| | |
|---|---|
| **Name:** | Sunit Sunil Khaire |
| **Roll No:** | 19 |
| **Class/Sem:** | TE/V |
| **Experiment No.:** | 5 |
| **Title:** | Implementation of A* search for problem solving. |
| **Date of Performance:** | |
| **Date of Submission:** | |
| **Marks:** | |
| **Sign of Faculty:** | |

**Aim:** Implementation of A* search for problem solving.

**Objective:** To study the informed searching techniques and its implementation for problem solving.

**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 informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called 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 h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

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.

1. f(n)= g(n).

Were, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

A* search algorithm:

**Step1:** Place the starting node in the OPEN list.
**Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
**Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
**Step 4:** Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
**Step 5:** Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
**Step 6:** Return to **Step 2**.

Advantages:

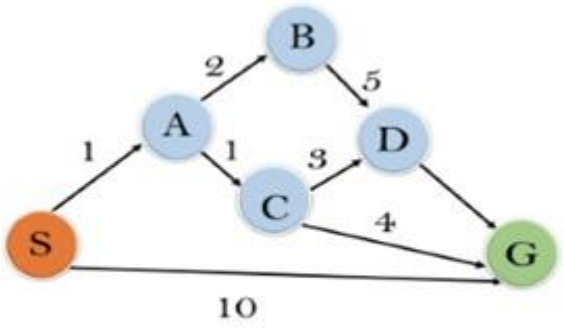
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm 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.

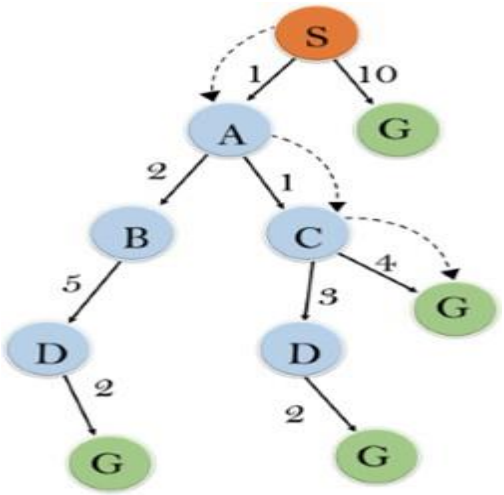
Example:

In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.




| State | h(n) |
|---|---|
| S | 5 |
| A | 3 |
| B | 4 |
| C | 2 |
| D | 6 |
| G | 0 |

**Solution:**



**Iteration1:** {(S--> A, 4), (S-->G, 10)}
**Iteration2:** {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}
**Iteration3:** {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}
**Iteration 4** will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

**Points to remember:**

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)<="" li="">

**Complete:** A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

**Optimal:** A* search algorithm is optimal if it follows below two conditions:

- **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

**Time Complexity:** The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

**Space Complexity:** The space complexity of A* search algorithm is **O(b^d)**

**Code :**

```
import heapq


# A* algorithm
def a_star(graph, start, goal, h):
    # Priority queue to store nodes to explore (f-score, node)
    open_list = []
    heapq.heappush(open_list, (0, start))


    # Cost from start to current node
```

```
g_score = {node: float('inf') for node in graph}
g_score[start] = 0

# To store the path
came_from = {}

# While there are nodes to explore
while open_list:
    # Get the node with the lowest f-score
    current_f_score, current_node = heapq.heappop(open_list)

    # If the goal is reached, reconstruct the path
    if current_node == goal:
        return reconstruct_path(came_from, current_node)

    # Explore neighbors
    for neighbor, cost in graph[current_node].items():
        tentative_g_score = g_score[current_node] + cost

        if tentative_g_score < g_score[neighbor]:
            came_from[neighbor] = current_node
            g_score[neighbor] = tentative_g_score
            f_score = tentative_g_score + h[neighbor]
            heapq.heappush(open_list, (f_score, neighbor))

return None  # No path found
```

```
# Function to reconstruct the path
def reconstruct_path(came_from, current):
    total_path = [current]
    while current in came_from:
        current = came_from[current]
        total_path.append(current)
    total_path.reverse()
    return total_path


# Example usage
if __name__ == "__main__":
    # Example graph as adjacency list with edge weights
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'A': 1, 'D': 2, 'E': 5},
        'C': {'A': 4, 'F': 3},
        'D': {'B': 2},
        'E': {'B': 5, 'F': 1},
        'F': {'C': 3, 'E': 1}
    }

    # Heuristic values (example based on straight-line distance)
    h = {
        'A': 7,
        'B': 6,
        'C': 2,
        'D': 5,
```

```
    'E': 3,
    'F': 0  # Goal node has heuristic 0
}

start_node = 'A'
goal_node = 'F'
path = a_star(graph, start_node, goal_node, h)

if path:
    print("Path found:", path)
else:
    print("No path found")
```

**Output:**

```
Path found: ['A', 'C', 'F']
```

**Conclusion:**

The A* algorithm implementation effectively demonstrates its ability to find the shortest path between two nodes in a weighted graph by using a combination of actual cost (`g`) and heuristic estimates (`h`). It intelligently explores nodes with the lowest estimated cost, ensuring both accuracy and efficiency. The use of a priority queue helps in reducing unnecessary exploration, making A* optimal for pathfinding in scenarios like GPS navigation, game development, and robotics. The experiment confirms that A* is highly effective in navigating complex graphs by balancing between exploration and exploitation through heuristic guidance.