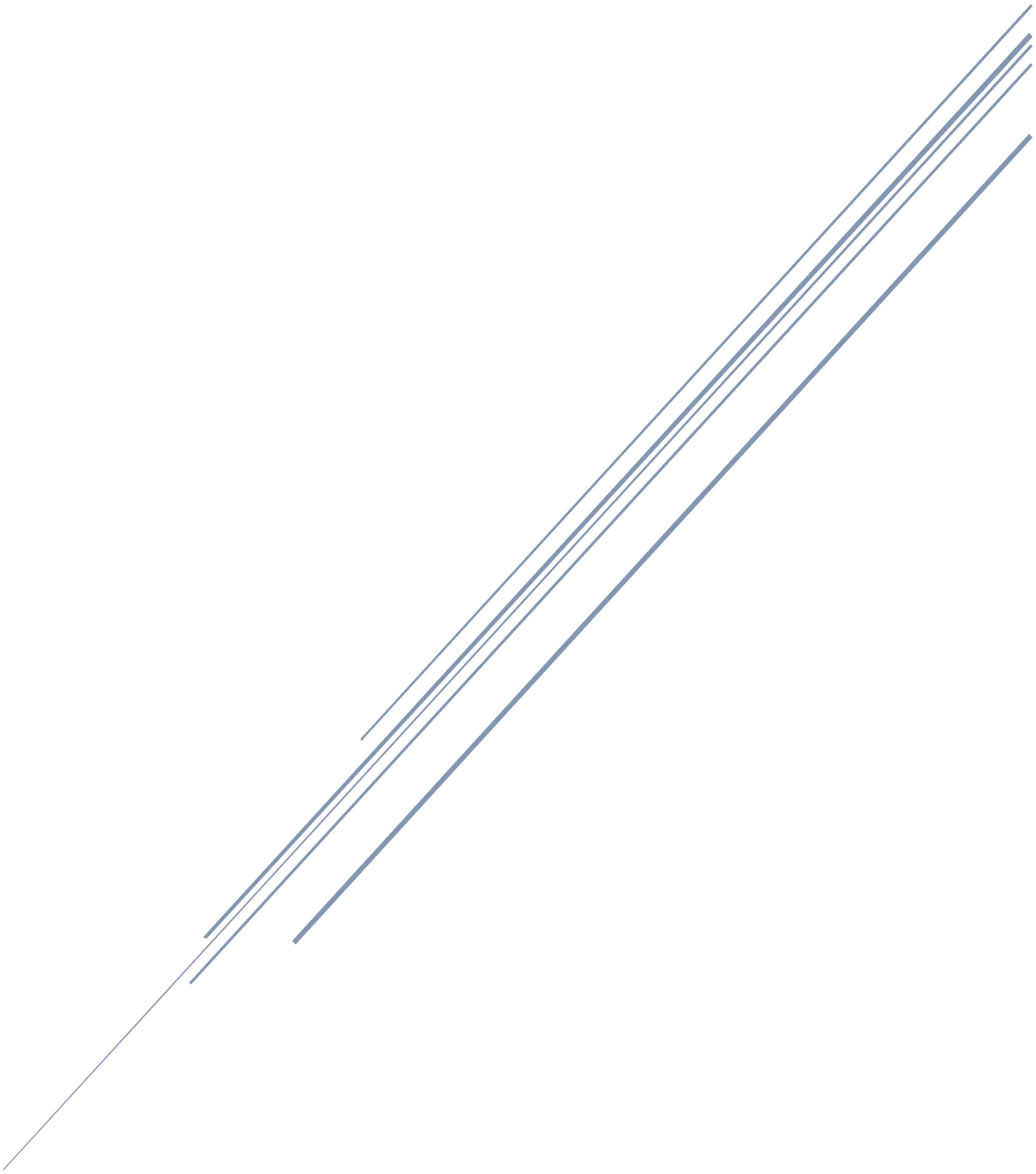
CHAPTER 1

Solving Problems by Searching



1.1 Introduction to intelligent agents

1.1.1 What is an agent?

An **agent** is anything that can be viewed as perceiving its environment through sensors and acting on that environment through actuators (see Figure 1.1).

For example, a software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

A robotic agent might have cameras for sensors and motors for actuators.

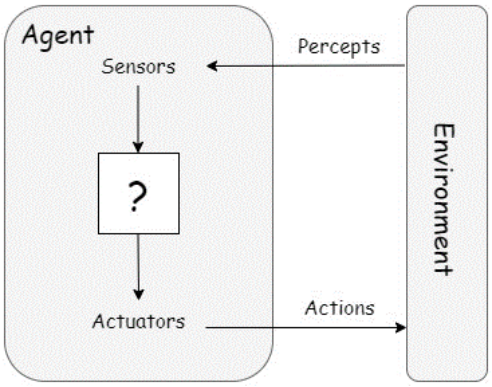


Figure 1. 1 Agent interacting with its environment

The agent's behavior is described by the **agent function** that maps any given percept sequence to an action. We evaluate the behavior of an agent using a performance measure; thus, the agent acts so as to maximize the expected value of the performance measure.

1.1.2 Intelligent agent

The job of AI is to design an agent program that implements the agent function.

There exists a variety of basic agent-program designs:

1- **Simple reflex agent:** This kind of agents selects actions on the basis of the current percept, ignoring the rest of the percept history (responds directly to percepts). It acts according to a set of condition-action rules. The structure of a simple reflex agent is shown in Figure 1.2.

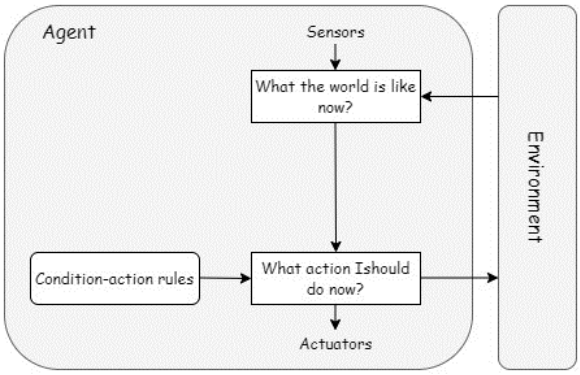


Figure 1. 2 Simple reflex agent

2- **Goal-based agent:** Intelligent agents are supposed to maximize their performance measure. Achieving this is sometimes simplified if the agent can adopt a goal and aim at satisfying it. The structure of a goal-based agent is shown Figure 1.3.

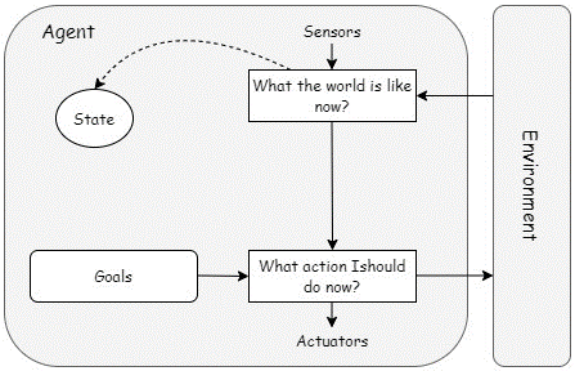


Figure 1. 3 Goal-based agent

1.2 Problem-solving agent

One kind of goal-based agent is called a problem-solving agent. A problem-solving agent has three phases:

- ✧ Problem formulation;
- ✧ Searching solution;
- ✧ Executing actions in the solution;

Example 1: Pathfinding Problem

Imagine an agent on holiday in a country **X**. Suppose this agent is currently in city **A** and he has a flight that leaves tomorrow from city **M**. The road map of the country **X** is given in Figure 1.4.

State: In which city is the agent, i.e., the current location of the agent.

Goal: The goal of the agent is to be in city **M** as soon as possible, i.e., finding the shortest path from city **A** to city **M**, according to the road map.

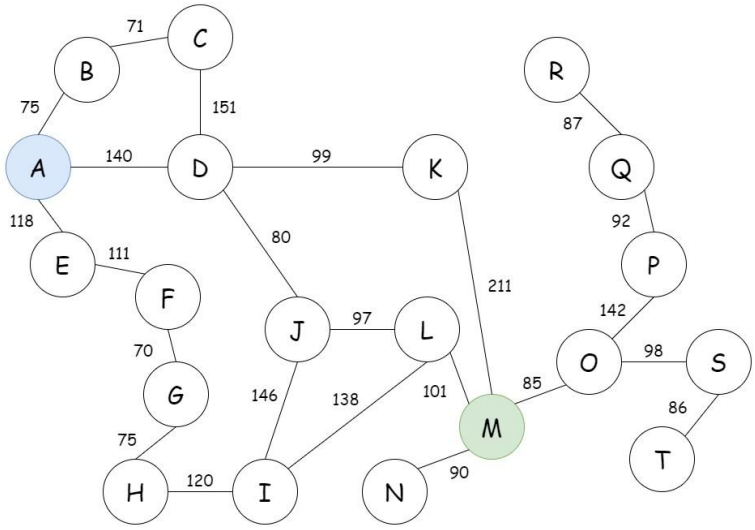


Figure 1. 4 Example 1 graph (Country X road map)

1.2.1 Problem Formulation

One of the dominant approaches to AI problem solving is to formulate a problem/task as as search in a state space. A **problem** can be defined formally by the following four components:

- ✧ **Initial state:** the state that the agent starts in. For example, the initial state for the agent in the pathfinding problem might be described as *In(A)*.
- ✧ **Successor function:** the function that changes the state of the agent.

  Given a particular state *s*, *successorsFn(s)* returns a set of *(action, successor)* ordered pairs, where each *action* is one of the legal actions in state *s* and each *successor* is a state that can be reached

from *s* by applying the action. An alternative formulation uses a set of operators that can be applied to a state to generate successors. For example, from the state *In(A)*, the successor function for the pathfinding problem would return {(*GoTo(B)*, *In(B)*), (*GoTo(D)*, *In(D)*), (*GoTo(E)*, *In(E)*)}, where *GoTo(B)*, *GoTo(D)* and *GoTo(E)* are the legal actions in state *In(A),* and *In(B)*, *In(D)*, *In(E)* are the successors.

- **Goal test:** which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. For example, the agent's goal in the pathfinding problem is the singleton set {*In(M)*}.

Sometimes the goal is specified by a property rather than a set of one or more states. For example, in chess, the goal is to reach a state called "checkmate," where the opponent's king is under attack and can't escape.

State space is defined by the set of all states reachable from the initial state. Hence, the initial state and the successor function together implicitly define the state space of the problem.
The state space forms a directed graph in which the nodes are states and the arcs between nodes are actions as shown in Figure 1.5. A path in the state space is a sequence of states connected by a sequence of actions.

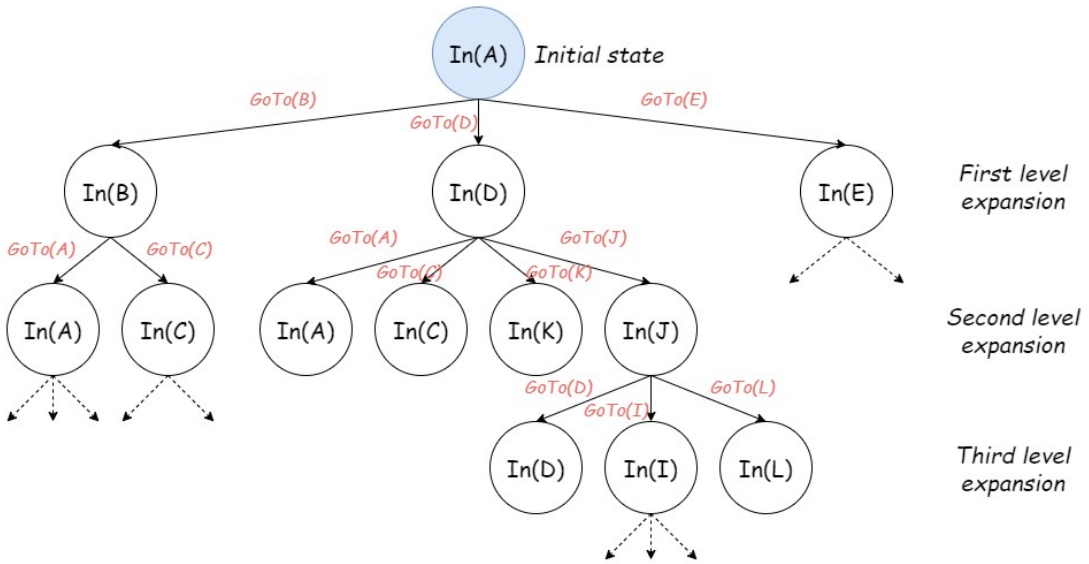


Figure 1. 5 Search tree for the pathfinding problem

- **Path cost:** It's the function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance. We can assume that the cost of a path can be described as the sum of the costs of the individual actions along the path. The **step cost** of taking action **a** to go from state **s1** to state **s2** is denoted by *c(s1, a, s2)*. The step cost for the pathfinding problem is the distance between the adjacent cities. For example, *c(In(A), GoTo(B), In(B)) = 75*.

A **solution** to a problem is a sequence of actions from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions. For example, in pathfinding problem, the optimal solution is to find the shortest path from city **A** to city **M**.

1.2.2 Example problems

Example (a): Vacuum Cleaner Problem

Suppose we have a house with two rooms and one vacuum cleaner, and there is dirt in both of the rooms. In this problem, the vacuum cleaner is the problem-solving agent. The starting point of the vacuum cleaner is one of the dirty rooms, and its goal is to clean up the entire area.

Problem formulation:

- ***States:*** the state is determined by the location of the agent (the agent is in one of two rooms) and the condition of the rooms (dirty or clean). Thus, there are 2 (possible agent locations) * 2^2 (possible room conditions) = 8 possible states.
- ***Initial state:*** Both rooms are dirty, and the vacuum cleaner is in one of these rooms (see Figure 1.6).

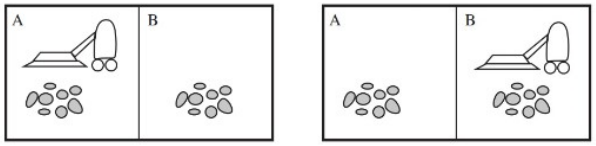


Figure 1. 6 Possible initial states for the vacuum cleaner problem

- ***Goal test*:** This test checks whether both rooms are clean (see Figure 1.7).

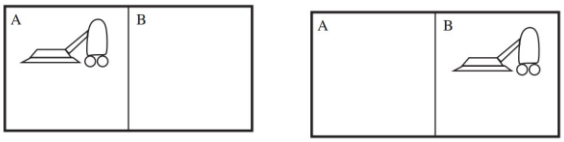


Figure 1. 7 Goal states for the vacuum cleaner problem

- ***Successor function*:** This function generates the possible states that result from applying the three legal actions:

  L: The vacuum cleaner moves to the left

  R: The vacuum cleaner moves to the right

  S: The vacuum cleaner sucks the dirt

  The complete state space is shown in Figure 1.8

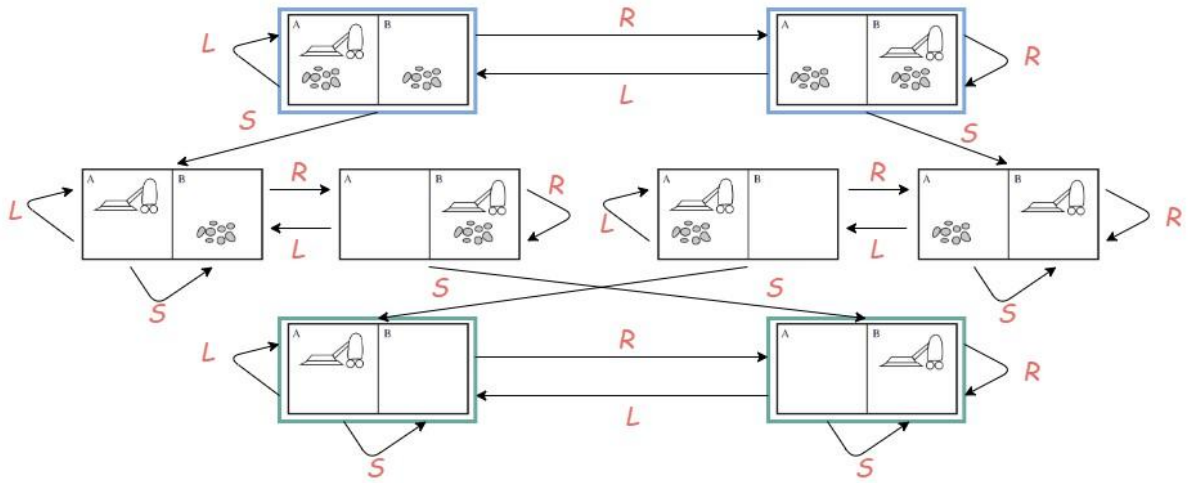


Figure 1. 8 State space for the vacuum cleaner problem

- ***Path cost*:** The cost of each step is 1, so the path cost corresponds to the number of steps in the path.

Example (b): Traveling Salesman Problem (TSP)

Given a set of cities and the distance between every pair of cities, the TSP problem is to find the shortest possible route (with the minimum cost) that visits every city exactly once and returns to the starting point (see Figure 1.9).

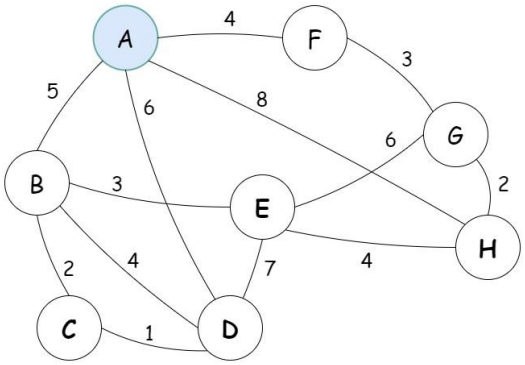


Figure 1. 9 TSP graph

Problem formulation:

- ***States:*** A state is defined by a sequence of one or more cities visited by the agent. Each state must include not only the agent's current location but also the set of cities it has visited.
- ***Initial state:*** The initial state is the state ***A***.
- ***Successor function:*** Each legal action corresponds to a trip to an adjacent city, with the precondition that each city must be visited exactly once. Therefore, the successor function generates the possible states that result from applying these actions.
- ***Goal test:*** The goal test checks whether the agent is in the state ***A*** and has visited all 8 cities with the lowest cost.
  An example of a path in the state space is shown in Figure 1.10.
- ***Path cost:*** The cost of each step is the distance between the adjacent cities, so the path cost is the sum of the distances.

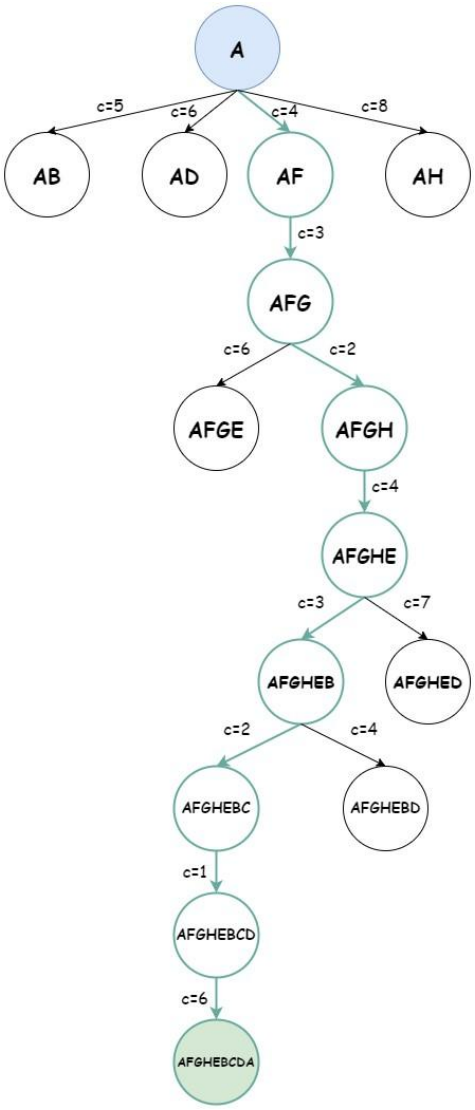


Figure 1. 10 Search tree for the TSP

Example (c): Blocks World Problem

We have a table with three blocks from A to C, arranged as shown in Figure 1.11 (a). Our goal is to rearrange them as shown Figure 1.11 (b). A robot arm can move only one block at a time, whether it's on the surface or on top of another block.

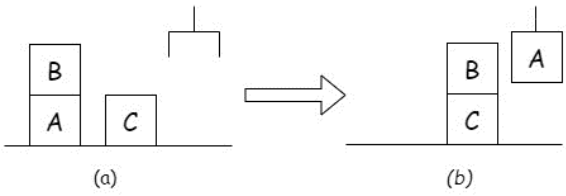


Figure 1. 11 Example of the Blocks world problem

Problem formulation:

✧ ***States:*** We will use a form of predicate logic to represent the state. Given below is the list of predicates as well as their meanings:

On(A, B): Block A is on block B.
OnTable(A): Block A is on the table.
Clear(A): Nothing is on top of block A.
Holding(A): The arm is holding block A.
ArmEmpty: The arm is holding nothing.

A state will be represented as the logical conjunction (AND) of the listed predicates.

✧ ***Initial state:*** *OnTable(A)^On(B, A)^OnTable(C)^Clear(B)^Clear(C)^ArmEmpty* (see Figure 1.12).

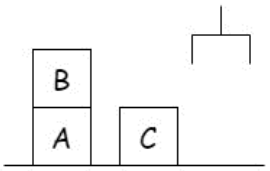


Figure 1. 12 Initial state for the blocks world problem

✧ ***Successor function:*** The robot arm can perform four actions:

Stack(A, B): stacking block A on top of block B.
Unstack(A, B): unstack block A from the top of block B.
PickUp(A): Pick up block A from the table.
PutDown(A): Place block A on the table.

Each of these actions has specific preconditions, which are represented using predicates. The effects of these actions are represented using two lists: DELETE and ADD. The DELETE list includes the predicates which will cease to be true once the action is performed, while the ADD list includes the predicates which will become true. Therefore, the successor function generates possible states (the conjunction of predicates) according to the information in Table 1.1.

Table 1. 1 Blocks world successor function

Action	**Precondition**	**DELETE**	**ADD**
Stack(A, B)	*Clear(B)^Holding(A)*	*Clear(B)^Holding(A)*	*On(A, B)^ArmEmpty*
Unstack(A, B)	*On(A, B)^ArmEmpty^Clear(A)*	*On(A, B)^ArmEmpty*	*Clear(B)^Holding(A)*
PickUp(A)	*OnTable(A)^ArmEmpty^Clear(A)*	*OnTable(A)^ArmEmpty*	*Holding(A)*
PutDown(A)	*Holding(A)*	*Holding(A)*	*OnTable(A)^ArmEmpty*

✧ ***Goal test:*** The goal test checks whether the goal state, represented as *OnTable(C)^Clear(B)^On(B, C)^Clear(A)^Holding(A)*, has been reached (see Figure 1.14).

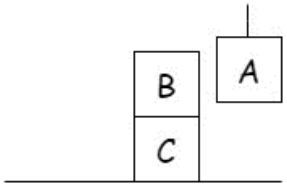


Figure 1. 13 Goal state for the blocks world problem

Figure 1.13 shows an example of a search tree for the blocks world problem.

Figure 1. 14 Search tree for the blocks world problem

✧ ***Path cost:*** The cost of each step is 1, so the path cost is the number of steps in the path.

1.3 Searching for solutions

Having formulated a problem, we now need to solve it. This is done by a search through the state space. Search techniques use an explicit search tree to explore the state space.

The building block of a search tree is the node. There are many ways to represent nodes, but we will assume that a node is a data structure with five components (see Figure 1.15):

✧ ***State:*** the state in the state space to which the node corresponds;

✧ ***Parent-node:*** the node in the search tree that generated this node;

✧ ***Action:*** the action that was applied to the parent to generate the node;

✧ ***Path-cost:*** the cost, usually denoted by ***g***, of the path from the initial node;

✧ ***Depth:*** the number of steps along the path from the initial state.

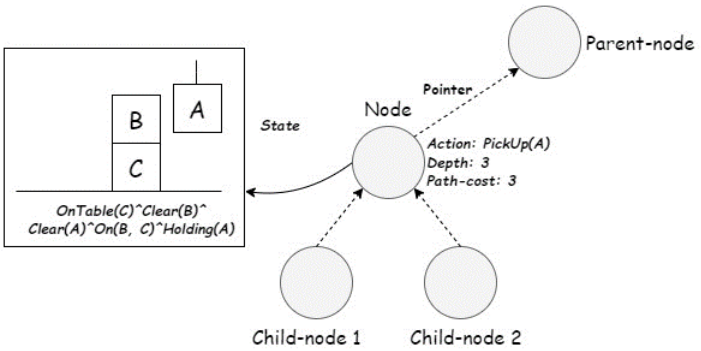


Figure 1. 15 Structure of a node in a search tree

1.3.1 General search algorithm

- The root of the search tree corresponds to the initial state;
- The first step is to test whether this initial state is a goal state. If it is not, we must explore other states;
- We expand the current state by applying the successor function, which generates a new set of states;
- We continue the process of choosing, testing, and expanding until we either reach a goal state (i.e., find a solution) or exhaust all available states.
- The choice of which state to expand is determined by the **search strategy/search algorithm**.
- To assess the effectiveness of a search algorithm; we consider both the search cost and the path cost of the solution found.

The pseudocode for the general tree search algorithm is presented in Figure 1.16.

Input: The problem to be solved
Output: A solution or failure
Function TreeSearch (problem)
Begin
  Initialize the *Open* list using the initial state of the problem
  While the *Open* list is not empty do
    Choose a leaf node and remove it from the *Open* list
    If the chosen node contains a goal state, then return the corresponding solution
    Expand the chosen node
    Add the resulting nodes to the *Open* list only if they are not already in the *Open* list
  Return failure
End

Figure 1. 16 Pseudocode for the general tree search algorithm

The pseudocode for the general graph search algorithm is shown in Figure 1.17.

Input: The problem to be solved
Output: A solution or failure
Function GraphSearch (problem)
Begin
  Initialize the *Open* list using the initial state of problem

```
Initialize the Closed list to be empty
While the Open list is not empty do
    Choose a leaf node and remove it from the Open list
    If the chosen node contains a goal state, then return the corresponding solution
    Add the chosen node to the Closed list
    Expand the chosen node
    Add the resulting nodes to the Open list only if they are not already in the Open or Closed lists
Return failure
End
```

Figure 1. 17 Pseudocode for the general graph search algorithm

1.3.2 Measuring problem-solving performance

Search strategies are evaluated based on the following dimensions:

- ***Completeness:*** Does the algorithm always find a solution if one exists?
- ***Optimality:*** Does it always find a solution with the lowest path cost?
- ***Time complexity:*** How many steps does it require to find a solution?
- ***Space complexity:*** How much memory is needed to perform the search?

Time and space complexity are expressed in terms of three quantities (refer to Figure 1.18):

b: The branching factor, which is the maximum number of successors of any node;

d: The depth of the shallowest goal node;

m: The maximum depth of the state space (which may be $+\infty$);

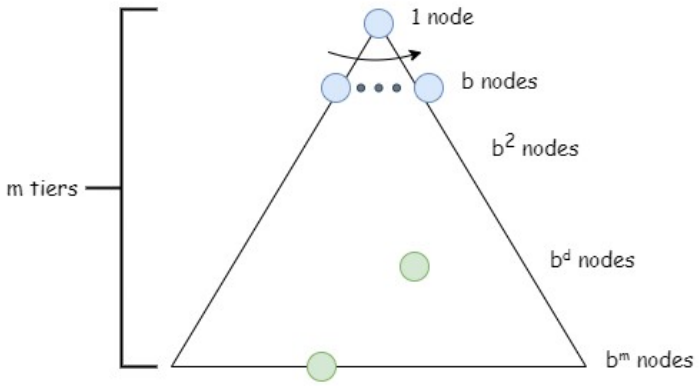


Figure 1. 18 Time and space complexity parameters

1.3.3 Uninformed search algorithms (blind search)

These algorithms have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state.

1.3.3.1 Breadth-first search (BFS)

This algorithm starts at the root node of the tree and visits all nodes at the current depth level before moving on to the nodes at the next depth level. The BFS algorithm uses a first-in-first-out (FIFO) list, known as a queue, for its *Open* list. This queue puts all newly generated successors at the end of the queue, ensuring that shallow nodes are expanded first. The BFS algorithm is presented in Figure 1.19.

Input:
- *s*: The initial state
- *successorsFn*: The function that generates the successor pairs (action, successor) for a given state
- *isGoal*: The function that checks if a state is a goal or not

Output: Goal node or None

```
Function BFS (s, successorsFn, isGoal)
Begin
  Open: queue /* FIFO list */
  Closed: list

  init_node <- Node (s, None, None) /* A data structure with (state, parentNode, action) attributes*/
  if (isGoal(init_node.state)) then return init_node
  Open.enqueue(init_node)
  Closed <- [ ]

  while (not Open.empty()) do
    current <- Open.dequeue() /* Choose the shallowest node in Open */
    Closed.add(current)
    for each (action, successor) in successorsFn(current.state) do
      child <- Node (successor, current, action) /* Create a new node and link it to its parent */
      if (isGoal(child.state)) then return child
      if (child.state not in Closed and not in Open) then
        Open.enqueue(child)

  return None
End
```

Figure 1.19 BFS algorithm

Example 2: Let G = (V, E) be a directed graph as shown in Figure 1.20. Find the possible path that the agent can take from the node **A** to the node **K** using the BFS algorithm.

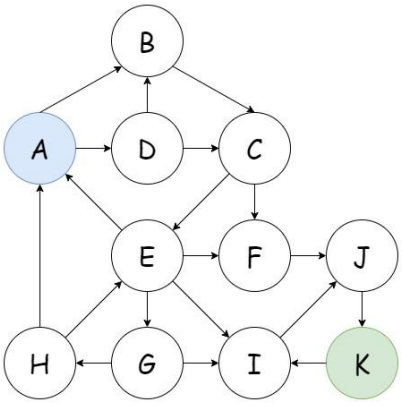


Figure 1.20 Graph of example 2

The BFS algorithm execution and the search tree are shown in Table 1.2 and Figure 1.21, respectively.

The path of the solution is: A → B → C → F → J → K.

Table 1. 2 BFS algorithm execution of example 2

Open	Closed	Description
(A)		
(B), (D)	(A)	
(D), (C)	(A), (B)	
(C)	(A), (B), (D)	State B is already in Closed State C is already in Open
(E), (F)	(A), (B), (D), (C)	
(F), (G), (I)	(A), (B), (D), (C), (E)	State A is already in Closed State F is already in Open
(G), (I), (J)	(A), (B), (D), (C), (E), (F)	
(I), (J), (H)	(A), (B), (D), (C), (E), (F), (G)	State I is already in Open
(J), (H)	(A), (B), (D), (C), (E), (F), (G), (I)	State J is already in Open
(H), (K)	(A), (B), (D), (C), (E), (F), (G), (I), (J)	Goal state K is reached

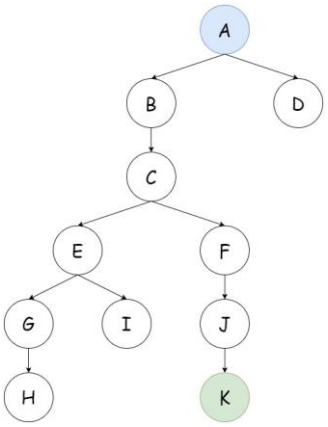


Figure 1. 21 BFS search tree of example 2

Properties:

- ✧ Is it complete? Yes, if a solution exists; that is, if the shallowest goal node is at some finite depth d.
- ✧ Is it optimal? Yes, if all actions have the same cost.
- ✧ Time complexity: If the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is: $b + b^2 + b^3 + \cdots + b^d = O(b^d)$.
- ✧ Space complexity: $O(b^d)$ since the maximum size of the Open queue is b^d.

1.3.3.2 Depth-first search (DFS)

This algorithm starts at the root node of the tree and explores as deep as possible before backtracking. The DFS algorithm uses a last-in-first-out (LIFO) list, known as a stack, for its Open list. As the nodes of deepest level are expanded, they are dropped from the stack, and the search then backs up to the next shallowest nodes that still have unexplored successors. The DFS algorithm is presented in the Figure 1.22.

Input:

- *s*: The initial state
- *successorsFn*: The function that generates the successor pairs (action, successor) from a state
- *isGoal*: as the function that checks if a state is a goal state or not

Output: Goal node or None

Function DFS (s, successorsFn, isGoal)

```
Begin
   Open: stack /* LIFO list */
   Closed: list

   init_node <- Node (s, None, None)
   if (isGoal(init_node.state)) then return init_node
   Open.push(init_node)
   Closed <- [ ]

   while (not Open.empty()) do
      current <- Open.pop() /* Choose the deepest node in Open */
      Closed.add(current)
      for each (action, successor) in successorsFn(current.state) do
         child <- Node (successor, current, action)
         if (isGoal(child.state)) then return child
         if (child.state not in Closed and not in Open) then
            Open.push(child)

   return None
End
```

Figure 1.22 DFS algorithm

Properties:

- ✧ Is it complete? Yes, if the state space is finite.
- ✧ Is it optimal? No, it always returns the leftmost solution, regardless of its cost.
- ✧ Time complexity: It Could potentially explore the entire state space. Therefore, if m (the maximum depth of the search tree) is finite, it takes $O(b^m)$ time.
- ✧ Space complexity: $O(bm)$ since we keep only the siblings in each level.

1.3.3.3 Depth-limited search (DLS):

The failure of DFS in infinite state spaces can be alleviated by supplying this algorithm with a predetermined depth limit l. That is, nodes at depth l are treated as if they have no successors. This approach is called depth-limited search and can solve the infinite-path problem. The DFS algorithm is presented in the Figure 1.23.

```
Input:
-  s: The initial state
-  successorsFn: The function that generates successor pairs (action, successor) from a state
-  isGoal: The function that checks if a state is a goal state or not
-  l: Depth limit (maximum depth to explore)

Output: Goal node or None
Function DLS (s, successorsFn, isGoal, l)
Begin
   Open: stack /* LIFO list */
```

```
   Closed: list

   init_node <- Node (s, None, None)
   init_node.d <- 0
   if (isGoal(init_node.state)) then return init_node
   Open.push(init_node)
   Closed <- [ ]

   while (not Open.empty()) do
      current <- Open.pop() /* Choose the deepest node in Open */
      if (current.d < l) then
         Closed.add(current)
         for each (action, successor) in successorsFn(current.state) do
            child <- Node (successor, current, action)
            child.d <- current.d + 1
            if (isGoal(child.state)) then return child
            if (child.state not in Closed and not in Open) then
               Open.push(child)

   return None /* No solution found within the depth limit */
End
```

Figure 1. 23 DLS algorithm

Example 3: Let G = (V, E) be a directed graph as shown in Figure 1.24. Find the possible path that the agent can take from the node **S** to the node **M** using the DLS algorithm. The depth limit is 3.

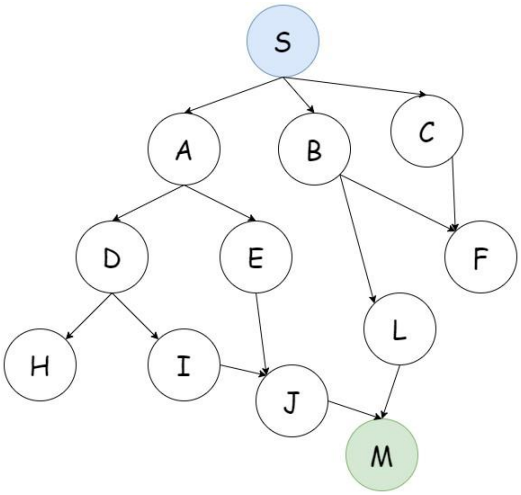


Figure 1. 24 Graph of example 3

The DLS algorithm execution and the search tree are shown in Table 1.3 and Figure 1.25, respectively. The path of the solution is: S → B → L → M.

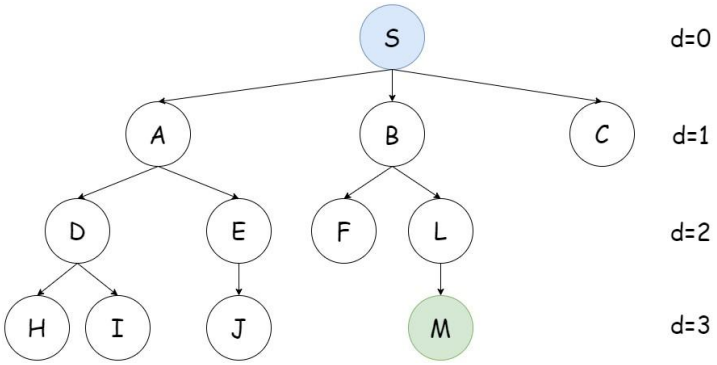


Figure 1. 25 DLS search tree of example 3

Table 1. 3 DLS algorithm execution of example 3

Open	Closed	Description
(S, 0)		
(A, 1), (B, 1), (C, 1)	(S, 0)	
(D, 2), (E, 2), (B, 1), (C, 1)	(S, 0), (A, 1)	
(H, 3), (I, 3), (E, 2), (B, 1), (C, 1)	(S, 0), (A, 1), (D, 2)	
(I, 3), (E, 2), (B, 1), (C, 1)	(S, 0), (A, 1), (D, 2)	Depth(H) ≥ limit => H is not expanded
(E, 2), (B, 1), (C, 1)	(S, 0), (A, 1), (D, 2)	Depth(I) ≥ limit => I is not expanded
(J, 3), (B, 1), (C, 1)	(S, 0), (A, 1), (D, 2), (E, 2)	
(B, 1), (C, 1)	(S, 0), (A, 1), (D, 2), (E, 2)	Depth(J) ≥ limit => J is not expanded
(F, 2), (L, 2), (C, 1)	(S, 0), (A, 1), (D, 2), (E, 2), (B, 1)	
(L, 2), (C, 1)	(S, 0), (A, 1), (D, 2), (E, 2), (B, 1), (F, 2)	F has no successors
(M, 3), (C, 1)	(S, 0), (A, 1), (D, 2), (E, 2), (B, 1), (F, 2), (L, 2)	Goal state M is reached

Properties:

- ✧ It's incomplete if the depth limit l is set such that $l < d$, where d is the depth of the shallowest goal node beyond the depth limit.
- ✧ Its time complexity is $O(b^l)$ and its space complexity is $O(bl)$.

1.3.3.4 Uniform-cost search (UCS)

UCS is an algorithm used to navigate a weighted search space with the goal of moving from a start node to a goal node while minimizing the cumulative cost. BFS is optimal when all step costs are equal because it always expands the shallowest unexpanded node. The UCS algorithm is a direct extension of the BFS, as it remains optimal with any step cost function. Instead of expanding the shallowest node, UCS expands the node with the lowest path cost. This is achieved through the use of a priority queue where nodes with lower costs are given higher priority. The UCS algorithm is detailed in Figure 1.26.

Input:
- *s*: The initial state
- *successorsFn*: The function that generates the successor pairs (action, successor) from a state
- *isGoal*: The function that checks if a state is a goal state or not

Output: Goal node or None

```
Function UCS (s, successorsFn, isGoal)
Begin
   Open: priorityQueue /* Ordered queue by path cost g */
   Closed: list

   init_node <- Node (s, None, None)
   init_node.g <- 0
   Open.insert(init_node)
   Closed <- [ ]

   while (not Open.empty()) do
      current <- Open.dequeue() /* Choose the node with the lowest cost g in the Open queue */
      if (isGoal(init_node.state)) then return current
      Closed.add(current)
      for each (action, successor) in successorsFn(current.state) do
         child <- Node (successor, current, action)
         child.g <- current.g + c(current, action, successor)
         if (child.state not in Open and not in Closed) then
            Open.insert(child)
         else if (child.state in Open with a higher value of g) then
                  replace that Open node with child
               else if (child.state in Closed with a higher value of g) then
                     remove that Closed node
                     Open.insert(child)

   return None
End
```

Figure 1.26 UCS algorithm

Example 4: Let G = (V, E) be a directed weighted graph as shown in Figure 1.27. Find the possible path that the agent can take from the node **S** to the node **G** using the UCS algorithm.

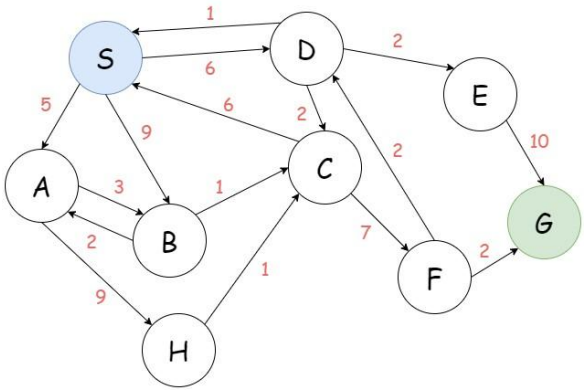


Figure 1. 27 Graph of example 4

The UCS algorithm execution and the search tree are shown in Table 1.4 and Figure 1.28, respectively. The resulting path is: S → D → C → F → G, with a path cost of 17.

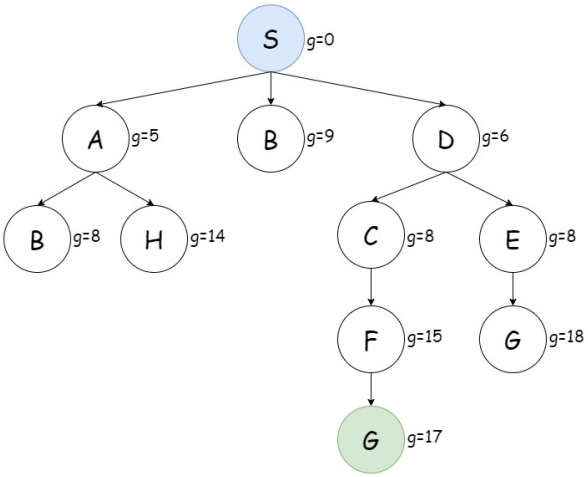


Figure 1. 28 UCS search tree of example 4

Table 1. 4 UCS algorithm execution of example 4

Open	**Closed**	**Description**
(S, 0)		
(A, 5), (B, 9), (D, 6)	(S, 0)	
~~(B, 9)~~, (D, 6), (B, 8), (H, 14)	(S, 0), (A, 5)	Replace (B, 9) by (B, 8) in Open
(B, 8), (H, 14), (C, 8), (E, 8)	(S, 0), (A, 5), (D, 6)	(S, 0) is in Closed => (S, 7) is not added to Open
(H, 14), (C, 8), (E, 8)	(S, 0), (A, 5), (D, 6), (B, 8)	(A, 5) is in Closed => (A, 10) is not added to Open (C, 8) is in Open => (C, 9) is not added to Open
(H, 14), (E, 8), (F, 15)	(S, 0), (A, 5), (D, 6), (B, 8), (C, 8)	(S, 0) is in Closed => (S, 14) is not added to Open
(H, 14), (F, 15), (G, 18)	(S, 0), (A, 5), (D, 6), (B, 8), (C, 8), (E, 8)	

(F, 15), (G, 18)	(S, 0), (A, 5), (D, 6), (B, 8), (C, 8), (E, 8), (H, 14)	(C, 8) is in Closed => (C, 15) is not added to Open
~~(G, 18)~~, (G, 17)	(S, 0), (A, 5), (D, 6), (B, 8), (C, 8), (E, 8), (H, 14), (F, 15)	(D, 6) is in Closed => (D, 17) is not added to Open
	(S, 0), (A, 5), (D, 6), (B, 8), (C, 8), (E, 8), (H, 14), (F, 15)	Goal state is reached: (G, 17)

Properties:

- ✧ Is it complete? Yes, if a solution exists, meaning the lowest-cost goal node is at some finite depth.
- ✧ Is it optimal? Yes, if all step costs are non-negative, ensuring that paths never get shorter as nodes are added.
- ✧ Time complexity and space complexity: let C^* be the cost of the optimal solution, and assume that every action costs at least ϵ. Then the algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, which can be much greater than b^d.

1.3.4 Informed search algorithms

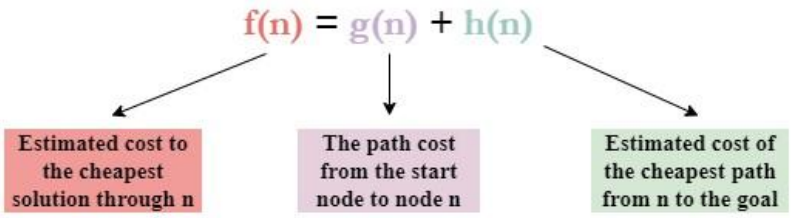
The informed search strategies utilize problem-specific knowledge to find solutions more efficiently than the blind search strategies. Often there is an extra knowledge that can be used to guide the search towards the goal. This problem-specific knowledge is typically referred to as a **heuristic**.

A **heuristic** is defined as:

- A function *h(n)* that estimates how close a state *n* is to a goal.
- Designed for a particular search problem.
- For example: for path finding, heuristics such as Euclidean distance or Manhattan distance can be used to estimate the proximity to the goal state.

1.3.4.1 A* search

The most widely known informed search algorithm is called A* search algorithm. It evaluates nodes by combining *g(n)*, the cost to reach node *n*, and *h(n)*, the cost to get from node *n* to the goal. This sum is referred to as the evaluation function *f(n)*.



To find the cheapest solution, the algorithm prioritizes nodes with the lowest value of *f(n)*, which is computed as *g(n)+h(n)*. The algorithm is similar to UCS, with the key difference being that A* uses *g(n)+h(n)* instead of just *g(n)*. The pseudocode of the A* search algorithm is provided in Figure 1.29.

Input:

- *s*: The initial state
- *successorsFn*: The function that generates the successor pairs (action, successor) from a state

- *isGoal*: The function that checks if a state is a goal state or not
- *h*: The heuristic function for a node *n*

Output: Goal node or None

```
Function AStar (s, successorsFn, isGoal, h)
Begin
  Open: priorityQueue /* Ordered queue by f */
  Closed: list

  init_node <- Node (s, None, None)
  init_node.g <- 0
  init_node.f <- h(init_node)
  Open.insert(init_node)
  Closed <- [ ]

  while (not Open.empty()) do
    current <- Open.dequeue() /* Choose the node with the lowest cost f in the Open queue */
    if (isGoal(current.state)) then return current
    Closed.add(current)
    for each (action, successor) in successorsFn(current.state) do
      child <- Node (successor, current, action)
      child.g <- current.g + c(current, action, successor)
      child.f <- child.g + h(child)
      if (child.state not in Open and not in Closed) then
        Open.insert(child)
      else if (child.state in Open with a higher value of f) then
            replace that Open node with child
          else if (child.state in Closed with a higher value of f) then
                remove that Closed node
                Open.insert(child)

  return None
End
```

Figure 1.29 A⋆ algorithm

Example 5: Let G = (V, E) be a weighted graph as shown in Figure 1.30. Find the possible path that the agent can take from the node **S** to the node **G** using the A* algorithm.

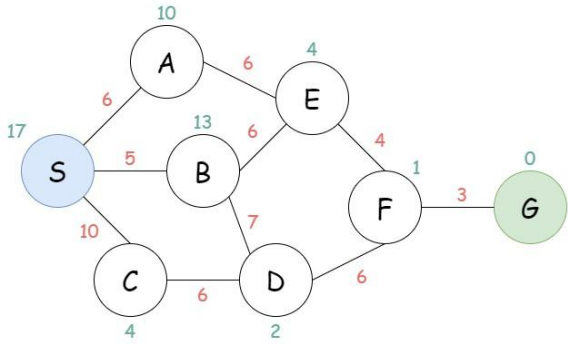


Figure 1. 30 Graph of example 5

The A* algorithm execution and the search tree are shown in Table 1.5 and Figure 1.31, respectively. The resulting path is: S → B → E → F → G, with a path cost of 18.

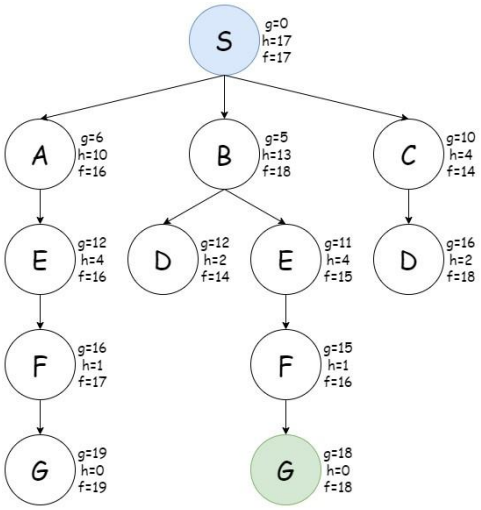


Figure 1. 31 A⋆ search tree of example 5

Table 1. 5 A⋆ algorithm execution of example 5

Open	**Closed**	**Description**
(S, 0, 17)		
(A, 6, 16), (C, 10, 14), (B, 5, 18)	(S, 0, 17)	
(A, 6, 16), (B, 5, 18), (D, 16, 18)	(S, 0, 17), (C, 10, 14)	(S, 0, 17) is in Closed => (S, 20, 37) is not added to Open
(B, 5, 18), (D, 16, 18), (E, 12, 16)	(S, 0, 17), (C, 10, 14), (A, 6, 16)	(S, 0, 17) is in Closed => (S, 12, 29) is not added to Open
(B, 5, 18), (D, 16, 18), (F, 16, 17)	(S, 0, 17), (C, 10, 14), (A, 6, 16), (E, 12, 16)	(A, 6, 16) is in Closed => (A, 18, 28) is not added to Open (B, 5, 18) is in Open => (B, 18, 31) is not added to Open
(B, 5, 18), (D, 16, 18), (G, 19, 19)	(S, 0, 17), (C, 10, 14), (A, 6, 16), (E, 12, 16), (F, 16, 17)	(D, 16, 18) is in Open => (D, 22, 24) is not added to Open (E, 12, 16) is in Closed => (E, 20, 24) is not added to Open
~~(D, 16, 18)~~, (G, 19, 19), (D, 12,	(S, 0, 17), (C, 10, 14), (A, 6, 16),	(S, 0, 17) is in Closed => (S, 10,

14), (E, 11, 15)	~~(E, 12, 16)~~, (F, 16, 17), (B, 5, 18)	27) is not added to Open Replace (D, 16, 18) by (D, 12, 14) in Open Remove (E, 12, 16) from Closed and add (E, 11, 15) to Open
(G, 19, 19), (E, 11, 15)	(S, 0, 17), (C, 10, 14), (A, 6, 16), (F, 16, 17), (B, 5, 18), (D, 12, 14)	(B, 5, 18) is in Closed => (B, 19, 32) is not added to Open (C, 10, 14) is in Closed => (C, 18, 22) is not added to Open (F, 16, 17) is in Closed => (F, 18, 19) is not added to Open
(G, 19, 19), (F, 15, 16)	(S, 0, 17), (C, 10, 14), (A, 6, 16), ~~(F, 16, 17)~~, (B, 5, 18), (D, 12, 14), (E, 11, 15)	(A, 6, 16) is in Closed => (A, 17, 27) is not added to Open (B, 5, 18) is in Closed => (B, 17, 30) is not added to Open Remove (F, 16, 17) from Closed and add (F, 15, 16) to Open
~~(G, 19, 19)~~, (G, 18, 18)	(S, 0, 17), (C, 10, 14), (A, 6, 16), (B, 5, 18), (D, 12, 14), (E, 11, 15), (F, 15, 16)	(D, 12, 14) is in Closed => (D, 21, 23) is not added to Open (E, 11, 15) is in Closed => (E, 19, 23) is not added to Open
	(S, 0, 17), (C, 10, 14), (A, 6, 16), (B, 5, 18), (D, 12, 14), (E, 11, 15), (F, 15, 16)	Goal state is reached: (G, 18, 18)

Properties:

1- Conditions for optimality: Admissibility and consistency:

✧ A heuristic h is **admissible** (also known as **optimistic**) if it never overestimates the cost to reach the goal, i.e., $h(n) \leq h^*(n)$, where $h^*(n)$ is the true or optimal cost from state n to the goal. An example of an admissible heuristic is the straight-line distance h_{SLD}. It's admissible because the shortest path between any two points is a straight line, so the straight-line distance cannot be an overestimate. In the example shown in Figure 1.32, the heuristic h is admissible because it satisfies the condition $h(n) \leq h^*(n)$ for both nodes n and n'.

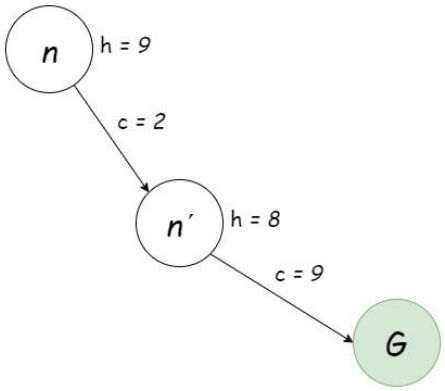


Figure 1.32 Example of an admissible heuristic

✧ A heuristic h is considered **consistent** (also known as **monotonic**) if, for every successor n' of n generated by any action a with a step cost $c(n, a, n')$, it holds that $h(n) \leq h(n') + c(n, a, n')$. In the example shown in Figure 1.33, h is consistent since $9 \leq 2 + 8$.

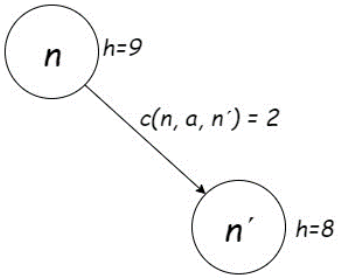


Figure 1. 33 Example of a consistent heuristic

✧ If a heuristic h is consistent, then h is admissible.

2- Optimality of A*:

A* has the following properties: the tree-search variant of A* guarantees optimality when $h(n)$ is admissible, whereas the graph-search variant ensures optimality when $h(n)$ is not just admissible, but consistent. We focus on demonstrating the second claim, as it represents a more general and robust criterion for optimality.

✧ The first step is to prove the following: If $h(n)$ is consistent, then the values of $f(n)$ are non-decreasing along any path.

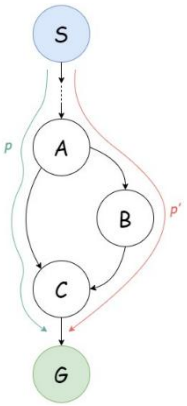
Proof

Suppose n' is a successor of n; then, by the definition of the A* search algorithm, we have:
$g(n') = g(n) + c(n, a, n')$ for some action a.
Additionally, we know that: $f(n)$ is the estimated total cost for node n in the A* algorithm, calculated as: $f(n) = g(n)+h(n)$, where $h(n)$ is the heuristic cost estimate from node n to the goal.
There for node n', we can calculate its $f(n')$ value as follows:
$f(n')=g(n')+h(n')=g(n)+c(n, a, n')+h(n')$.
Since $h(n)$ is a consistent heuristic (which means $c(n, a, n')+h(n') \geq h(n)$) , we can conclude:
$f(n')=g(n)+c(n, a, n')+h(n') \geq g(n)+h(n)=f(n)$.

✧ The next step is to prove that: A* is optimal if $h(n)$ is consistent.

Proof

A* selects a path p, which we assume to be the shortest path because A* is optimal. To derive a contradiction, let's assume that there exists another path p' that is actually shorter than p.



Now, consider the first state, C, along path p that differs between p and p'. Just before the state C is chosen from the Open list to be expanded, node B will also be on the Open list.
Since node C was expanded before B, we have $f(C) \leq f(B)$ because A* selects the node with the cheapest cost at each step … (1)
On the other hand, because the heuristic h is consistent, the values of f are non-decreasing along any path, including the path p'. Therefore, $f(B) \leq f(C)$ along the path p'. Since B is on the path to C, then $f(B) < f(C)$ … (2)
From (1) and (2), we have a contradiction.

1.3.5 Other search algorithms

Other blind and informed search algorithms are shown in Figure 1.34.

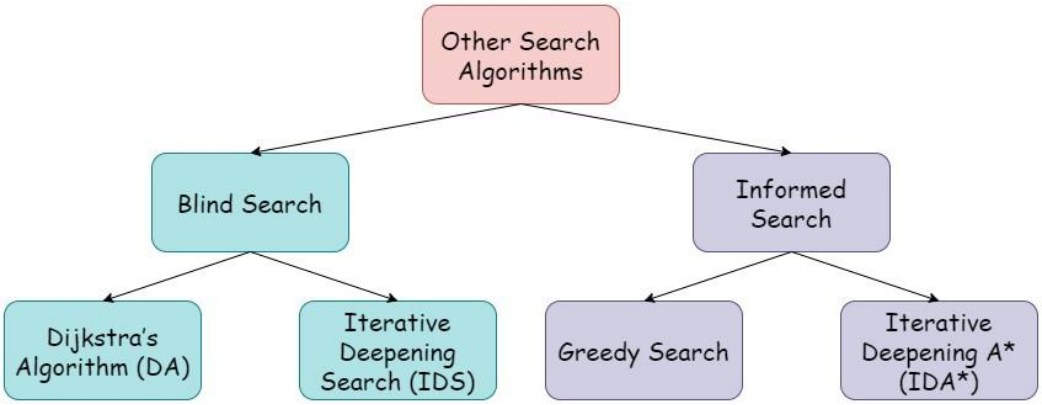


Figure 1.34 Other search algorithms

1.3.5.1 Dijkstra's Algorithm (DA)

Dijkstra's algorithm is almost identical to UCS, except that it is used to find the shortest path from a single source node to all other nodes in the graph. It does stop until it has found the shortest path to every reachable node, whereas UCS stops as soon as a goal node is found. DA is typically employed in scenarios like network routing or road navigation systems, while in the field of artificial intelligence, UCS is commonly favored.

1.3.5.2 Iterative Deepening Search (IDS)

IDS is an iterative graph searching strategy that takes advantage of the completeness of the BFS strategy but uses much less memory in each iteration (similar to DLS). IDS gradually increases the depth of the search while still leveraging the efficiency of DFS. Here's how it works:

1. IDS starts with a very shallow search depth limit.
2. It performs a depth-limited search, exploring nodes at the current depth limit.
3. If the goal state is not found at the current depth limit, IDS increments the depth limit by 1 and starts another depth-limited search from the root node. It repeats this process iteratively, gradually increasing the depth limit with each iteration.

The difference between the DFS algorithm and IDS is shown in the example below (Figure 1.35 and Figure 1.36).

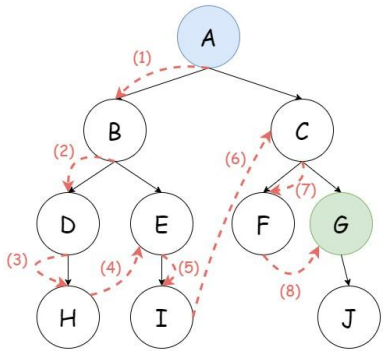


Figure 1. 35 Execution of DFS

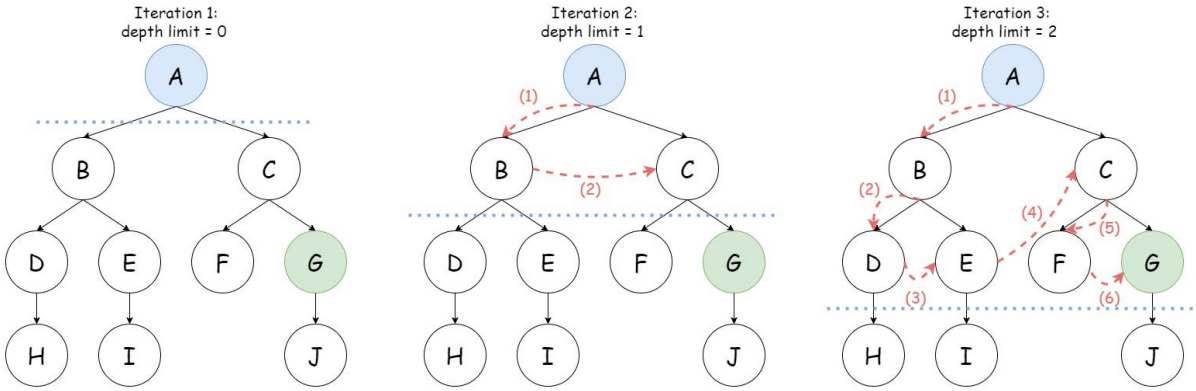


Figure 1.36 Execution of IDS

1.3.5.3 Greedy search

Greedy search, also known as greedy best-first search (GBFS), uses a heuristic to estimate the cost from the current node to the goal. Greedy search is solely driven by this heuristic, with the aim of selecting the node that appears to be closest to the goal based on the heuristic alone.

1.3.5.4 Iterative deepening A*

The simplest way to reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative deepening A* (IDA*) algorithm. IDA* employs a depth-limited version of the A* algorithm.

1. It starts with a threshold set to the heuristic estimate of the cost from the start node to the goal: *threshold=h(init_node).*
2. It performs a depth-limited A* search, expanding a node *n* if: *f(n)=g(n)+h(n) <= threshold*. Save the *f'=(*smallest *f(n) > threshold).*
3. If the goal state is not found at the current threshold, *threshold= f'*, and repeat IDA*