UNIT 1

CHAPTER 3

Informed Search in Artificial Intelligence

Informed search, also known as heuristic search, is a type of search strategy in artificial intelligence (AI) that utilizes problem-specific knowledge (heuristics) to guide the search process toward the goal more efficiently. Unlike uninformed search algorithms like Breadth-First Search (BFS) and Depth-First Search (DFS), which explore the search space systematically but without any specific guidance, informed search algorithms make use of heuristics to prioritize which paths to explore, often leading to faster and more resource-efficient solutions.

Key Concepts in Informed Search

- **Heuristic Function (h(n)):**
  - The core of informed search is the heuristic function, denoted as h(n)h(n)h(n). A heuristic is an estimate of the cost from a given node nnn to the goal. This estimate is used to rank nodes in the search process, prioritizing those that are considered closer to the goal.
  - A good heuristic is one that closely estimates the actual minimum cost from the current node to the goal, leading to more efficient searches.
- **Evaluation Function (f(n)):**
  - In some informed search algorithms, the heuristic is combined with the actual cost from the start node to the current node nnn (denoted as g(n)g(n)g(n)) to form an evaluation function f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n). This function is used to select the next node to explore, balancing the cost so far with the estimated cost to the goal.
- **Optimality and Completeness:**
  - The optimality and completeness of an informed search algorithm depend on the heuristic used. A heuristic is **admissible** if it never overestimates the true cost to reach the goal. This property is crucial for ensuring that the search algorithm finds the optimal solution.
- **Search Strategy:**
  - Informed search strategies use the heuristic function to prioritize nodes. This means they don't explore all possible paths but rather focus on those that are

more likely to lead to the goal. This result in faster searches and less memory usage compared to uninformed methods.

Characteristics of Informed Search

- **Efficiency:**
  - By leveraging heuristics, informed search algorithms can dramatically reduce the number of nodes explored, leading to faster solutions. They avoid wasting time on paths that are unlikely to lead to the goal.
- **Heuristic Quality:**
  - The effectiveness of an informed search algorithm depends heavily on the quality of the heuristic. A good heuristic can make the search process much more efficient, while a poor heuristic might lead to suboptimal paths or longer search times.
- **Resource Usage:**
  - Informed search typically uses less memory and time than uninformed search because it narrows down the search space. However, in some cases, especially with complex heuristics, the computational overhead of calculating the heuristic can offset these gains.
- **Trade-offs:**
  - Informed search algorithms often involve trade-offs between accuracy and efficiency. A more sophisticated heuristic might provide better guidance but at the cost of increased computational complexity.
- **Admissibility and Consistency:**
  - **Admissibility** ensures that the heuristic never overestimates the cost to reach the goal, allowing the search algorithm to be both complete and optimal.
  - **Consistency (or Monotonicity)** ensures that the heuristic respects the triangle inequality, meaning the estimated cost to reach the goal from one node is never greater than the cost to reach another node plus the cost from that node to the goal.

Advantages of Informed Search

- **Reduced Search Space:** Informed search algorithms can drastically reduce the number of nodes explored by focusing on the most promising paths, which can lead to faster solutions.
- **Better Scalability:** For large problems, informed search scales better than uninformed search, as it doesn't need to exhaustively explore all possibilities.

- **Flexibility:** Informed search algorithms can be tailored to specific problems by designing appropriate heuristics, making them highly flexible and powerful for a wide range of applications.
- **Optimal Solutions:** When using admissible heuristics, informed search algorithms can guarantee finding the optimal solution.

Disadvantages of Informed Search

- **Heuristic Design Complexity:** Designing effective heuristics can be challenging, requiring deep knowledge of the problem domain. Poor heuristics can lead to inefficiencies or even incorrect results.
- **Computational Overhead:** Calculating the heuristic value for each node can add significant computational overhead, particularly in complex problems or when the heuristic is sophisticated.
- **Dependence on Heuristic Quality:** The success of informed search is heavily dependent on the quality of the heuristic. A weak heuristic can render the search no better than an uninformed approach.
- **Not Always Optimal:** If the heuristic is not admissible or consistent, the informed search may find suboptimal solutions.

Applications of Informed Search

- **Pathfinding:** In robotics, video games, and navigation systems, informed search algorithms like A* are commonly used to find the shortest path between two points.
- **Puzzle Solving:** In AI, informed search is often used in solving puzzles (e.g., the 8-puzzle, Rubik's cube) where the heuristic can significantly reduce the search space.
- **Optimization Problems:** In various optimization problems, such as scheduling and resource allocation, informed search helps in finding optimal solutions efficiently.
- **Artificial Intelligence Planning:** In AI planning, informed search is used to find the sequence of actions that lead to a desired outcome, where the heuristic can guide the search through the space of possible plans.

Strategies for providing heuristics information

The informed search algorithm is more useful for large search space. The entire informed search algorithm uses the idea of heuristic, so it is also called Heuristic search. “Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal” Heuristic Function: Heuristic information is provided in form of function called heuristic function.

Heuristic Function:

A heuristic function, denoted as h(n), is a key component of heuristic search algorithms. It serves as an estimate or guess of the cost from a given state (node n) to the goal state.

- **Functionality:**
    - The heuristic function takes the current state as input and returns an estimate of the remaining cost to reach the goal.
    - It helps the search algorithm to focus on paths that seem more promising based on the estimated cost.
    - The effectiveness of a heuristic search largely depends on the quality of the heuristic function used.
- **Examples of Heuristic Functions:**
    - **Euclidean Distance:** For problems involving physical distances (like route planning), the heuristic might be the straight-line (Euclidean) distance between two points.
    - **Manhattan Distance:** In grid-based environments (like puzzles), the Manhattan distance, which is the sum of the absolute differences in the x and y coordinates, is often used.

Key Properties of Heuristic Function h(n):

- **Estimate of Cost:** h(n) provides an estimate of the cheapest cost to reach the goal from node n.
- **Goal-Oriented:** It is goal-oriented, meaning it helps the algorithm move towards the goal more efficiently by estimating how "good" or "close" a state is to the goal.
- **Computationally Feasible:** The heuristic function must be computationally feasible to calculate for every state during the search process.

Heuristic Search Characteristics:

- **Efficiency**: Heuristic searches are more efficient than uninformed searches because they use additional information to prioritize paths that are more likely to lead to a solution.
- **Not Always Optimal:** While heuristic searches can quickly find good solutions, they might not always find the optimal solution (the absolute best one). However, they often find a satisfactory solution in a reasonable amount of time.
- **Domain-Specific:** The effectiveness of a heuristic function depends on the specific problem domain. A well-designed heuristic function can greatly enhance the performance of the search algorithm.

Types of Heuristic Search

Generate-and-Test Algorithm

The Generate-and-Test algorithm is a straightforward problem-solving method used in artificial intelligence. The basic idea is to generate potential solutions and then test each one to see if it meets the desired criteria or solves the problem.

Overview:

- **Generate:** The algorithm generates potential solutions or candidates.
- **Test:** Each candidate is tested to determine if it satisfies the problem's constraints or if it is the correct solution.
  This approach is a form of trial and error, where the algorithm iterates through possible solutions until it finds one that works.

Steps of the Generate-and-Test Algorithm:

1. **Generate a Candidate Solution:**
   - The algorithm begins by generating a possible solution. This can be done randomly, systematically, or based on some heuristics.
2. **Test the Solution:**
   - The generated solution is tested against the problem's constraints or criteria. If the solution meets the criteria, it is accepted as the correct solution.

3. **Accept or Reject:**
    - If the solution is correct (i.e., it solves the problem), the algorithm terminates and returns the solution.
    - If the solution is not correct, it is rejected, and the process returns to the generation step to produce a new candidate.
4. **Repeat:**
    - The process is repeated until a satisfactory solution is found or all possible solutions have been tested.

Characteristics:

1. **Simple to Implement:**
    - The Generate-and-Test algorithm is easy to implement because it doesn't require complex logic or sophisticated data structures.
2. **No Heuristic Required:**
    - Unlike informed search algorithms, Generate-and-Test doesn't rely on heuristic information. It simply generates and tests solutions.
3. **Exhaustive Search:**
    - The algorithm may end up performing an exhaustive search if no intelligent generation method is used, potentially leading to inefficiencies.
4. **Suitability:**
    - The algorithm is suitable for problems where generating a solution is straightforward, and the solution space is small enough that testing all possibilities is feasible.

Advantages:

1. **Simplicity:**
    - The algorithm is straightforward and easy to understand, making it suitable for simple problem-solving tasks.
2. **Versatility:**
    - It can be applied to a wide range of problems since it doesn't depend on specific problem domain knowledge.
3. **Deterministic or Random:**
    - The generation step can be deterministic or random, providing flexibility depending on the problem.

Disadvantages:

1. **Inefficiency:**

- If the solution space is large, the algorithm can be inefficient, as it may generate and test many invalid solutions before finding the correct one.

2. **No Guarantee of Optimality:**
   - The first solution found may not be the optimal one unless additional mechanisms are put in place.
3. **Potential for Non-Termination:**
   - If the algorithm continues to generate and test without any constraints on termination, it may run indefinitely.

Hill Climbing

Hill Climbing is an iterative optimization algorithm used for solving computational problems where the goal is to find the best solution (optimum) from a large set of possible solutions. It is a heuristic search algorithm that belongs to the family of local search techniques. Hill Climbing focuses on moving towards the optimal solution by iteratively making small adjustments to a current solution.

Key Concepts:

1. **Current State**: The current solution in the search space.
2. **Neighboring State**: A solution that is slightly modified from the current solution.
3. **Objective Function**: A function that measures how good a solution is (often represented as a cost or utility).
4. **Optimization Direction**:
   - **Maximization**: Moving towards higher values of the objective function.
   - **Minimization**: Moving towards lower values of the objective function.
5. **Termination Condition**: The algorithm stops when it reaches a peak (no better neighbors) or after a certain number of iterations.

Hill Climbing Process:

1. **Initialization**: Start with a randomly chosen initial solution (state).
2. **Evaluation**: Evaluate the objective function at the current state.
3. **Selection**: Look at the neighboring states and select the one with the best value according to the objective function.
4. **Move**: Move to the neighboring state with the best value.
5. **Repeat**: Repeat the evaluation and move steps until a termination condition is met.

Types of Hill Climbing

1. **Simple Hill Climbing** In simple hill climbing, the algorithm evaluates one neighbor at a time and moves to the first neighbor that offers an improvement in the objective function. It's a straightforward, step-by-step approach.

   Process:

   1. Start at an initial state.
   2. Evaluate the first neighbor.
   3. If it improves the objective function, move to that neighbor.
   4. If not, evaluate the next neighbor.
   5. Continue until no further improvements are possible.

   Limitation: This method can be inefficient because it might not find the best neighbor in cases where the first few neighbors are suboptimal.

2. **Steepest-Ascent Hill Climbing (Gradient Ascent):** This variant evaluates all neighboring states and moves to the one with the steepest ascent (or the greatest improvement in the objective function). It's more systematic than simple hill climbing.

   Process:

   1. Start at an initial state.
   2. Generate all neighbors.
   3. Evaluate all neighbors and select the one with the highest objective function value.
   4. Move to the best neighbor.
   5. Repeat the process until no neighbor offers an improvement.

   Advantage: More likely to find a better local optimum since it considers all neighbors.

   Limitation: Can be computationally expensive if the number of neighbors is large.

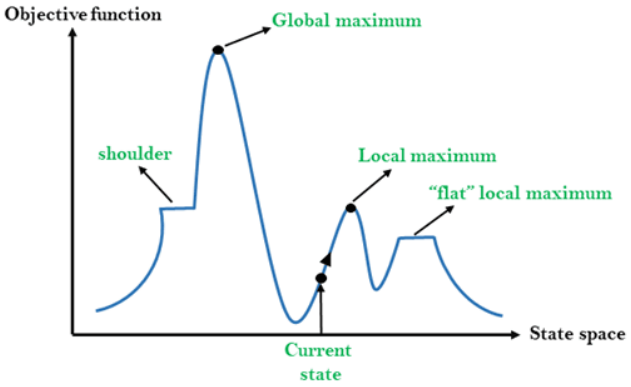
3. **Stochastic Hill Climbing:** Instead of evaluating all neighbors, stochastic hill climbing selects one neighbor at random and decides whether to move to that neighbor based on some probability criterion.

   Process:

   1. Start at an initial state.
   2. Randomly select a neighbor.
   3. If the selected neighbor improves the objective function, move to it.
   4. If not, select another neighbor and evaluate.
   5. Repeat until no improvement is possible.

   Advantage: Faster than steepest-ascent because it doesn't evaluate all neighbors, but can still escape some local optima.

   Limitation: The random nature might cause the algorithm to miss better solutions or take longer to find the optimum.



Advantages of Hill Climbing:

- **Simplicity**: Hill climbing is conceptually simple and easy to implement. The algorithm only requires a straightforward evaluation of neighbors and can often be coded with minimal effort.
- **Efficiency**: For problems where the solution space is well-behaved (e.g., a unimodal function where there's a single peak), hill climbing can efficiently converge to the optimal solution without needing to explore the entire search space.
- **Local Search**: It's a local search algorithm, which means it focuses on optimizing in the immediate vicinity of the current state. This makes it useful when the global structure of the search space is unknown or when computation needs to be localized.
- **Memory Usage**: Hill climbing generally uses very little memory, as it only needs to store the current state and its neighbors. This makes it suitable for problems with limited computational resources.
- **Fast Convergence**: In cases where the search space is smooth and the algorithm starts close to the optimum, hill climbing can converge rapidly to a solution.
- **Deterministic Nature**: In simple and steepest-ascent hill climbing, the algorithm is deterministic, which means that running the algorithm multiple times from the same starting point will always yield the same result.

Disadvantages of Hill Climbing:

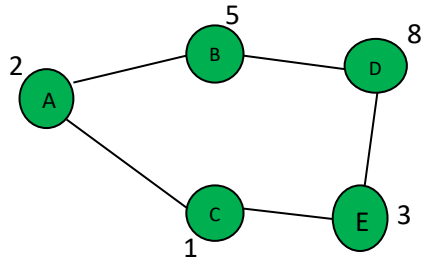
- **Local Optima**: One of the major drawbacks is that hill climbing can get stuck in local optima, especially in complex search spaces with multiple peaks and valleys. This means it might find a solution that is better than its neighbors but not the best possible solution in the entire search space.
- **Plateaus**: If the search space contains flat regions (plateaus) where the objective function does not change, hill climbing can struggle to make progress and may stop prematurely.

- **Ridges**: In some search spaces, the optimal path might be a narrow ridge, which is difficult for hill climbing to follow due to its greedy nature. It may miss the ridge entirely or fall off into a less optimal area.
- **No Backtracking**: Once hill climbing moves to a new state, it does not backtrack to consider previous states. This can be problematic if the algorithm makes a poor choice early on and cannot recover.
- **Dependency on Starting Point**: The outcome of hill climbing is heavily dependent on the starting point. A poor choice of the initial state can lead to suboptimal solutions. This makes the algorithm less reliable for problems with multiple local optima.
- **Lack of Exploration**: Hill climbing is an exploitative algorithm that focuses solely on immediate improvement. It lacks mechanisms for exploration, such as random jumps or backtracking, which means it may fail to discover better solutions that are far from the current state.
- **Not Suitable for All Problems**: Hill climbing is best suited for problems where the solution space is relatively simple and smooth. For more complex problems, especially those with many local optima, plateaus, or ridges, other algorithms like Simulated Annealing, Genetic Algorithms, or Particle Swarm Optimization might be more effective.

Example: Hill Climbing on a Graph

Let's consider a simple example to illustrate the Hill Climbing algorithm. Suppose you have a graph where nodes represent different states, and the edges between nodes represent the possible transitions between these states. The goal is to find the highest point (global maximum) on the graph.

Graph Example: Imagine a graph with the following nodes (states) and their corresponding objective function values:



Hill Climbing Steps:

1. **Initialization**:
   - Start at node **A** (Objective function value = 2).
2. **Evaluation**:
   - The neighbors of **A** are **B** and **C**.
   - Objective function values: **B** = 5, **C** = 1.
3. **Selection**:

- **B** has a higher objective function value than **C**.
- Move to **B**.

4. **Evaluation**:
   - The neighbors of **B** are **A** and **D**.
   - Objective function values: **A** = 2, **D** = 8.
5. **Selection**:
   - **D** has a higher objective function value than **A**.
   - Move to **D**.
6. **Evaluation**:
   - The neighbors of **D** are **B** and **E**.
   - Objective function values: **B** = 5, **E** = 3.
7. **Selection**:
   - Both neighbors have lower values than **D** (current state).
   - No move is made, and the algorithm terminates as it has reached a local maximum at **D** (Objective function value = 8).

Conclusion:

The Hill Climbing algorithm has found the local maximum at node **D** with a value of 8. In this example, **D** is the global maximum as well.

Best First Search

Best-First Search (BFS) Overview:

Best-First Search (BFS) is a search algorithm that explores a graph by expanding the most promising node chosen according to a specified rule or heuristic. Unlike Breadth-First Search (BFS) or Depth-First Search (DFS), which explore nodes uniformly or in a specific order, Best-First Search uses an **evaluation function** to prioritize which node to expand next, aiming to reach the goal as efficiently as possible. This function assesses the estimated cost or value of each node. Essentially, it predicts how promising a node is in terms of reaching the goal.

How It Works:

1. **Initialization**: Start with an open list (priority queue) containing the initial node. The open list is prioritized based on the heuristic function, h(n) which estimates the cost to reach the goal from node n.
2. **Node Expansion**:
    - Remove the node with the lowest heuristic value from the open list.
    - Expand this node by generating its successors (children).
    - Evaluate the successors using the heuristic function and add them to the open list.
3. **Goal Test**: If the node selected for expansion is the goal node, the search terminates successfully.
4. **Repeat**: If the goal is not found, return to step 2.

Complexity, Completeness, and Optimality:

- **Time Complexity**: The time complexity of Best-First Search depends on the heuristic function. In the worst case, it can be as bad as $O(b^d)$, where b is the branching factor and d is the depth of the shallowest goal node. However, if the heuristic is good (close to the actual cost), the time complexity can be much lower.
- **Space Complexity**: The space complexity is also $O(b^d)$ because all generated nodes are stored in memory.
- **Completeness**: Best-First Search is complete if the graph is finite and the heuristic function does not overestimate the cost to reach the goal.
- **Optimality**: Best-First Search is not guaranteed to be optimal unless the heuristic function is admissible (i.e., it never overestimates the true cost) and consistent (i.e., it satisfies the triangle inequality).

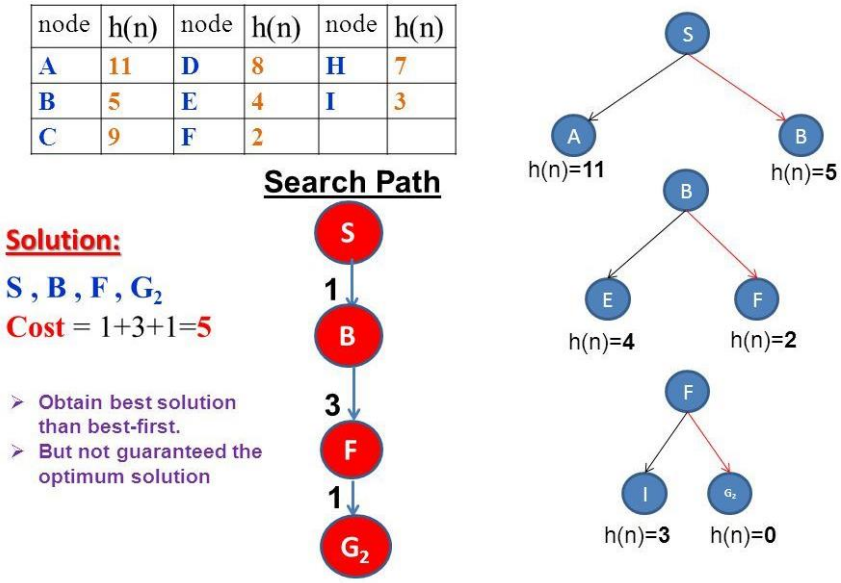
Advantages:

- **Efficiency**: Best-First Search can be more efficient than uninformed search algorithms like Breadth-First Search or Depth-First Search, especially when a good heuristic is used.

- **Guided Search**: It uses heuristics to guide the search, potentially reducing the number of nodes expanded.

Disadvantages:

- **Heuristic Dependence**: The performance of Best-First Search heavily depends on the quality of the heuristic. A poor heuristic can lead to inefficient searches.
- **Memory Usage**: Best-First Search can consume a lot of memory, as it needs to store all generated nodes in the open list.
- **Suboptimal Solutions**: It may find suboptimal solutions if the heuristic is not admissible or consistent.



node	h(n)	node	h(n)	node	h(n)
A	11	D	8	H	7
B	5	E	4	I	3
C	9	F	2		

Search Path

Solution:

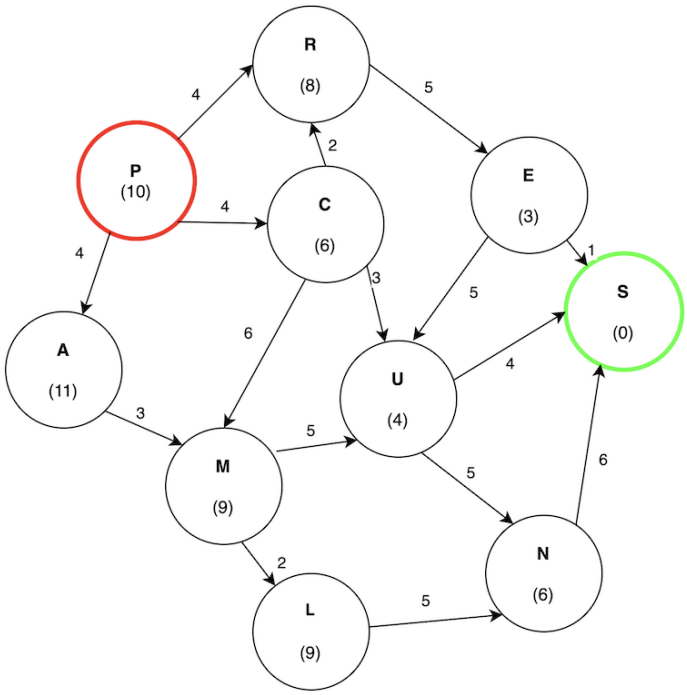
S, B, F, G_2

Cost = 1+3+1=5

- Obtain best solution than best-first.
- But not guaranteed the optimum solution

A* Algorithm

Consider finding the path from P to S in the following graph. In this example, the cost is measured strictly using the heuristic value. In other words, how close it is to the target.



The **total cost for the path (P -> C -> U -> S) evaluates to 11**. The **potential problem** with a greedy best-first search is revealed by the path **(P -> R -> E -> S) having a cost of 10**, which is lower than (P -> C -> U -> S). Greedy best-first search **ignored this path because it does not consider the edge weights.**

The A* algorithm is one of the most widely used search algorithms in artificial intelligence (AI) for finding the shortest path between two nodes in a graph. It combines features of both **Dijkstra's algorithm** (which considers the cost to reach a node) and **Greedy Best-First Search** (which uses a heuristic to estimate the cost to reach the goal). A* is both complete and optimal when the heuristic used is admissible and consistent.

How A* Works

1. **Initialization**:
    - Start with an open list (priority queue) that initially contains the start node.
    - Each node nnn has an associated cost f(n)=g(n)+h(n):
        - g(n): The exact cost of the path from the start node to nnn.
        - h(n): The heuristic estimate of the cost from nnn to the goal.

2. **Selection**:
    - Select the node with the lowest f(n) from the open list (priority queue).
3. **Expansion**:
    - Expand the selected node by generating all its successors and calculating their f(n) values.
    - If a successor is the goal node, the algorithm terminates, and the path is returned.
    - If a successor is not in the open list, it is added. If it is already in the open list with a higher f(n) value, the node's value is updated.
4. **Repeat**:
    - Continue selecting and expanding nodes until the goal node is reached or the open list is empty.

Completeness, Optimality, and Complexity

- **Completeness:** A* is complete, meaning it will find a solution if one exists, provided the graph has a finite number of nodes.
- **Optimality:** A* is optimal, meaning it will find the least-cost path to the goal, as long as the heuristic h(n)h(n)h(n) is admissible (never overestimates the true cost) and consistent (satisfies the triangle inequality).
- **Time and Space Complexity:**
    - The time and space complexity of A* is generally O(bd)O(b^d)O(bd), where bbb is the branching factor (the average number of successors per state) and ddd is the depth of the optimal solution.
    - In the worst case, A* might need to explore a large number of nodes, making it computationally expensive in terms of both time and memory.

Heuristic Function

The heuristic function h(n) is crucial to A*'s efficiency:

- **Admissible Heuristic:** A heuristic is admissible if it never overestimates the cost to reach the goal. This ensures the optimality of A*.

- **Consistent (or Monotonic) Heuristic:** A heuristic is consistent if, for every node n and every successor n’ of n, the estimated cost of reaching the goal from n is no greater than the step cost from n to n’ plus the estimated cost from n’ to the goal. Consistency ensures that A* will not need to revisit nodes.

Applications in AI

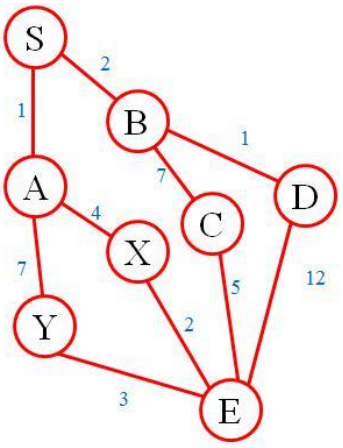
- **Pathfinding:** A* is extensively used in AI for pathfinding in maps, such as in video games, robotics, and geographic information systems (GIS).
- **Game AI:** In turn-based strategy games, A* can be used to determine the best move by searching through potential future states.
- **Route Planning:** Autonomous vehicles use A* for planning routes through complex environments, considering both the distance to the goal and potential obstacles.

Advantages

1. Optimal and Complete: As long as the heuristic is admissible and consistent, A* guarantees finding the shortest path.
2. Flexible: Can be adapted to different problems by changing the heuristic function.

Disadvantages

1. High Memory Usage: A* can consume a lot of memory, especially for large search spaces, as it stores all explored nodes.
2. Computationally Expensive: A* can be slow for very large or complex graphs, particularly when the heuristic function is not well-chosen.



- Values for h:
  A:5, B:6, C:4, D:15, X:5, Y:8

Expand S
{S,A} f=1+5=6
{S,B} f=2+6=8

Expand A
{S,B} f=2+6=8
{S,A,X} f=(1+4)+5=10
{S,A,Y} f=(1+7)+8=16

Expand B
{S,A,X} f=(1+4)+5=10
{S,B,C} f= (2+7)+4=13
{S,A,Y} f=(1+7)+8=16
{S,B,D} f= (2+1)+15=18

Expand X
{S,A,X,E} is the best path... (costing 7)

AO* Algorithm

The AO* (And-Or) algorithm is a search algorithm used in AI for solving problems that can be represented as **AND-OR trees**. Unlike traditional search trees, which are OR trees (where each node has multiple alternative branches), AND-OR trees have nodes that represent subproblems that must all be solved (AND nodes) or any one of which can be solved (OR nodes). AO* is particularly useful in situations where the solution to a problem requires combining the solutions of several subproblems.

Overview

AO* is designed to handle problems that can be decomposed into smaller subproblems, with dependencies between them. It finds the optimal solution by exploring the AND-OR tree, updating the cost estimates, and using a best-first strategy to guide the search. The algorithm iteratively refines its estimate of the best solution by exploring promising branches and updating the cost estimates based on the results of these explorations.

Key Concepts

1. **AND-OR Tree**:
    - **OR Nodes**: A node where solving any one of its child nodes leads to solving the parent node.
    - **AND Nodes**: A node where solving all its child nodes is necessary to solve the parent node.
2. **Heuristic Cost**: Similar to A*, AO* uses heuristic costs to estimate the cost of reaching the goal from a node.
3. **Path**: A sequence of nodes from the root node to a goal node in the AND-OR tree.
4. **Solution Graph**: A subgraph of the AND-OR tree that represents the best solution found by the AO* algorithm.

AO* Algorithm Steps

1. **Initialization**:
    - Start at the root node of the AND-OR tree.
    - Initialize the cost of the root node to the heuristic estimate h(root).
2. **Expansion**:
    - Select the most promising unexpanded node (based on the lowest cost estimate).
    - Expand this node by generating its children.
    - If the node is an OR node, add all possible children.
    - If the node is an AND node, add all necessary children.

3. **Cost Propagation**:
    - Update the cost estimates of the node and its ancestors based on the costs of its children.
    - For OR nodes: **f(parent)=min{f(child$_i$)}+cost to reach the child.**
    - For AND nodes: **f(parent)=∑{f(child$_i$)}+cost to reach all children**.
4. **Check for Termination**:
    - If the selected node is a goal node (no further expansion required), backtrack to update the cost estimates and the solution graph.
    - If the root node's cost estimate is stable and represents a complete solution, the algorithm terminates with the optimal solution graph.
5. **Repeat**:
    - Continue expanding and updating the tree until a solution graph is identified.

Completeness, Optimality, and Complexity

- **Completeness**: AO* is complete as long as the AND-OR tree is finite and the heuristic is admissible. It will eventually find a solution if one exists.
- **Optimality**: AO* is optimal, finding the least-cost solution graph, provided the heuristic is admissible (never overestimates the true cost).
- **Time and Space Complexity**: The complexity of AO* is heavily dependent on the structure of the AND-OR tree and the heuristic used. In the worst case, the complexity can be exponential.

Advantages

1. **Handles Complex Problems**: AO* can efficiently handle problems that require solving multiple interdependent subproblems.
2. **Optimal Solutions**: It guarantees finding the optimal solution if the heuristic is admissible.
3. **Modular**: The AND-OR structure allows for modular problem-solving, making it easier to break down and solve complex problems.

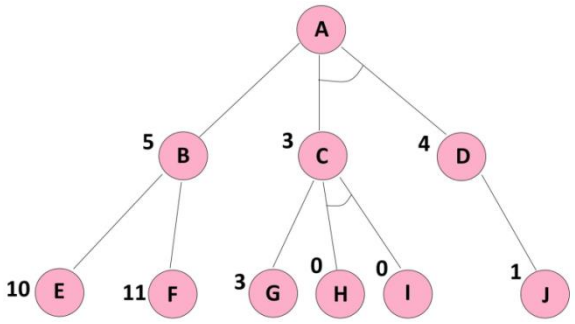
Disadvantages

1. **High Memory Usage**: AO* may require significant memory resources as it explores multiple branches of the AND-OR tree simultaneously.
2. **Complexity**: The algorithm can become complex to implement, especially for large or intricate AND-OR trees.

Applications

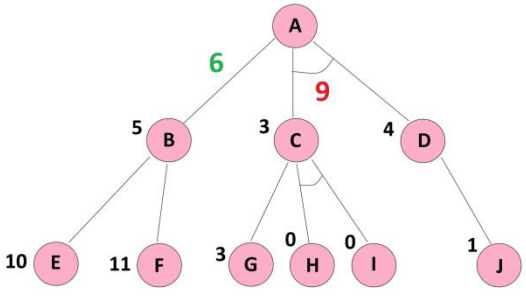
- **Planning and Decision Making**: Used in AI for planning problems where actions have multiple outcomes , and some outcomes require the completion of multiple sub-tasks.
- **Problem Decomposition**: Useful in problems where a large problem can be broken down into smaller, interdependent subproblems , such as in hierarchical task networks (HTN).
- **Expert Systems**: Applied in AI systems that involve reasoning with AND-OR trees, such as expert systems for diagnostic purposes.

Example: Let's see an example below to explain AO* step by step to find the lowest cost path from the starting node A to the goal node:



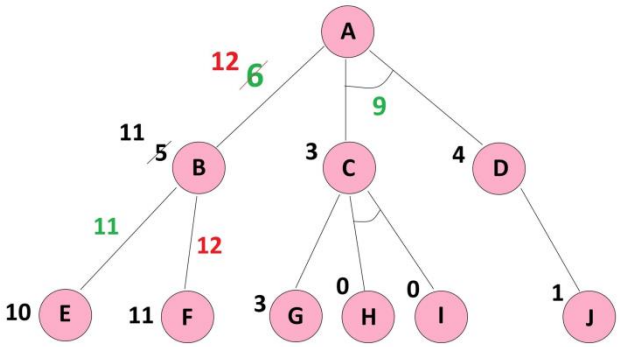
It should be noted that the cost of each edge is the same as 1, and the heuristic cost to reach the goal node from each node of the graph is shown beside it.

- First, we begin from node A and calculate each of the OR side and AND side paths. The OR side **path P(A-B) = G(B) + H(B) = 1 + 5 = 6**, where 1 is the cost of the edge between A and B, and 5 is the estimated cost from B to the goal node.
- The **AND side path P(A-C-D) = G(C) + H(C) + G(D) + H(D) = 1 + 3 + 1 + 4 = 9,** where the first 1 is the cost of the edge between A and C, 3 is the estimated cost from C to the goal node, the second 1 is the cost of the edge between A and D, and 4 is the estimated cost from D to the goal node. Let's see an example below:



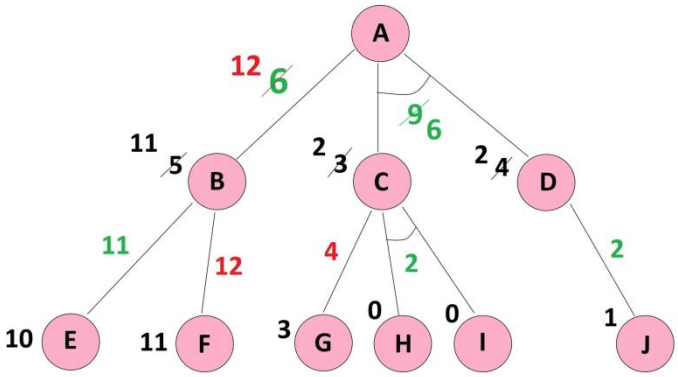
Since the cost of **P(A-B) is the minimum cost path**, we proceed on this path in the next step.

- In this step we **continue on the P(A-B) from B to its successor nodes** i.e., E and F, where **P(B-E) = 1 + 10 = 11 and P(B-F) = 1 + 11 = 12**. Here, **P(B-E) has a lower cost** and would be chosen.
- Now, we have reached the bottom of the graph where no more level is given to add to our information. Therefore, **we can do the backpropagation** and correct the heuristics of upper levels. In this vein, the **updated H(B) = P(B-E) = 11**, and as a consequence the **updated P(A-B) = G(B) + updated H(B) = 1 + 11 = 12**. Let's see an example below:
  Now, we can see that **P(A-C-D) with a cost of 9 is lower than the updated P(A-B) with a**



cost of 12. Therefore, we need to proceed on this path to find the minimum cost path from A to the goal node.

- In this step, we do the **calculations for the AND side path**, i.e., P(A-C-D), and first explore the paths attached to node C. In this node again we have an OR side where P(C-G) = 1 + 3 = 4, and an AND side where P(C-H-I) = 1 + 0 + 1 + 0 = 2, and as a consequence the updated H(C) = 2.
- Also, the updated H(D) = 2, since P(D-J) = 1 + 1 = 2. By these updated values for H(C) and H(D), the updated P(A-C-D) = 1 + 2 + 1 + 2 = 6. Let's see an example below:



This **updated P(A-C-D) with the cost of 6 is still less than the updated P(A-B) with the cost of 12,** and therefore, the **minimum cost path** from A to the goal node goes from **P(A-C-D)** by the cost of 6. We are done.

Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is a type of problem in artificial intelligence where the goal is to find a solution that satisfies a set of constraints. CSPs are widely used in various fields, including AI, operations research, and computer science, to model and solve problems that involve a large number of variables with specific constraints.

Components of a CSP

A CSP consists of three main components:

1. **Variables:**
    - These are the unknowns in the problem that need to be assigned values.
    - Example: X1,X2,…,XnX_1, X_2, \dots, X_nX1,X2,…,Xn.
2. **Domains:**
    - Each variable has a domain, which is a finite set of possible values that the variable can take.
    - Example: Domain(X1)={1,2,3} \text{Domain}(X_1) = \{1, 2, 3\}Domain(X1)={1,2,3}.
3. **Constraints:**
    - These are restrictions on the values that variables can simultaneously take.
    - Constraints can be unary (involving one variable), binary (involving two variables), or higher-order (involving three or more variables).
    - Example: X1≠X2X_1 \neq X_2X1優=X2 (a binary constraint ensuring that X1X_1X1 and X2X_2X2 cannot have the same value).
    -

Example of a CSP: Sudoku Puzzle

In a Sudoku puzzle:

- Variables: Each empty cell in the Sudoku grid.
- Domains: The set of possible numbers (1-9) that can be placed in each cell.
- Constraints:
    - Each row must contain the numbers 1-9 without repetition.
    - Each column must contain the numbers 1-9 without repetition.
    - Each 3x3 subgrid must contain the numbers 1-9 without repetition.

Solving CSPs

There are various methods to solve CSPs, ranging from simple to complex:

1. **Backtracking:**
    - A depth-first search method where variables are assigned values one by one.
    - If a variable assignment leads to a violation of constraints, the algorithm backtracks to the previous variable and tries a different value.
2. **Forward Checking:**
    - An enhancement of backtracking where after assigning a value to a variable, the domains of the unassigned variables are checked and reduced.
    - If a variable's domain becomes empty, the algorithm backtracks.
3. **Constraint Propagation:**
    - Techniques like Arc Consistency (AC-3) are used to reduce the search space by propagating constraints.
    - If a variable's value can't satisfy the constraints with another variable, that value is removed from the domain.
4. **Heuristics:**
    - Minimum Remaining Values (MRV): Choose the variable with the fewest legal values left in its domain.
    - Degree Heuristic: Choose the variable involved in the largest number of constraints on other unassigned variables.
    - Least Constraining Value: Choose the value that leaves the most options open for the other variables.
5. **Local Search:**
    - Techniques like Min-Conflicts try to resolve conflicts in an incomplete assignment by iteratively adjusting variable values.

Properties of CSPs

1. **Completeness**: A CSP-solving algorithm is complete if it is guaranteed to find a solution if one exists.
2. **Optimality:** A CSP-solving algorithm is optimal if it finds the best solution according to some criterion, such as minimizing the number of constraint violations.
3. **Complexity:**
    - The complexity of solving a CSP depends on the number of variables, the size of their domains, and the number of constraints.
    - The worst-case time complexity of a backtracking algorithm is exponential in the number of variables.

Applications of CSPs

- **Scheduling:** Assigning tasks to time slots while satisfying constraints like resource availability and deadlines.
- **Resource Allocation:** Allocating resources (like bandwidth or CPU time) in a way that satisfies constraints on resource usage.
- **Design Problems:** Ensuring that all design specifications and constraints are met in engineering.
- **Puzzle Solving:** Problems like Sudoku, crosswords, and map coloring.

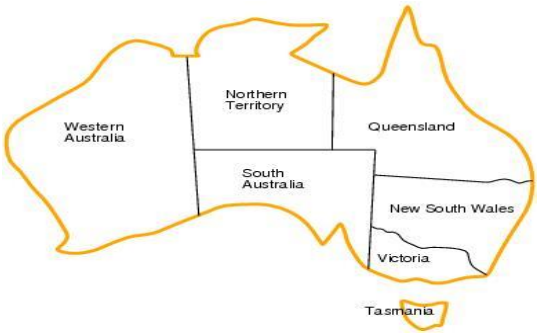
Advantages of CSPs

- **Flexibility:** Can model a wide range of problems in a declarative way.
- **Powerful Search Techniques:** Many sophisticated algorithms have been developed to efficiently solve CSPs.
- **Incremental Problem Solving:** Solutions can be built incrementally, making it easy to integrate new constraints

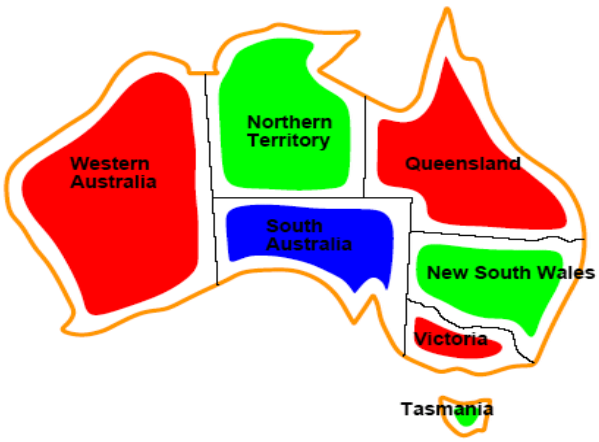
Disadvantages of CSPs

- **Scalability:** The complexity of CSPs can grow rapidly with the number of variables and constraints, making them difficult to solve for large problems.
- **Combinatorial Explosion:** As the problem size increases, the search space can become prohibitively large.

CSP example: map coloring



- Variables: *WA, NT, Q, NSW, V, SA, T*
- Domains: D_i*={red,green,blue}*
- Constraints: adjacent regions must have different colors.
    - E.g. $WA \neq NT$ (if the language allows this)
    - E.g. $(WA,NT) \neq$ *{(red,green),(red,blue),(green,red),…}*



- Solutions are assignments satisfying all constraints, e.g.
  {WA=red,NT=green,Q=red,NSW=green,V=red,SA=blue,T=green}