so we have

*f(n) = estimated cost of best path from start node to goal*

A\* search is complete. Whether A\* is cost-optimal depends on certain properties of the heuristic. A key property is admissibility: an admissible heuristic is one that never overestimates the cost to reach a goal. (An admissible heuristic is therefore optimistic.) With an admissible heuristic, A\* is cost-optimal, which we can show with a proof by contradiction. Suppose the optimal path has cost but the algorithm returns a path with cost Then there must be some node which is on the optimal path and is unexpanded (because if all the nodes on the optimal path had been expanded, then we would have returned that optimal solution). So then, using the notation to mean the cost of the optimal path from the start to and to mean the cost of the optimal path from to the nearest goal, we have:

*f (n) > C ∗ (otherwise n would have been expanded)*

*f (n) = g (n) + h (n) (by definition)*

*f (n) = g ∗ (n) + h (n) (because n is on an optimal path)*

*f (n) ≤ g ∗ (n) + h ∗ (n) (because of admissibility, h (n) ≤ h ∗ (n))*

*f (n) ≤ C ∗ (by definition, C ∗ = g ∗ (n) + h ∗ (n))*

Limitations   
Although being the best path finding algorithm around, A\* Search Algorithm doesn’t produce the shortest path always, as it relies heavily on heuristics / approximations to calculate – h

Applications   
This is the most interesting part of A\* Search Algorithm. They are used in games!

Tower defense is a type of strategy video game where the goal is to defend a player’s territories or possessions by obstructing enemy attackers, usually achieved by placing defensive structures on or along their path of attack.  A\* Search Algorithm is often used to find the shortest path from one point to another point. You can use this for each enemy to find a path to the goal. One example of this is the very popular game- Warcraft III

What if the search space is not a grid and is a graph ? The same rules applies there also. The example of grid is taken for the simplicity of understanding. So we can find the shortest path between the source node and the target node in a graph using this A\* Search Algorithm, just like we did for a 2D Grid.

Time Complexity

Considering a graph, it may take us to travel all the edge to reach the destination cell from the source cell [For example, consider a graph where source and destination nodes are connected by a series of edges, like – 0(source) –>1 –> 2 –> 3 (target)] So the worst case time complexity is O(E), where E is the number of edges in the graph.

Auxiliary Space

In the worse case we can have all the edges inside the open list, so required auxiliary space in worst case is O(V), where V is the total number of vertices.

**Algorithm/ Pseudo Code/ Flowchart :**

A\* Search Algorithm

1. function A\*Search(start\_state, goal\_state):
2. open\_set = priority queue with (heuristic(start\_state), start\_state)
3. came\_from = empty dictionary
4. g\_score = dictionary with start\_state as key and 0 as value
5. while open\_set is not empty:
6. current = state with lowest f\_score from open\_set
7. remove current from open\_set
8. if current == goal\_state:
9. return construct\_path(came\_from, current)
10. for each possible move from current:
11. new\_state = perform move on current
12. new\_state\_g\_score = g\_score[current] + cost of move from current to new\_state
13. if new\_state\_g\_score < g\_score[new\_state] or new\_state not in g\_score:
14. g\_score[new\_state] = new\_state\_g\_score
15. f\_score = new\_state\_g\_score + heuristic(new\_state, goal\_state)
16. add (f\_score, new\_state) to open\_set
17. came\_from[new\_state] = current
18. return failure
19. function construct\_path(came\_from, current):
20. path = [current]
21. while current in came\_from:
22. current = came\_from[current]
23. prepend current to path
24. return path

# Call A\*Search function with start\_state and goal\_state

1. solution\_path = A\*Search(start\_state, goal\_state)
2. if solution\_path is not failure:
3. for step, state in enumerate(solution\_path):
4. print "Step", step, ":"
5. print state

else:

1. print "No solution found."

**Program :**

Code

import queue

import heapq

start\_state = [

[1, 8, 4],

[5, 0, 3],

[6, 2, 7]

]

goal\_state = [

[8, 3, 4],

[1, 0, 5],

[7, 2, 6]

]

goal\_positions = {}

for i in range(3):

for j in range(3):

goal\_positions[goal\_state[i][j]] = (i, j)

def heuristic(state):

total\_distance = 0

for i in range(3):

for j in range(3):

if state[i][j] != 0:

goal\_row, goal\_col = goal\_positions[state[i][j]]

total\_distance += abs(i - goal\_row) + abs(j - goal\_col)

return total\_distance

def solve(start\_state, goal\_state):

open\_set = []

heapq.heappush(open\_set, (heuristic(start\_state), start\_state))

came\_from = {tuple(map(tuple, start\_state)): None}

g\_score = {tuple(map(tuple, start\_state)): 0}

while open\_set:

\_, current = heapq.heappop(open\_set)

if current == goal\_state:

path = []

while current:

path.append(current)

current = came\_from.get(tuple(map(tuple, current)), None)

path.reverse()

return path

for move in [(0, 1), (0, -1), (1, 0), (-1, 0)]:

new\_state = [list(row) for row in current]

zero\_row, zero\_col = [(i, row.index(0)) for i, row in enumerate(current) if 0 in row][0]

new\_row, new\_col = zero\_row + move[0], zero\_col + move[1]

if 0 <= new\_row < 3 and 0 <= new\_col < 3:

new\_state[zero\_row][zero\_col], new\_state[new\_row][new\_col] = new\_state[new\_row][new\_col], new\_state[zero\_row][zero\_col]

new\_state\_tuple = tuple(map(tuple, new\_state))

tentative\_g\_score = g\_score[tuple(map(tuple, current))] + 1

if new\_state\_tuple not in g\_score or tentative\_g\_score < g\_score[new\_state\_tuple]:

g\_score[new\_state\_tuple] = tentative\_g\_score

f\_score = tentative\_g\_score + heuristic(new\_state)

heapq.heappush(open\_set, (f\_score, new\_state))

came\_from[new\_state\_tuple] = current

return None

solution\_path = solve(start\_state, goal\_state)

if solution\_path:

for step, state in enumerate(solution\_path):

print(f"Step {step}:\n")

for row in state:

print(row)

print("\n")

else:

print("No solution found.")

Output

Step 0:

[1, 8, 4]

[5, 0, 3]

[6, 2, 7]

Step 1:

[1, 8, 4]

[5, 2, 3]

[6, 0, 7]

Step 2:

[1, 8, 4]

[5, 2, 3]

[0, 6, 7]

Step 3:

[1, 8, 4]

[0, 2, 3]

[5, 6, 7]

Step 4:

[1, 8, 4]

[2, 0, 3]

[5, 6, 7]

Step 5:

[1, 8, 4]

[2, 6, 3]

[5, 0, 7]

Step 6:

[1, 8, 4]

[2, 6, 3]

[5, 7, 0]

Step 7:

[1, 8, 4]

[2, 6, 0]

[5, 7, 3]

Step 8:

[1, 8, 4]

[2, 0, 6]

[5, 7, 3]

Step 9:

[1, 8, 4]

[2, 7, 6]

[5, 0, 3]

Step 10:

[1, 8, 4]

[2, 7, 6]

[0, 5, 3]

Step 11:

[1, 8, 4]

[0, 7, 6]

[2, 5, 3]

Step 12:

[1, 8, 4]

[7, 0, 6]

[2, 5, 3]

Step 13:

[1, 8, 4]

[7, 5, 6]

[2, 0, 3]

Step 14:

[1, 8, 4]

[7, 5, 6]

[2, 3, 0]

Step 15:

[1, 8, 4]

[7, 5, 0]

[2, 3, 6]

Step 16:

[1, 8, 4]

[7, 0, 5]

[2, 3, 6]

Step 17:

[1, 8, 4]

[7, 3, 5]

[2, 0, 6]

Step 18:

[1, 8, 4]

[7, 3, 5]

[0, 2, 6]

Step 19:

[1, 8, 4]

[0, 3, 5]

[7, 2, 6]

Step 20:

[0, 8, 4]

[1, 3, 5]

[7, 2, 6]

Step 21:

[8, 0, 4]

[1, 3, 5]

[7, 2, 6]

Step 22:

[8, 3, 4]

[1, 0, 5]

[7, 2, 6]

**Result and Discussion :**

Thus we have successfully implemented A\* Search in Python. We were successfully able to implement an informed search technique to design a Problem-Solving Agent.