Chapter 3

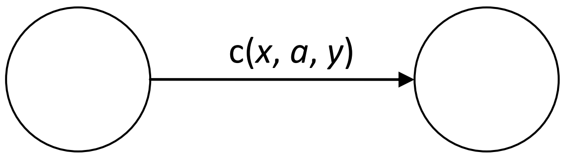
A problem-solving agent is a kind of **goal-based agent** . It decide what to do by finding sequences of actions that lead to desirable states.

A **problem** can be defined by four components :

1. **The initial state** that the agent starts in.
2. Possible **actions**. The common uses a **successor function***,* SUCCESSOR-FN(*x*).
3. **The goal test,** which determines whether a given state is a goal state.
4. **A path cost** is a function that assigns a numeric cost to each path.

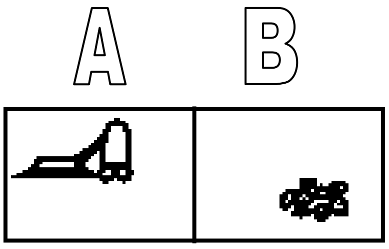
- A **solution** to a problem is a path from the initial state to a goal state.
- A **path** in the state space is a sequence of states connected by a sequence of actions.
- Solution quality is measured by **path cost function**.
- The **optimal solution** has the lowest path cost among all solutions.

- The **state space** is the set of all states reachable from the initial state.
- c(*x*, *a*, *y*) is **step cost** of taking action *a* to go from state *x* to state *y* .



Examples:

1. Vacuum World

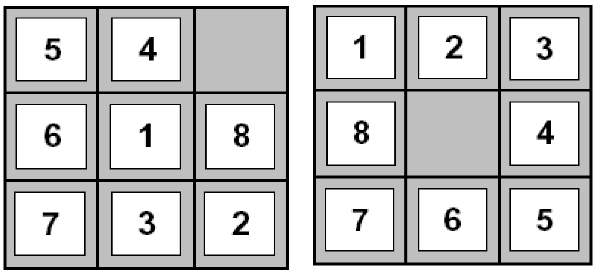


The formulation:

- **<u>States:</u>** the agent can be in one of two locations, each might or might not contain dirt. (2×2^2 = 8 possible states, for ***n*** locations → ***n***$\times2^n$ states)
- **<u>Initial state:</u>** any state can be considered an initial state
- **<u>Successor function:</u>** generates the legal states that result from trying the three actions (Left, Right, and Suck)
- **<u>Goal test:</u>** checks whether all the squares are clean
- **<u>Path cost:</u>** number of steps in the path (each step costs 1)

2. Eight -puzzle

- **<u>States</u>:** a state description specifies the location of each of the 8 tiles and the blank in one of the 9 squares.
- **<u>Initial state:</u>** any state can be considered an initial state
- **<u>Successor function:</u>** generates the legal states that result from trying the four actions (Blank move Left, Right, Up or Down)
- **<u>Goal test:</u>** checks whether all the state matches the goal configuration
- **<u>Path cost:</u>** number of steps in path (each step costs 1)

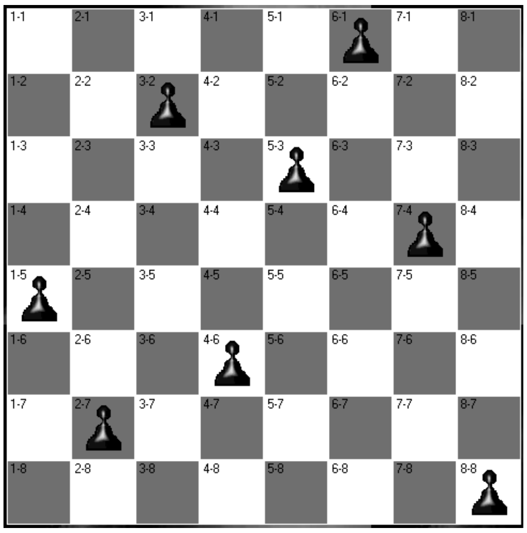


Start State

3. **Eight queens**

The incremental formulation :

- **<u>States</u>:** any arrangement of 0 to 8 queens on board.
- **<u>Initial state</u>:** No queens on the board.
- **<u>Successor Function</u>:** add queens to any empty square.
- **<u>Goal test:</u>** 8 queens on board, none attacked.



(A Solution to Eight Queen Puzzle)

Touring problem

Visit all cities at least once, starting and ending in same city.

Traveling salesperson problems (TSP)

A touring problem in which each city must be visited exactly once. The aim is to find the *shortest* tour.

VLSI layout (Very Large-Scale Integrated circuits)

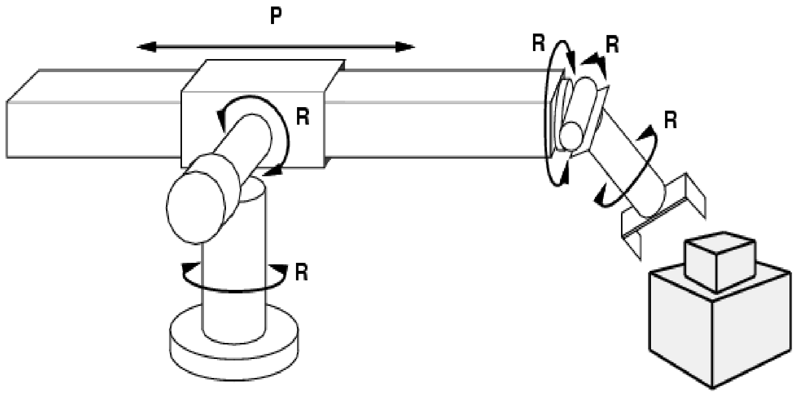
Two tasks are **cell layout** and **channel routing.**

Airline travel problem (example of Route finding)

States : each is represented by a location (airport) and the current time
Initial state : specified by the problem
Successor function : the states resulting from taking any scheduled flight, leaving later than the current time.
Goal test : are we at the destination on time**.**
Path cost : depends on monetary cost, waiting time, etc.

Robot navigation is a generalization of the **route-finding problem**.
Assembly sequencing
to find an order in which to assemble the parts of some object.

- **states**: real-valued coordinates of robot joint angles parts of the object to be assembled
- **actions**: continuous motions of robot joints
- **goal test:** complete assembly
- **path cost:** time to execute



Searching For Solutions

```
function PROBLEM-SOLVING-AGENT(percept) returns an action
    inputs: percept, a percept
    static: solution, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)

    if solution is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        solution ← SEARCH( problem)

    action ← FIRST(solution)
    return action
```

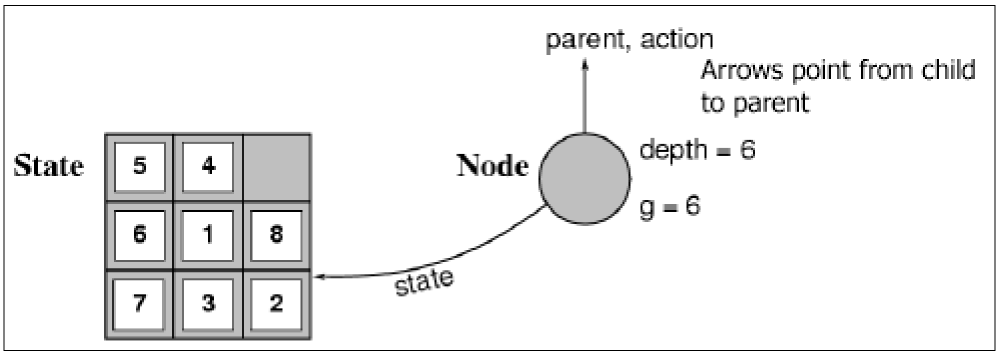
```
function GENERAL-SEARCH(problem) returns solution, or failure

initialize the search tree using the initial state of problem
loop do
        if there are no candidates for expansion
                then return failure

        choose a leaf node for expansion according to strategy

        if the node contains a goal
                then return the corresponding solution
        else
                expand the node and add resulting nodes to the search tree
end
```

The **root** of tree is initial state.
The **leaf** nodes do not have successors.

states vs. nodes

A **state** is a (representation of) a physical configuration within the problem state-space.

While a **node** is a (bookkeeping) data structure constituting part of a search tree includes state, parent node, action, path cost , depth



State: the state to which node corresponds to
Parent-Node: node in search tree that generated this node
Action: action applied to the parent to generate this node
Path-cost: g(x),of the path from initial state to the node
Depth: number of steps from initial state to this

A *collection of nodes that are generated but not yet expanded* is called the **fringe**, each element of fringe is a leaf node (i.e. with no successors in tree)

Uninformed search strategies

Uninformed search strategies (**blind search**) use only information available in the problem definition

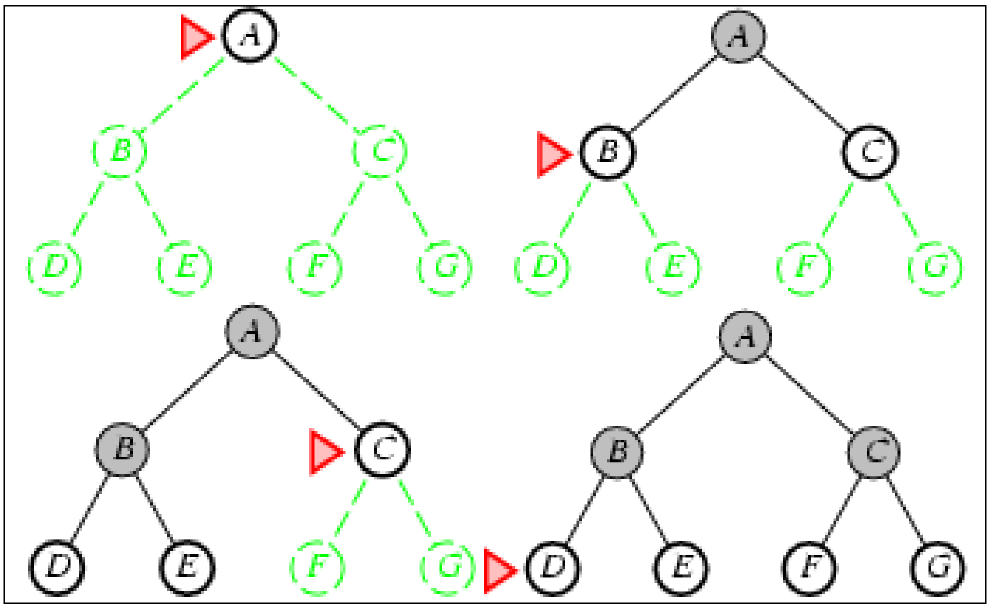
- Breadth-first search(BFS)
- Uniform-cost search(UCS)
- Depth-first search(DFS)
- Depth-limited search(DLS)
- Iterative deepening search(IDS)
- Bidirectional

The tree trace begin with the root which is always the initial, and continues expanding with the following colors.

white: unvisited
gray : expanded
black: dead

1- Breadth-first Search (BFS)

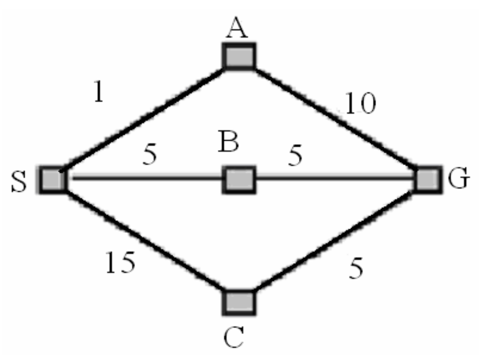
- ☒ Expand **shallowest** unexpanded node.
- ☒ Fringe is a **FIFO** queue, i.e., new successors go at end
- ☒ **Stack trace** []➔[A]➔[BC]➔[CDE]➔[DEFG]➔[EFG]➔[FG]➔[G]➔[]
- ☒ **tree trace:**

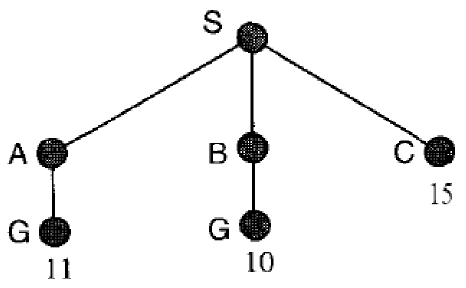


2- Uniform Cost Search (UCS) or Dijkstra's algorithm

- ☒ Uniform cost search is BFS by expanding the **lowest-cost**.

$$\boxed{g(\text{Successor}(n)) \geq g(n)}$$

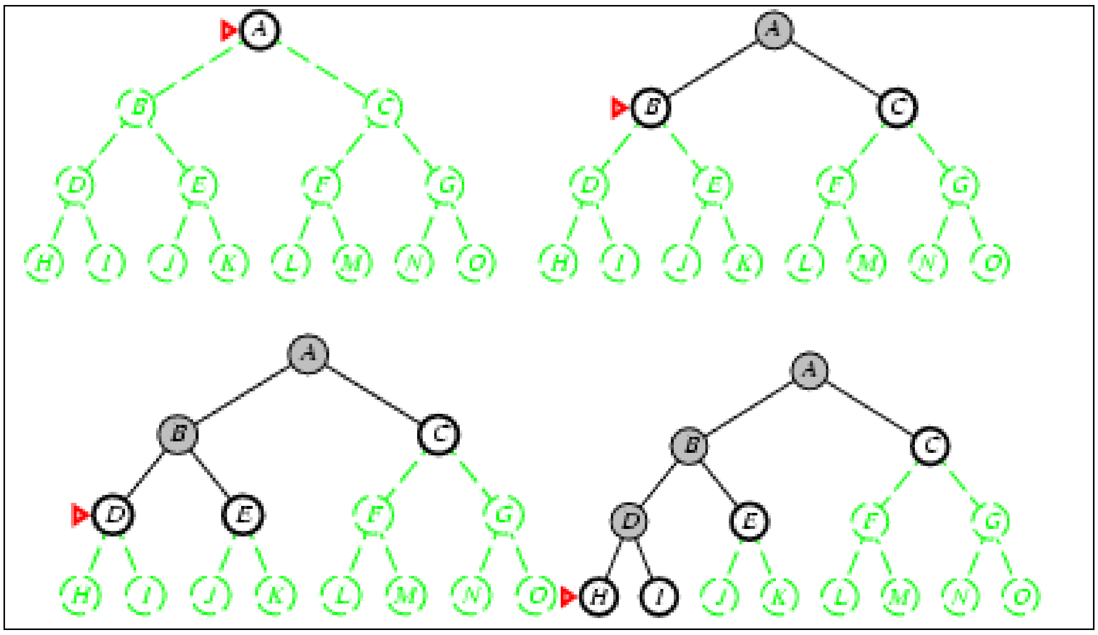


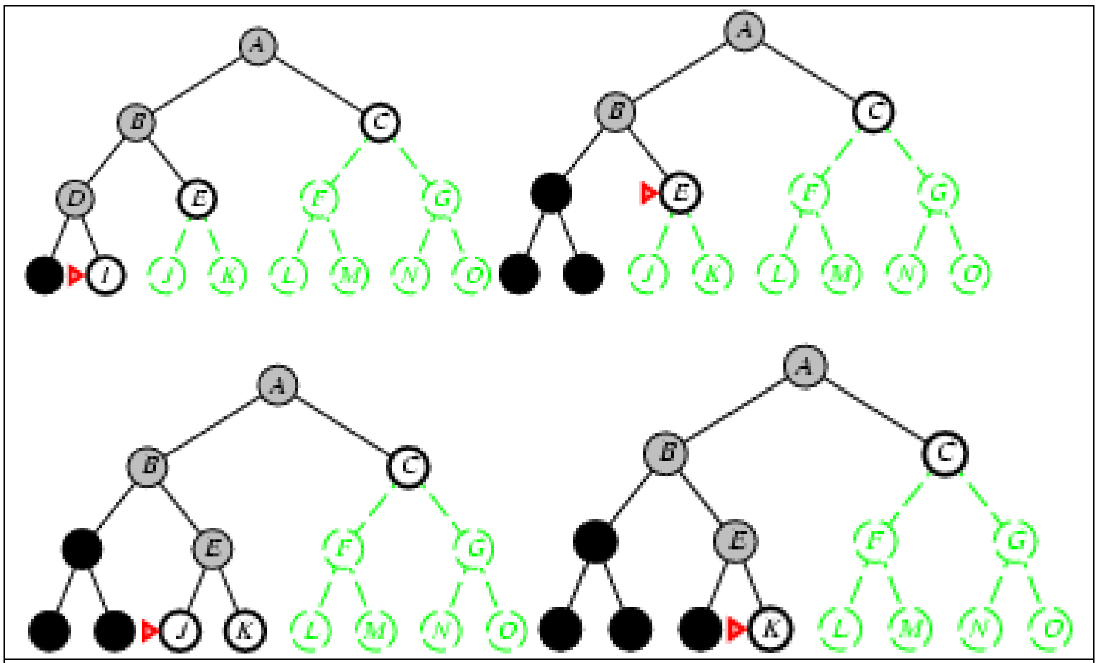


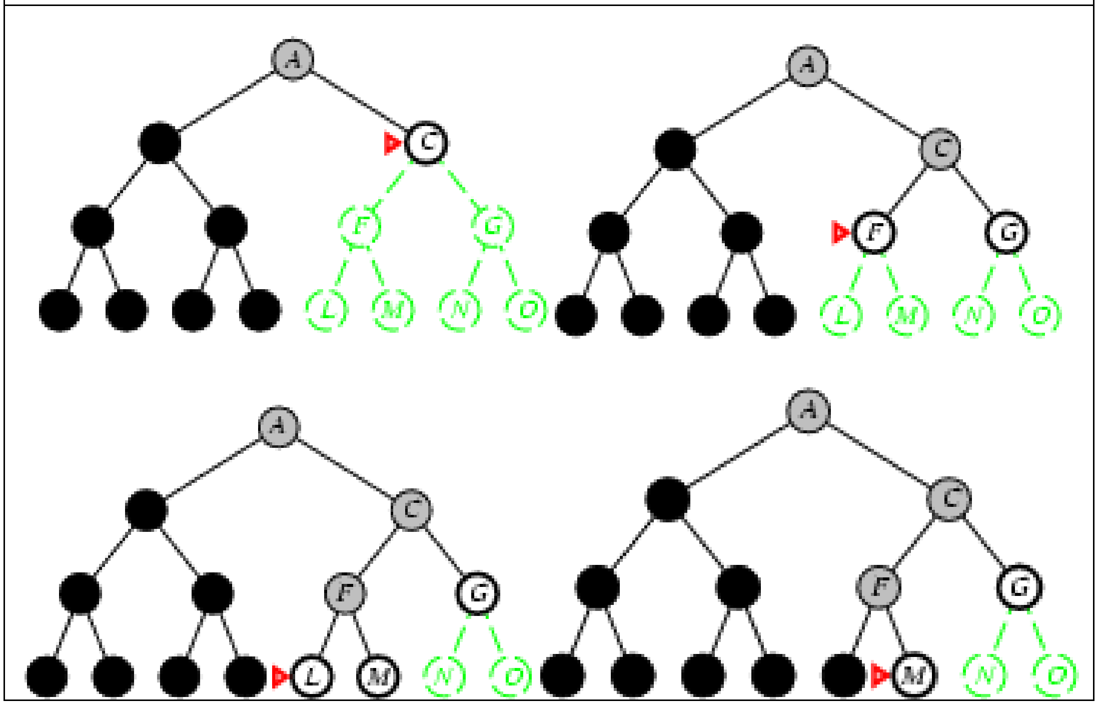
3- Depth-First Search (DFS)

- ☒ Always expands the **deepest** node in search tree.
- ☒ **Modest memory requirements**.
- ☒ fringe = **LIFO** queue (or **stack**), i.e., put successors at front
- ☒ **stack trace**:

  []➔ [A]➔ [BC]➔[DEC] ➔[HIEC]➔ [IEC]➔
[EC]➔[JKC]➔[KC]➔[C]➔[FG]➔[LMG]➔[MG]➔[G]




A
B
C
D
E
F
G
I
J
K
L
M
N
O


A
C
F
G
L
M
N
O

transitive closure using Warshall

$$W_0 = M_R = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$, n =4

$$W1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

k =1, W0 has 1's in location (2,$\underline{1}$), ($\underline{1}$,2)
$\rightarrow$ put 1 in (2,2)

$$W2 = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

k=2, check row2, column2
W1 has 1 in locations 1,2 of column 2, 1,2,3 of row 2
$\rightarrow$ put 1 in 1,1 , 1,2 , 1,3 , 2,1 , 2,2 , 2,3

$$W3 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Column 3 in has 1 in locations 1,2, and row 3 has 1 in location 4
$\rightarrow$ put 1 in 1,4 , 2,4

$M_R n$ = $M_R 2$ if n is even and $M_R n$ = $M_R 3$ if n is odd greater than 1

$$M_R \infty = W4 = W3 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Aspect	Breadth-First	Depth-First
Memory Usage	Require more memory (the whole tree is stored)	Less memory (Only one path is stored)
Finding a goal state	Takes more time especially if there are many goal states and the tree is limited	Faster if successors are ordered so, the goal state is placed to the left path
Possibility of getting trapped	Does not get trapped in trees where the left path can have infinite successors	Can get trapped in trees representing problems like the water-jag search space
Solution finder	If there is a solution, it is guaranteed to find it	There is no guarantee to find the solution

4. Depth-limited search(DLS)

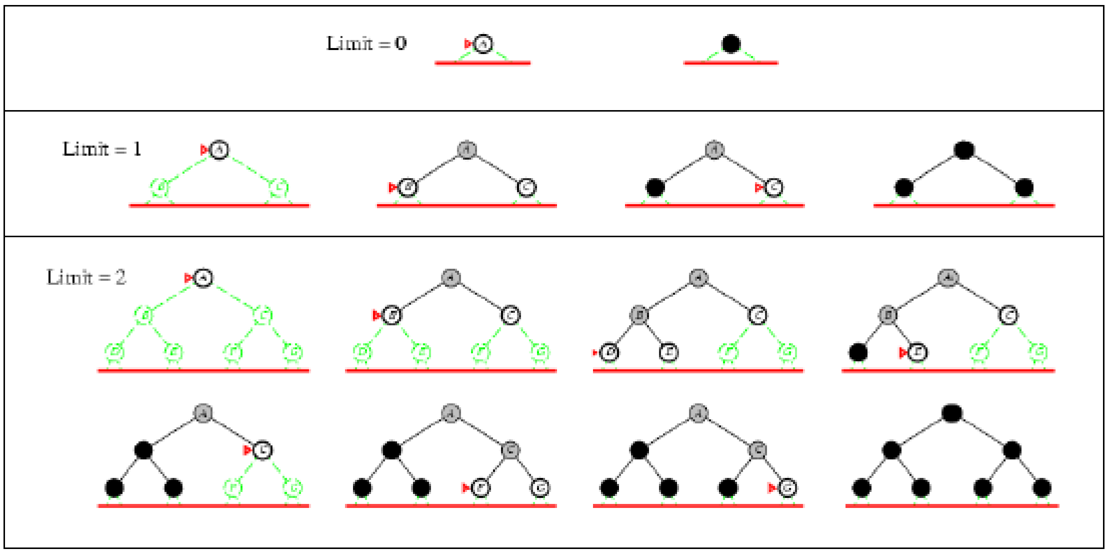
Depth-limited search (DLS) avoids the pitfalls of depth-first search by imposing a cutoff on the maximum depth of a path.

5. Iterative deepening search (IDS)

choosing the best depth limit by trying all possible depth limits: first depth 0, then depth 1, then depth 2, and so on.

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns
                                        a solution or failure
    inputs: problem , a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```



6-Bidirectional search

The idea behind bidirectional search is to simultaneously search both forward from the initial state and backward from the goal, and stop when the two searches meet in the middle. It is **2 BFS**.

Measuring problem- solving performance

A search strategy is defined by the **order of node expansion**.

Search strategies are evaluated based on four criteria:

- **Completeness**: is the strategy guaranteed to find a solution?
- **Optimality**: does the strategy find the highest quality solution when there are several different solution?
- **Time Complexity**: how long does it take to find a solution?
- **Space Complexity**: how much memory does it need to perform the search?

Time and space complexity are expressed in terms of 3 quantities:

- **b**: maximum branching factor of the search tree (or maximum number of successors of any node)
- **d**: depth of the least-cost solution (shallowest goal node)
- **m**: maximum depth (i.e. maximum length of any path) in state space (may be ∞)

Summary (Comparing algorithms)

Criterion	BFS	UCS	DFS	DLS	IDS	Bidirectional
Complete	Yes (a)	Yes(a, b)	No	No	Yes (a)	Yes (a, d)
Time	$O(b^{d+1})$	$O(b^{c^*})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{c^*})$	O(bm)	O(bl)	O(bd)	$O(b^{d/2})$
Optimal	Yes (c)	Yes	No	No	Yes(c)	Yes (c, d)

Conditions

a : complete if b is finite

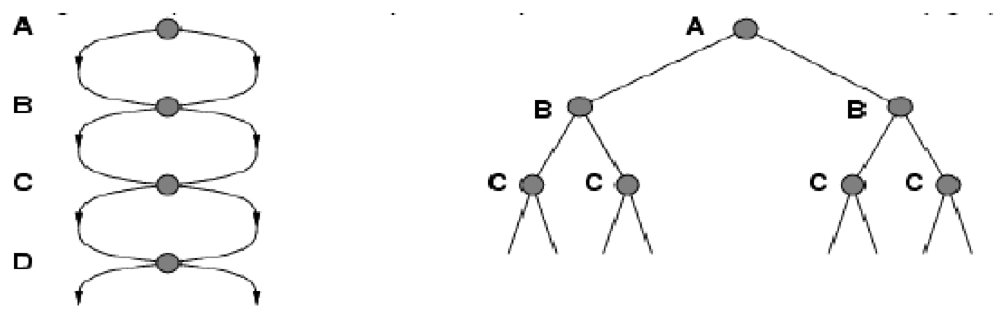
b : complete if costs ≥ 1

c : optimal if step costs are all identical

d ***:*** if both directions use BFS

Avoiding Repeated states

state space of size d (left) becomes a tree with 2^{d-1} leaves (right)



Searching with partial information

What happens when knowledge of the states or actions is incomplete ? We find that different types of incompleteness lead to three distinct problem types :

1. **sensor less problems (conformant problems)**

   If the agent has *no sensors at all*. Then it could be in one of several possible initial states. And each action might therefore lead to one of several possible successor states.

2. **contingency problems**

   If the environment is *partially observable or if actions are uncertain*, then the agent's percepts provides new information after each action. Each possible percept defines a contingency that must be planned for.

   The problem is called **adversarial** if the uncertainty is caused by the actions of another agent.

3. **exploration problems**

   When the states and *actions of the environment are unknown*, the agent must act to discover them.

   Can be considered an extreme case of contingency problems.