# A* Search Algorithm

Introduction

A* Search finds the shortest path between two points by efficiently exploring possible routes. It uses a heuristic to estimate the distance to the goal and prioritizes paths that seem most promising.

Overview: A widely used pathfinding and graph traversal algorithm known for its efficiency and accuracy in finding the shortest path between two points.

Efficiency and Accuracy: Excels in finding the shortest path due to its informed search strategy.

Applications: Widely used in various applications, including:

- Robotics
- Games
- Maps
- Navigation systems
- Resource planning

---

Key Concepts – Heuristic Function

- Definition: A heuristic function, often denoted as ‘h’, estimates the cost of reaching the

goal from a given node. It provides an informed way to guess the remaining distance to the goal.

- Importance in Prioritization: The heuristic function plays a crucial role in prioritizing which nodes to explore during the search. By estimating the distance to the goal, it guides the algorithm towards more promising paths, potentially avoiding unnecessary exploration of less likely paths.
- Impact on Search Speed: A good heuristic function can significantly speed up the search process. By effectively guiding the exploration towards the goal, it helps reduce the number of nodes that need to be examined. This can be particularly important in large or complex search spaces where exhaustive exploration is computationally expensive.

---

Cost Function

- Definition: The cost function, often denoted as ‘g’, determines the cost of moving from one node to another. This cost can represent various factors, such as distance, time, or energy, depending on the specific problem.
- Combination with Heuristic: The cost function is combined with the heuristic function to calculate the total cost of a path. The total cost, often denoted as ‘f’, is typically calculated as:

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

where:

- ‘f(n)’ is the total cost of the path through node ‘n’
- ‘g(n)’ is the cost of reaching node ‘n’ from the start node
- ‘h(n)’ is the heuristic estimate of the cost from node ‘n’ to the goal node

---

Open and Closed Lists

- Open List:
  - The open list keeps track of nodes that have been discovered but not yet explored.
  - It acts as a frontier, holding nodes that are candidates for expansion.
- Closed List:
  - The closed list stores nodes that have already been explored.
  - This prevents the algorithm from revisiting nodes and getting stuck in cycles.

---

Algorithm Steps

1. Initialization: Begin with a start node and an open list containing only the start node. Set the start node’s g value (cost from start) to 0, its h value (heuristic estimate to goal) to its initial estimate, and its f value (total cost) to g + h. Create an empty closed list.

2. Node Selection:
    - If the open list is empty, no path exists; terminate.
    - Otherwise, select the node from the open list with the lowest `f` value. Call this node BESTNODE.
3. Expansion and Goal Check:
    - Remove BESTNODE from the open list and add it to the closed list.
    - If BESTNODE is the goal node, the path has been found; terminate and reconstruct the path by tracing back from the goal node to the start node.
    - Otherwise, generate the successors (neighbors) of BESTNODE.
4. Successor Evaluation: For each SUCCESSOR of BESTNODE:
    - Set SUCCESSOR to point back to BESTNODE.
    - Compute `g(SUCCESSOR) = g(BESTNODE)` + the cost of moving from BESTNODE to SUCCESSOR.
    - If SUCCESSOR is already on the open list:
        - Compare the new path cost to SUCCESSOR with its existing `g` value.
        - If the new path is cheaper, update SUCCESSOR's `g` value and its parent link to BESTNODE.
    - If SUCCESSOR is already on the closed list:
        - Compare the new path cost to SUCCESSOR with its existing `g` value.
        - If the new path is cheaper, update SUCCESSOR's `g` value, its parent link to BESTNODE, and propagate the improvement to its successors (re-evaluate their costs).
    - If SUCCESSOR is not on either list:
        - Add SUCCESSOR to the open list.
        - Set its parent link to BESTNODE.
        - Compute `f(SUCCESSOR) = g(SUCCESSOR) + h(SUCCESSOR)`.
5. Iteration: Repeat steps 2-4 until a solution is found or the open list is empty.

---

Admissibility and Optimality

- Admissibility
    - An algorithm is admissible if it guarantees finding the optimal path to the goal, if one exists.
    - A* is admissible if the heuristic function never overestimates the actual cost to reach the goal.
- Optimality
    - An algorithm is optimal if it finds the optimal path by exploring the fewest possible nodes.
    - A* is optimal under certain conditions, including:
        - The heuristic function is consistent (satisfies the triangle inequality).
        - The search space is a tree or the algorithm uses a graph search variant that prevents revisiting nodes on a less costly path.

---

Difference between A* and AO* Search Techniques

| Algorithm | Graph | Description |
|---|---|---|
| A* | OR | Finds the shortest path between two points |
| AO* | AND-OR | Finds the most cost-effective path between two points |

# A* and AO* Search Algorithm

What is AO* ?

- AO* is a knowledge representation and problem-solving technique for searching in AND-OR graphs.
- It is a generalization of the A* algorithm that can handle problems with multiple possible solutions.
- It is particularly well-suited for problems with uncertain or incomplete information.

How does AO* work ?

- AO* searches the graph by expanding the most promising nodes first.
- It uses a heuristic function to estimate the cost of reaching a goal from a given node.
- It also uses a cost function to keep track of the cost of reaching the current node.
- AO* continues searching until it finds a solution that satisfies all the constraints of the problem.

What are the advantages of AO* ?

- It can handle problems with multiple possible solutions.
- It is efficient for problems with large or infinite search spaces.
- It can handle problems with uncertain or incomplete information.

What are the disadvantages of AO* ?

- It can be computationally expensive for problems with many constraints.
- It may not find the optimal solution if the heuristic function is not accurate.

Applications of AO*

AO* can be used in a variety of applications, including:

- Game playing
- Robotics
- Natural language processing
- Machine learning

## Related posts:

1. Artificial Intelligence Intelligence Tutorial for Beginners
2. Difference between Supervised vs Unsupervised vs Reinforcement learning
3. What is training data in Machine learning
4. What other technologies do I need to master AI?
5. How Artificial Intelligence (AI) Impacts Your Daily Life ?
6. Like machine learning, what are other approaches in AI ?
7. Best First Search in AI
8. Heuristic Search Algorithm
9. Hill Climbing in AI
10. Knowledge Representation in AI
11. Propositional Logic and Predicate Logic
12. Resolution and refutation in AI
13. Deduction, theorem proving and inferencing in AI
14. Monotonic and non-monotonic reasoning in AI

15. Probabilistic reasoning in AI
16. Bayes' Theorem
17. Artificial Intelligence Short exam Notes
18. Transformer Architecture in LLM
19. Input Embedding in Transformers
20. Positional Encoding in Transformers
21. Multi-Head Attention in Transformers