2.3 INFORMED SEARCH

Informed search A* Algorithm

•A* search is an informed search algorithm used in artificial intelligence to find the most efficient path between a starting point and a goal.

•It utilizes a heuristic function to estimate the cost of reaching the goal from any given node, guiding the search towards promising paths while considering the actual cost already incurred.

How it Works:

1. Initialization:

The starting node is added to the open list with its f(n) value calculated.

2. Iteration:

•The node with the lowest f(n) value is selected from the open list and moved to the closed list.

•If the selected node is the goal, the algorithm terminates and returns the path.

•Otherwise, the node's successors are generated, and their f(n) values are calculated.

•If a successor is already in the open list with a higher f(n) value, its value is updated.

•If a successor is not in the open list, it's added.

3. Termination:

The algorithm stops when the goal node is reached or the open list is empty (indicating no solution).

A* Example Diagram

Let's consider the following graph:

```
   A
 /   |   \
B   C   D
 / \        \
E F        G
```

Edge costs:

A→B = 1, A→C = 3, A→D = 2

B→E = 4, B→F = 5   D→G = 2

Heuristic h(n) to Goal:

A=5, B=4, C=6, D=2, E=7, F=6, G=0 Path A → D → G has:

- g(n) = 2 (A→D) + 2 (D→G) = 4
- h(n) = 0 (G)
- f(n) = 4 + 0 = 4 (lowest path)

Python Program for A*

```
# Graph represented as adjacency list with edge costs
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'D': 2, 'E': 5},
    'C': {'F': 3},
    'D': {},
    'E': {'F': 1},
    'F': {}
}

# Heuristic values (estimated cost to goal 'F')
heuristic = {
    'A': 7,
    'B': 6,
    'C': 2,
    'D': 1,
    'E': 1,
    'F': 0
}

def a_star(start, goal):
    open_set = set([start])          # Nodes to explore
    came_from = {}                   # To reconstruct the path
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0               # Cost from start to node
    f_score = {node: float('inf') for node in graph}
    f_score[start] = heuristic[start] # Estimated total cost

    while open_set:
        # Get node with lowest f_score
        current = min(open_set, key=lambda x: f_score[x])

        if current == goal:
            # Reconstruct path
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            path.reverse()
            return path

        open_set.remove(current)
        for neighbor, cost in graph[current].items():
            tentative_g_score = g_score[current] + cost
            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score[neighbor] + heuristic[neighbor]
                open_set.add(neighbor)

    return None  # No path found


# Example usage
path = a_star('A', 'F')
print("Path found:", path)
```

Output:

```
Path found: [‘A’, ‘C’, ‘F’]
```

Key Characteristics:

•Completeness: A* is complete, meaning it will find a solution if one exists.

•Optimality: A* is optimal if the heuristic function is admissible (never overestimates the actual cost to the goal). This guarantees that A* will find the shortest path. Benefits:

•Efficiency:

A* is generally very efficient in finding optimal paths due to the guidance provided by the heuristic function.

•Versatility:

It can be applied to various pathfinding and search problems in different domains.

Example:

Imagine navigating a maze. A* would use a heuristic like the straight-line distance to the exit to guide its search, prioritizing paths that appear closer to the exit while also considering the actual distance traveled so far.

INFORMED SEARCH IN BEST FIRST

•Best-first search is a graph traversal or pathfinding algorithm that explores a graph by prioritizing nodes based on an evaluation function.

•It selects the most promising node to expand next, aiming to efficiently find a solution, especially in large search spaces.

How it Works:

1.Initialization: Start with an open list containing the initial node and an empty closed list.

2.Node Selection: Repeatedly select the node with the best (lowest or highest, depending on the heuristic) evaluation function value from the open list.

3.Expansion: Expand the selected node by generating its successor nodes.

4.Heuristic Evaluation: Evaluate the successor nodes using the heuristic function.

5.Open List Management: Add the generated nodes to the open list, potentially updating their evaluation values if better paths are found.

6.Closed List Update: Move the expanded node from the open list to the closed list.

7.Goal Check: If the selected node is the goal node, terminate the search. Example:

Imagine a map with cities and roads, and you want to find the shortest route between two cities. You could use best-first search, where the heuristic estimates the distance to the goal city (perhaps using the straight-line distance). The algorithm would prioritize exploring cities that are closer to the goal, potentially finding the shortest path more efficiently than a simple breadth-first or depth-first search.

Benefits:

•Efficiency:

By using a heuristic, best-first search can explore the search space more efficiently than uninformed search algorithms.

•Flexibility:

It can be adapted to different problems by using different evaluation functions and heuristics.

•Good for optimization:

It's particularly useful for finding optimal paths or solutions in complex problems.

Limitations:

•Heuristic Dependence:

The performance of best-first search heavily relies on the quality of the heuristic function. A poorly designed heuristic can lead to suboptimal or inefficient search.

•Can get stuck in local optima:

Like other informed search algorithms, it may not always find the globally optimal solution if the heuristic is misleading.

Tree Based Graph Diagram:

```
          [Start (h=3)]
             /      \
         (h=5)       (h=2)
           /            \
       [A (h=5)]       [B (h=2)]
          |               |
       (h=4)            (h=1)
          |               |
      [C (h=4)]         [D (h=1)]
          |               |
       (h=0)            (h=0)
          |               |
      [Goal (h=0)]      [Goal (h=0)]
```

Example Program Using the Tree Diagram:

```
# Graph represented as adjacency list
graph = {
    'Start': ['A', 'B'],
    'A': ['C'],
    'B': ['D'],
    'C': ['Goal'],
    'D': ['Goal'],
    'Goal': []
}

# Heuristic values (h(n)) – lower is closer to goal
heuristic = {
    'Start': 3,
    'A': 5,
    'B': 2,
    'C': 4,
    'D': 1,
    'Goal': 0
}

def best_first_search(start, goal):
    open_list = [start]    # Nodes to explore
    visited = set()        # Visited nodes
    path = []

    while open_list:
        # Pick node with lowest heuristic value
        current = min(open_list, key=lambda x: heuristic[x])
        open_list.remove(current)
        path.append(current)
        visited.add(current)

        if current == goal:
            return path

        # Add unvisited neighbors to open_list
        for neighbor in graph[current]:
            if neighbor not in visited and neighbor not in open_list:
                open_list.append(neighbor)

    return None  # Goal not found
# Run Best-First Search
result = best_first_search('Start', 'Goal')
print("Path found:", result)
```

Output:

```
Path found: [‘Start’, ‘B’, ‘D’, ‘Goal’]
```

Key Concepts:

Heuristic Function: A function that estimates the cost from a given node to the goal node. This guides the search towards promising paths.

Evaluation Function: A function that combines the heuristic with other factors (like path cost from the start) to determine the "best" node to explore next.

Priority Queue: A data structure that stores nodes and prioritizes them based on their evaluation function value, allowing the algorithm to easily pick the most promising one.

Open List: A list of nodes that have been generated but not yet explored.

Closed List: A list of nodes that have already been explored.

GREEDY SEARCH:

•Greedy search, or more broadly, a greedy algorithm, is an algorithmic paradigm that makes the locally optimal choice at each step with the hope that this choice will lead to a globally optimal solution.

•It is a heuristic approach, meaning it prioritizes immediate gains without necessarily considering the long-term consequences of its decisions.

How it works:

•Start with an initial state: The problem begins at a defined starting point.

•Evaluate choices: From the current state, all possible choices or actions are considered.

•Make the "greedy" choice: The option that maximizes or minimizes the objective function at that moment is selected.

•Move to a new state: The problem transitions to the state resulting from the chosen option.

•Repeat: Steps 2-4 are repeated until a goal state is reached or no further progress is possible.

Conditions for success:

For a greedy algorithm to produce an optimal solution, the problem typically needs to exhibit two properties:

•Greedy Choice Property:

The problem must be structured such that a globally optimal solution can be achieved by making locally optimal choices at each step.

•Optimal Substructure:

The optimal solution to the overall problem must contain optimal solutions to its sub problems.

Key characteristics of greedy search:

- Local Optimization:

At each stage, the algorithm selects the option that appears to be the best at that specific moment, based on a defined objective function.

- No Backtracking:

Once a choice is made, the greedy algorithm typically does not revisit or reverse that decision, even if it later proves to be suboptimal for the overall solution.

- Simplicity and Efficiency:

Greedy algorithms are often straightforward to understand and implement, and can be computationally efficient for certain problems.

- Not Always Optimal:

While effective for some problems (e.g., Dijkstra's algorithm for shortest path, Huffman coding for data compression), greedy algorithms do not guarantee a globally optimal solution for all problems. This is because a series of locally optimal choices may not always lead to the best overall outcome.

Greedy Algorithm with a simple example using Problem: Activity Selection Problem

Goal: Select the maximum number of non-overlapping activities that a person can attend, where each activity has a start and end time.

Example Diagram:

Activity	Start Time	End Time
A1	1	4
A2	3	5
A3	0	6
A4	5	7
A5	8	9
A6	5	9

Greedy Choice:

*Sort activities by end time.

*Always pick the next activity that starts after the last selected activity ends.

Selected Activities (after sorting by end time):

*A1 (1–4)

*A4 (5–7)

*A5 (8–9)

Python Program:

```
# List of activities with (Activity, Start, End)
activities = [
    ('A1', 1, 4),
    ('A2', 3, 5),
    ('A3', 0, 6),
    ('A4', 5, 7),
    ('A5', 8, 9),
    ('A6', 5, 9)
]
# Step 1: Sort activities by their end time
activities.sort(key=lambda x: x[2])
# Step 2: Select activities
selected_activities = []
last_end_time = 0
for activity in activities:
    name, start, end = activity
    if start >= last_end_time:
        selected_activities.append(name)
    last_end_time = end
# Output selected activities
print("Selected Activities:", selected_activities)
```

Output:

```
Selected Activities: [‘A1’, ‘A4’, ‘A5’]
```

HEURISTIC FUNCTION:

•A heuristic function, in the context of artificial intelligence and computer science, is a mathematical function that estimates the cost or distance from a given state to a goal state in a search problem.

•It provides an informed "guess" or rule of thumb to guide search algorithms, enabling them to explore the most promising paths first and thus improve efficiency, especially in complex search spaces where exhaustive exploration is computationally expensive.

Key characteristics and concepts:

•Estimation:

A heuristic function does not provide the exact cost or distance, but rather an approximation.

•Guidance:

It helps search algorithms prioritize which paths or nodes to explore, directing the search towards the goal.

•Efficiency:

By guiding the search, it significantly reduces the number of states or nodes that need to be examined, leading to faster problem-solving.

•Admissibility:

An admissible heuristic never overestimates the actual cost to reach the goal. This property is crucial for algorithms like A* search to guarantee finding an optimal solution.

•Consistency (or Monotonicity):

A consistent heuristic satisfies the triangle inequality, meaning the estimated cost from a node to the goal is less than or equal to the actual cost to move to an adjacent node plus the estimated cost from that adjacent node to the goal. Consistency implies admissibility.

Examples:

•Pathfinding (e.g., GPS navigation):

The straight-line distance (Euclidean distance) between a current location and the destination can serve as a heuristic, estimating the travel time or distance.

•Puzzle Solving (e.g., 8-puzzle):

The number of misplaced tiles or the Manhattan distance (sum of horizontal and vertical distances of each tile from its goal position) are common heuristics.

•Antivirus Software:

Heuristic analysis in antivirus software uses predefined rules and patterns to identify suspicious code or behavior that might indicate malware, even if the specific signature is unknown.

What is a Heuristic Function?

A heuristic function `h(n)` estimates the cost from a node `n` to the goal. It helps guide the search toward the goal more efficiently.

Problem: Find shortest path from Start (A) to Goal (G)

We'll use a graph with costs and a heuristic table.

Example Graph:

Nodes: A, B, C, D, E, F, G

Edges with Cost:

*A → B (1)

*A → C (3)

*B → D (3)

*C → D (1)

* D → G (2)

*C → E (6)

*E → G (2)

Heuristic Table (h(n)):

Node	h(n)
A	7
B	6
C	5
D	3
E	4
F	∞
G	0

Best First Search using h(n):

Choose node with lowest heuristic value from the frontier.

Steps:

1.Start at A → h(A)=7

2.From A → B (h=6), C (h=5) → Choose C

3.From C → D (h=3), E (h=4) → Choose D

4.From D → G (h=0) → Goal reached ✅

Path: A → C → D → G

Python Program (Best First Search using h(n)):

```
# Graph represented as adjacency list
graph = {
   'Start': ['A', 'B'],
   'A': ['C'],
   'B': ['D'],
   'C': ['Goal'],
   'D': ['Goal'],
   'Goal': []
}
# Heuristic values h(n) for each node (lower is closer to goal)
heuristic = {
   'Start': 3,
   'A': 5,
   'B': 2,
   'C': 4,
   'D': 1,
   'Goal': 0
}
def best_first_search(start, goal):
   open_list = [start]  # Nodes to explore
   visited = set()      # Visited nodes
   path = []
   while open_list:
       # Choose the node with lowest heuristic value
       current = min(open_list, key=lambda x: heuristic[x])
       open_list.remove(current)
       path.append(current)
       visited.add(current)
       if current == goal:
           return path
       # Add unvisited neighbors to open list
       for neighbor in graph[current]:
           if neighbor not in visited and neighbor not in open_list:
               open_list.append(neighbor)
   return None  # Goal not found
# Run Best-First Search
result = best_first_search('Start', 'Goal')
print("Path found:", result)
```

Output:

```
Path found: [‘Start’, ‘B’, ‘D’, ‘Goal’]
```