Solving Problems by Searching

Describe a kind of ***goal-based*** **problem-solving agents** which decide what to do by finding sequences of actions that achieve goals.

Problem-Solving Agents

When a problem-solving agent adopts a **goal** and aims to satisfy it, it has the greatest chance to maximize the performance measure.

◆ Problem-Solving Steps:

1. **Goal formulation :** selecting a desirable goal.

   - A goal is set of ***acceptable world states*** -- just those states in which the goal is satisfied.
   - Such goals help organizing behavior by limiting the objectives that the agent probably tries to do.

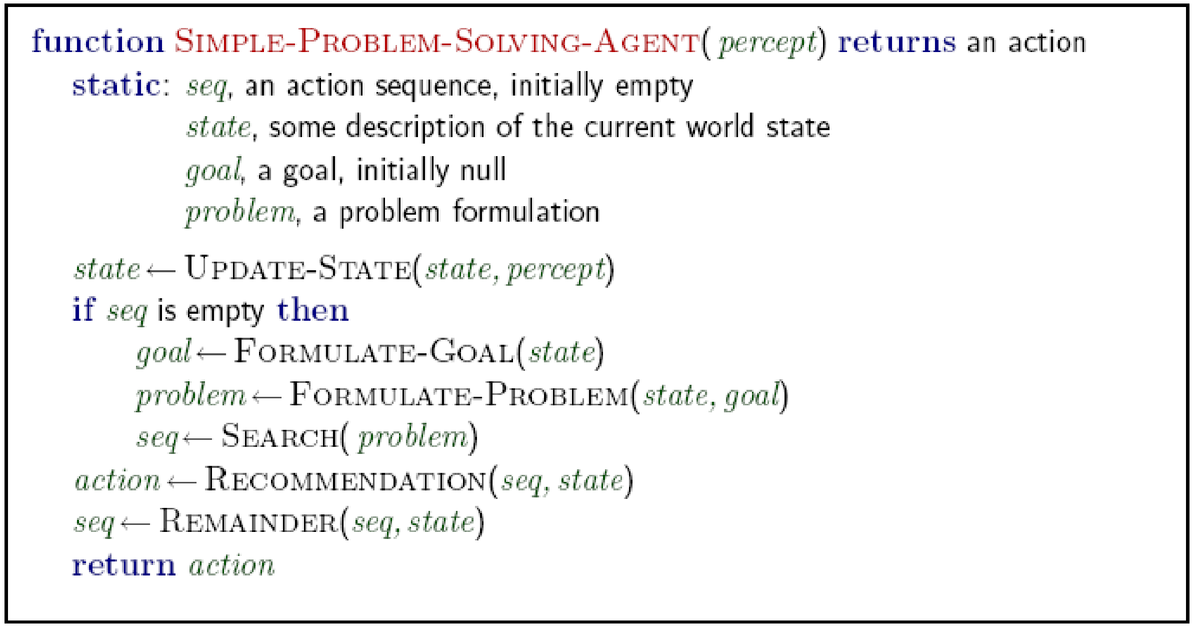
2. **Problem formation :** deciding what actions and states to consider.

   - Additional knowledge may be needed for the agent to formulate the immediate world states and its possible actions to reach those states.

3. **Search :** examining different possible ***sequences*** of actions and choosing the best one.

- A search algorithm takes a ***problem*** as **input** and returns a **solution** in the form of ***an action sequence***.

4. **Execute solution :** carrying out the recommended actions



```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
    static: seq, 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 seq is empty then
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH( problem)
    action ← RECOMMENDATION(seq, state)
    seq ← REMAINDER(seq, state)
    return action
```

Formulating Problems

How an agent is connected to its environment through its **percepts** and **actions** will affect the different amounts of ***knowledge*** that the agent can have concerning its actions and the possible states.

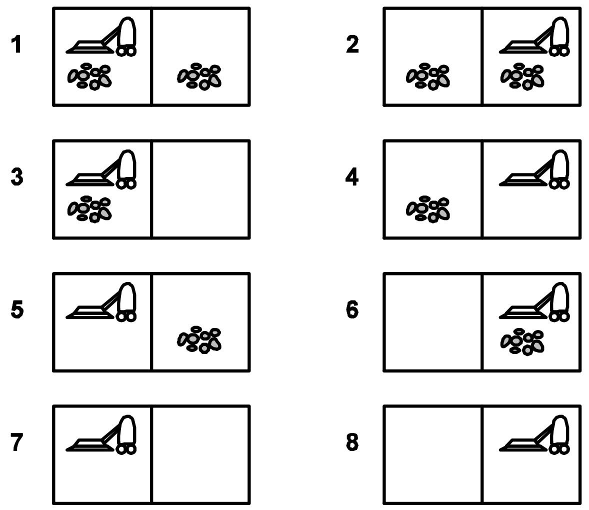
- There are four essentially different types of problems -- ***single state problems***, ***multiple state problems***, ***contingency problems***, and ***exploration problems***.

Knowledge and problem types

Consider a simplified vacuum world with two locations. Each location may or may not contain dirt and the agent may be in one location or the other. There are 8 possible world states as in

※**Single state problems:** the state is always known with certainty.

- Agents must be able to:
  1. clearly identify the ***current state*** (i.e. the world is accessible).
  2. exactly know the ***state transitions*** caused by calculated ***actions***.



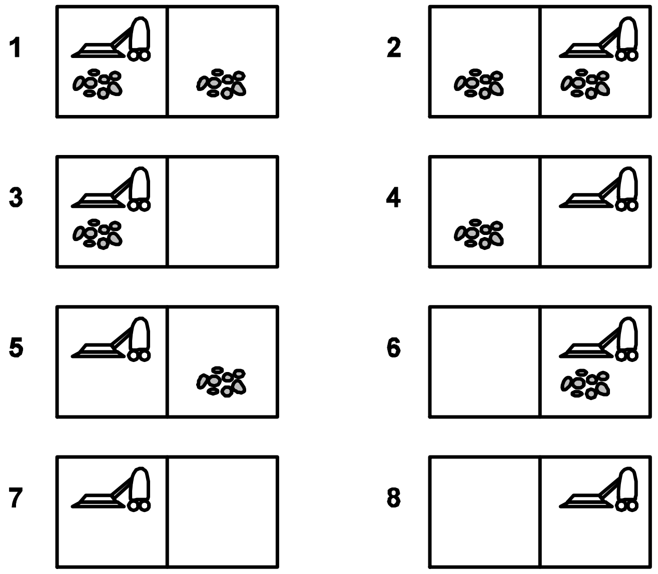
- For example:

initial state 5
⇨ [Right, Suck] (calculated action)
⇨ goal state 8.

※**Multiple-state problems:** agents know which states they might be in.

- Agents may :

  1. have limited access to the world state.
  2. know exactly how their actions will change the world states.



- For example:

  initial state in {1,2,3,4,5,6,7,8}
  ⇨ [Right,Suck,Left,Suck]
  ⇨ goal state {7}.

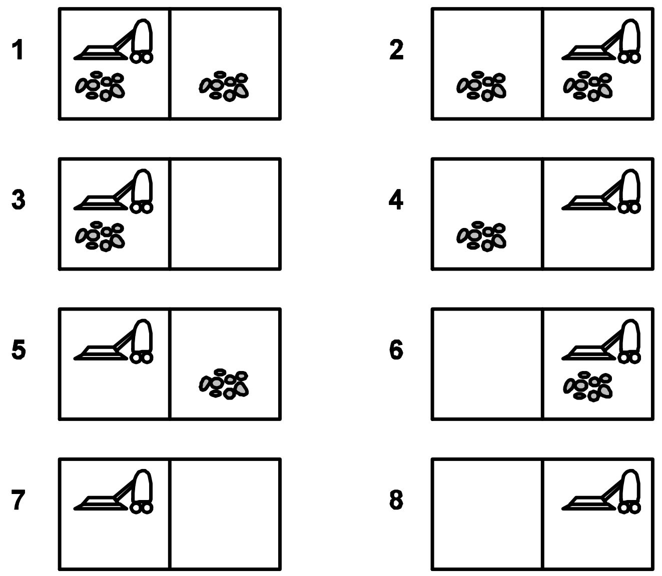
※ **Contingency problems:** constructed plans with conditional parts based on sensors.

- Environment may be ***nondeterministic***. For example, it obeys **Murphy‘s Law** :

  > *Suck* action *sometimes* deposits dirt on the carpet *but only if there is no dirt there originally*.

- Agents may :
  1. have limited access to the world state.
  2. do not know exactly how their actions will change the world states.

- For example: an agent in the Murphy‘s world can sense it’s current position (***note, not the state***) and local dirt, but not dirt in the next square.



```
initial state in {1,3}
⇨    [Suck, Right, Suck]
       ⇩       ⇩       ⇩
     {5, 7}    ⇩       ⇩
      ↳⇨ {6, 8}        ⇩
              ⇩  ↳⇨⇨fail
              ↳⇨⇨⇨success
```

- To solve the problem, agents must ***sense during the execution phase***, i.e. **interleave** search and execution(e.g. two-player games).

[Suck, Right] ⇨ sense ⇨ [Suck] or [ ]

※**Exploration problems**: agent must learn the effect of actions.

- Agents are unable to:

  1. clearly identify the ***current state*** (i.e. the world is inaccessible).
  2. exactly know the ***state transitions*** caused by guessed ***actions***.

- Agents must ***experiment***, gradually discovering how its actions affect the world and what sorts of states exist.

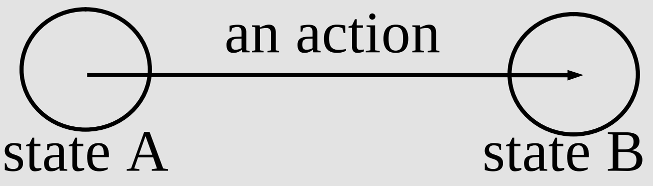
◆ Well-defined problems and solutions

A **problem** is really a collection of information that the agent will use to decide what to do.

※**Formal definition of a single-state problem contains:**

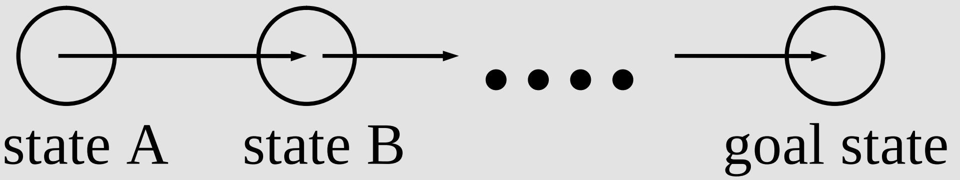
- The **initial state** (or set of states)
- The **set of possible actions, i.e. the set of operators** available to the agent.

an **operator** = description of a **state transition** due to an action.



The above two elements define the **state space** of the problem: the set of all states reachable from initial state by any sequence of actions.

A **path :**



- The **goal test**, used to test on a single state description to see if it is a goal state.

A **goal** state may be:

1. one of the possible goal states explicitly enumerated in a set.
2. specified by an ***abstracted property***, e.g. "checkmate" state in a chess game.

- The **path cost** function(***g***), a function that assign a cost to a path.

The cost of a path is the sum of the costs of the individual actions along the path.

```
datatype PROBLEM
        components: INITIAL-STATE,
                    OPERATORS,
                    GOAL-TEST,
                    PATH-COST-FUNCTION.
```

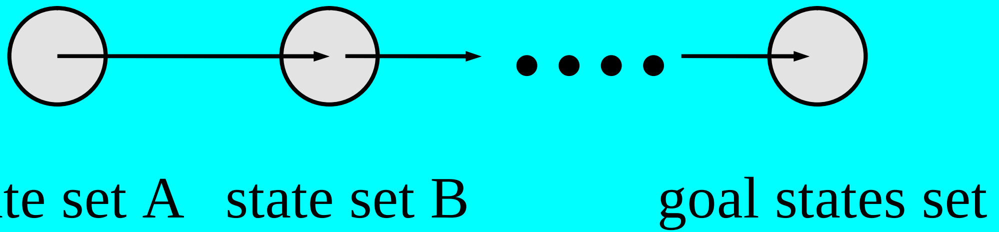
Instances of the datatype ⇨ search algorithms ⇨ **solution**

※Adapted to multiple-state problems:

datatype PROBLEM (Multiple-state)
    components: INITIAL-STATE-**SET**,
        OPERATORS-**SET**,
        GOAL-TEST,
        PATH-COST-FUNCTION.

The state space is now a **state set space** and a **solution** is now a path that leads to a set of states ***all of which are goal states***.

A **path :**



Measuring problem-solving performance:

- Measuring the **search performance (cost)**:

  1. Does it find a solution?
  2. Is it a good solution (**path cost**)?
  3. What is the **search cost**?

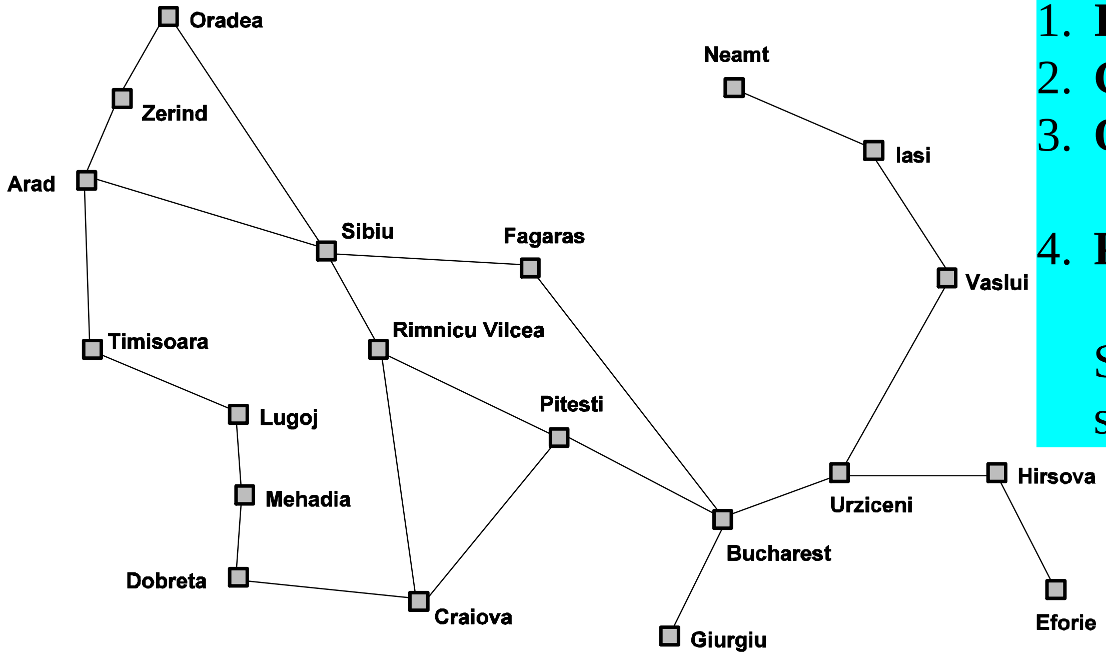
- **total cost = path cost + search cost**

path cost
trade-off
search cost

The agent must decide what resources to devote to **search** and what resource to devote to **execution.**

Choosing states and actions

※For example, drive from Arad to Bucharest in.



1. **Initial state** : Arad
2. **Goal state** : is this Bucharest?
3. **Operators** : driving along the roads between cities.
4. **Path cost** : number of steps.

So which path is the best possible solution?

※The real art of problem solving is in deciding what goes into the description of the states and operators and what is left out.

※**Abstraction**: remove unnecessary information from a representation.

Abstraction must be carried out on both **state description** and **actions**.

● **Makes it cheaper to find a solution :** removing as much details as possible while retaining **validity** and ensuring that the abstract actions are easy to carry out.

Example Problems

Examples can be distinguished between so-called **toy problems** and so-called **real-world problems**.

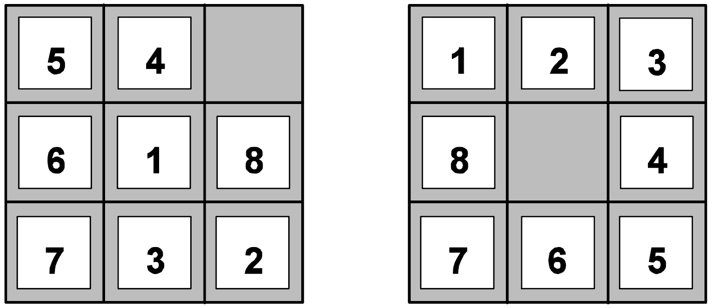
- **Toy problems**

  This kind of problems can be given a concise, exact description, and are usually used for performance evaluation of algorithms.

※The 8-puzzle problem:

Along with the 15-puzzle, is a standard test problems for new search algorithms in AI.

- **States:** a state description specifies the location of each of the eight tiles in one of the nine squares.
- **Operators:** blank moves left, right, up, or down.
- **Goal test:** state matches the goal configuration.
- **Path cost:** each step costs 1, so the path cost is just the length of the path.



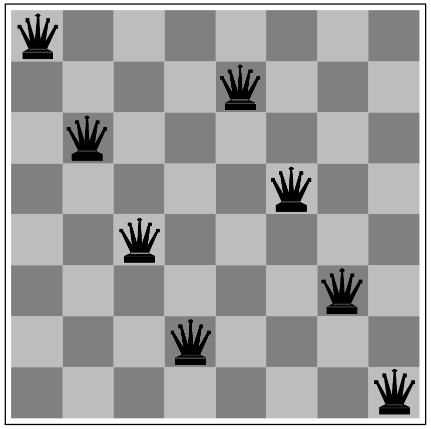
Start State Goal State

※The 8-queens problem :

There are two main kinds of formulation:

1. ***Incremental :*** placing queens one by one.
2. ***Complete-state :*** placing all 8 queens on the board and moving them around.

- **Goal test:** 8 queens on board, none attacked.
- **Path cost:** zero, the final state counts. Different algorithms are compared only on **search cost.**



Simple-minded formulation: 64^8 sequences

- **States:** any arrangement of 0 to 8 queens on board.
- **Operators:** add a queen to any square.

A more sensible formulation: 2057 sequences.

The right formulation makes a big difference to the size of the search space.

- **States:** arrangement of 0 to 8 queens with none attacked.
- **Operators:** place a queen in the left-most empty column such that it is not attacked by any other queen.

Note: the actions generate only state with no attack, but sometimes no actions will be possible.

Complete-state formulation

- **States:** arrangement of 8 queens, one in each column.
- **Operators:** move any attacked queen to another square in the same column.

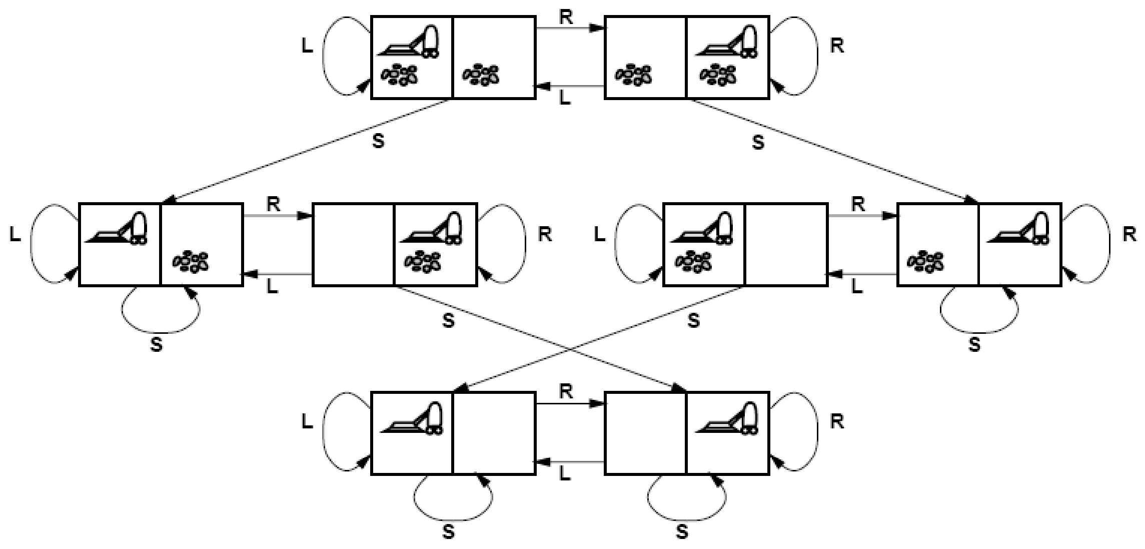
※The cryptarithmetic problem:

```
  FORTY   Solution: 29786    (F=2, O=9, etc)
+    TEN                 +   850
+    TEN                 +   850
-----------              ---------
  SIXTY                   31486
```

- **States:** a cryptarithmetic puzzle with some letters replaced by digits.
- **Operators:** replace all occurrences of a letter with a digit not already appearing in the puzzle.
- **Goal test:** puzzle conditions only digits, and represents a correct sum.
- **Path cost:** zero. All solutions equally valid.

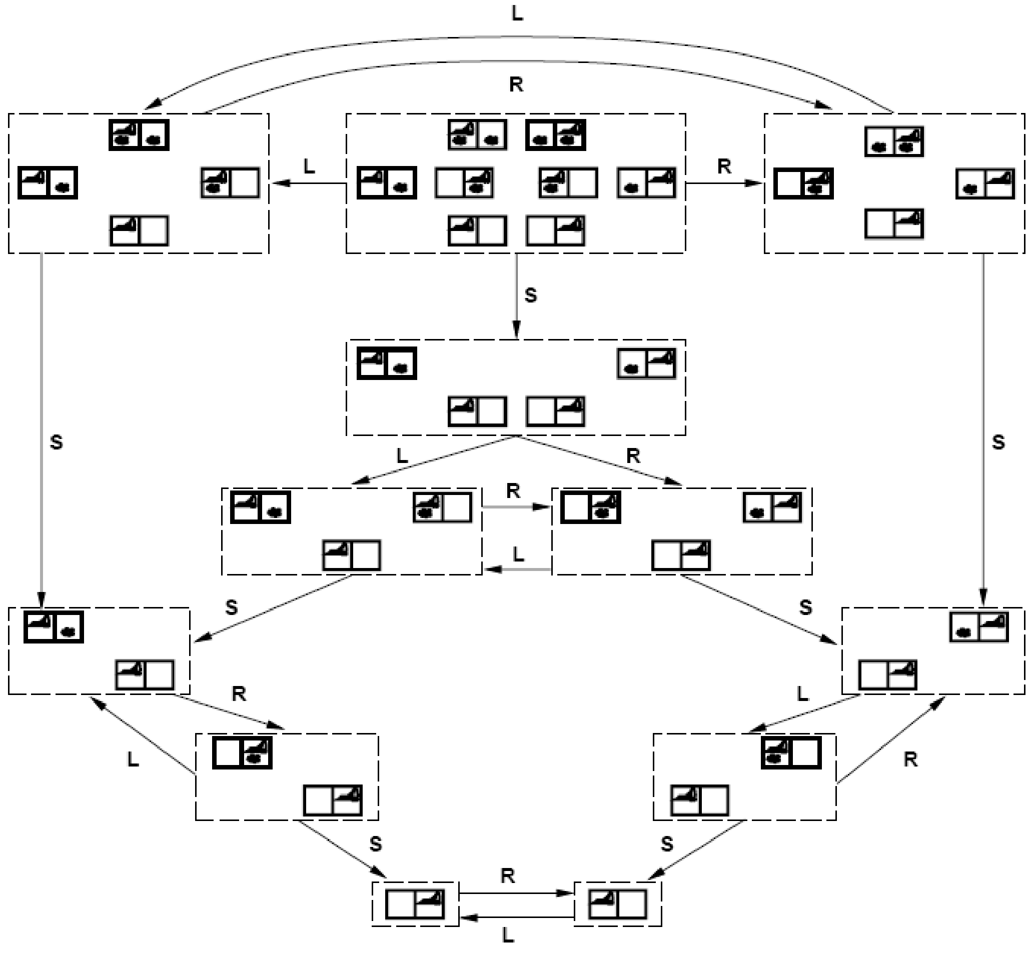
※The vacuum world problem:

Single-state problem with accessible world



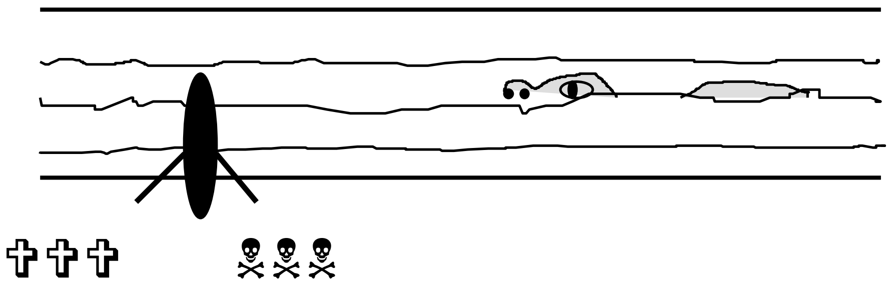
- **States:** one of the eight states shown in figure 6.
- **Operators:** move left, move right, suck.
- **Goal test:** no dirt left in any square.
- **Path cost:** each action cost 1.

Multiple-state problem with accessible world



- **States:** subsets of states 1-8 shown in figure 6.
- **Operators:** move left, move right, suck.
- **Goal test:** no dirt left in any state of the state set.
- **Path cost:** each action cost 1.

※Missionaries and cannibals problem:



- **States:** a state consists of an ordered sequence of three numbers representing the number of missionaries, cannibals, and boats on the bank of the river from which they started. Thus the **initial state** is **(3,3,1).**
- **Operators:** from each state the **five** possible operators are to take either one missionary, two missionaries, one cannibal, two cannibals, or one of each across in the boat.
- **Goal test:** reached state (0,0,0).
- **Path cost:** number of crossing.

Real-world problems

- ※**Route finding:**
- ※**Touring and travelling salesperson problems:**

  Each city must be visited exactly once. The aim is to find the shortest path.

  Developed algorithms can be used for tasks such as planning movements of automatic circuit board drills.

- ※**VLSI layout:**

  Cell layout and **channel routing**.

- ※**Robot navigation and Assembly sequencing:**

  Automatic assembly of complex objects by a robot was first demonstrated by FREEDY the robot (Michie, 1972).

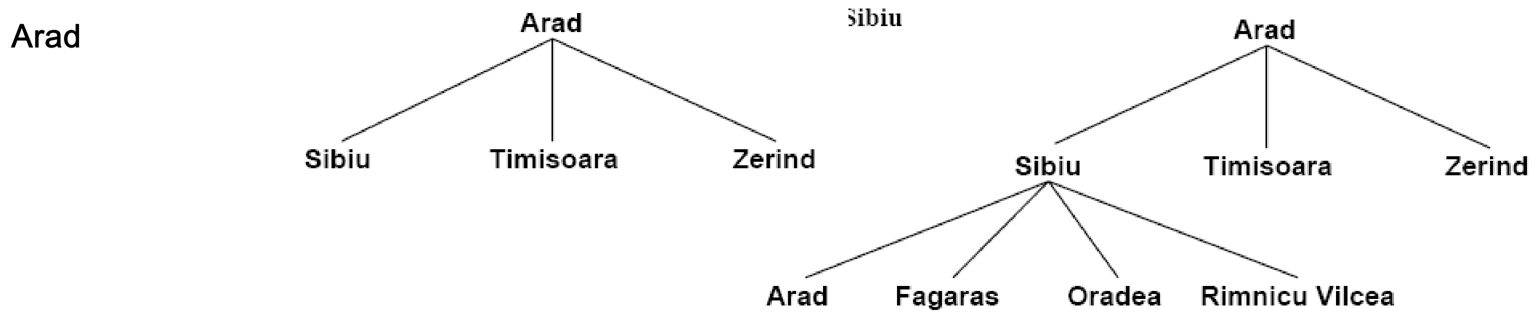
Searching for Solutions

To find a solution by searching through the defined state space-- ***the idea is to maintain and extend a set of partial solution sequences***.

Generating action sequences

Expanding a state: applying the operators to the current state and **generating** a new set of states.

Initial state   After expanding Arad   After expanding Sibiu



- To expand the newly generated states, the agent must choice one state to consider next, and put queue the others.
- The choice of which state to expand first is determined by the **search strategy**.
- The search process is abstractly a **search tree** building process with the ***root*** as the **search node** of the initial state.

- An informal general search algorithm is as:

```
function GENERAL-SEARCH(problem, strategy) returns a 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 state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
  end
```

- Distinguish between the **state space** and the **search tree**.

  state space : usually finite vs. **search tree :** usually infinite**.**

Data structures for search trees

※Data structure

Generally the data structure for a search tree node consists of five components:

1. the state in the state space to which the node corresponds;
2. the node in the search tree that generated this node (the **parent node**);
3. the operator that was applied to generated the node;
4. the number of nodes on the path from root to this node (the **depth** of the node);
5. the path cost of the path from the initial state to the node.

```
datatype node
components: STATE,
            PARENT-NODE,
            OPERATOR,
            DEPTH,
            PATH-COST
```

※Fringe or frontier

The collection of nodes that are waiting to be expanded--it is usually implemented as a **queue** with the following functions:

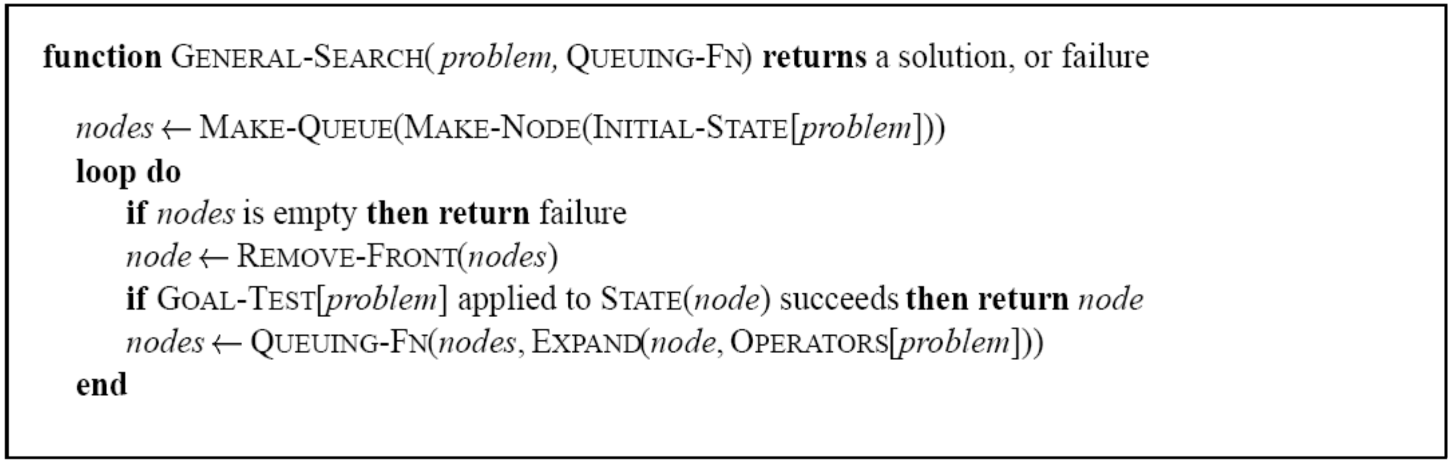
MAKE-QUEUE(*Elements*)

EMPTY?(*Queue*)

REMOVE-FRONT(*Queue*)

QUEUING-FN(Elements, *Queue*)

※ A more formal general search algorithm



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

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
      if nodes is empty then return failure
      node ← REMOVE-FRONT(nodes)
      if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
      nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

Search Strategies

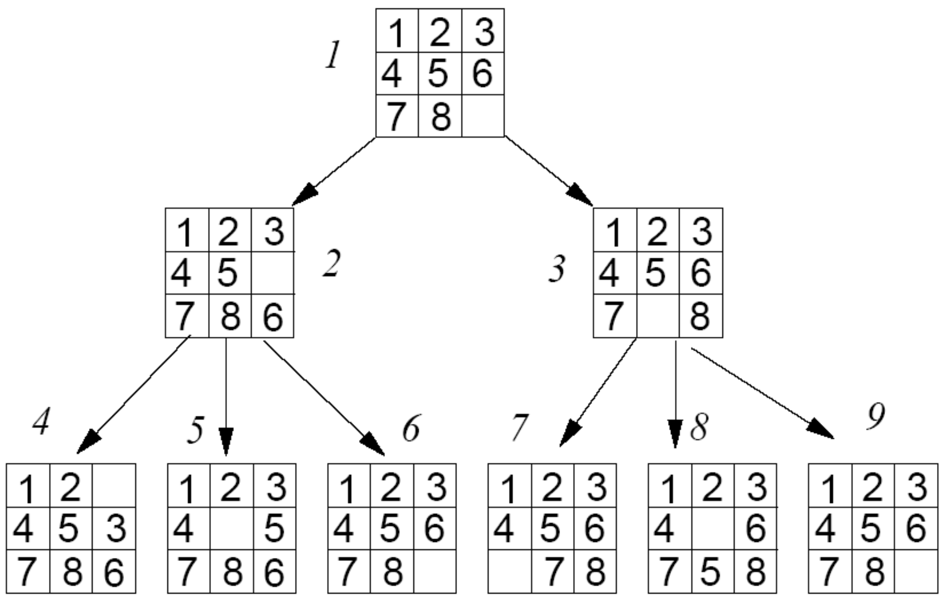
Evaluation criteria

- ✧**completeness:** will the strategy guarantee to find a solution if one exists?
- ✧**time efficiency**: time spent to find a solution?
- ✧**space efficiency**: space (memory) required to perform the search?
- ✧**optimality of solution**: will the strategy choose the highest-quality solution when there are several different solutions?

Uninformed search strategies (blind search)

Strategies that have no information about the number of steps or the path cost from the current state to the goal--***just to distinguish if a state is a goal or not***.

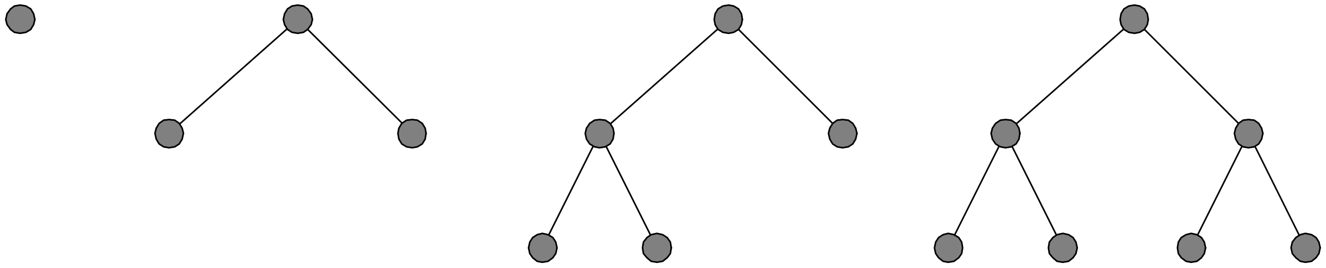
✧Contrast to **Informed search** or **heuristic search**



※**Breadth-first search** (FIFO queuing strategy)

All the nodes at depth ***d*** in the search tree are expanded before the nodes at depth ***d+1***.

```
function BREADTH-FIRST-SEARCH(problem)
        returns a solution or failure
  return GENERAL-SEARCH(problem,
              ENQUEUE-AT-END)
```



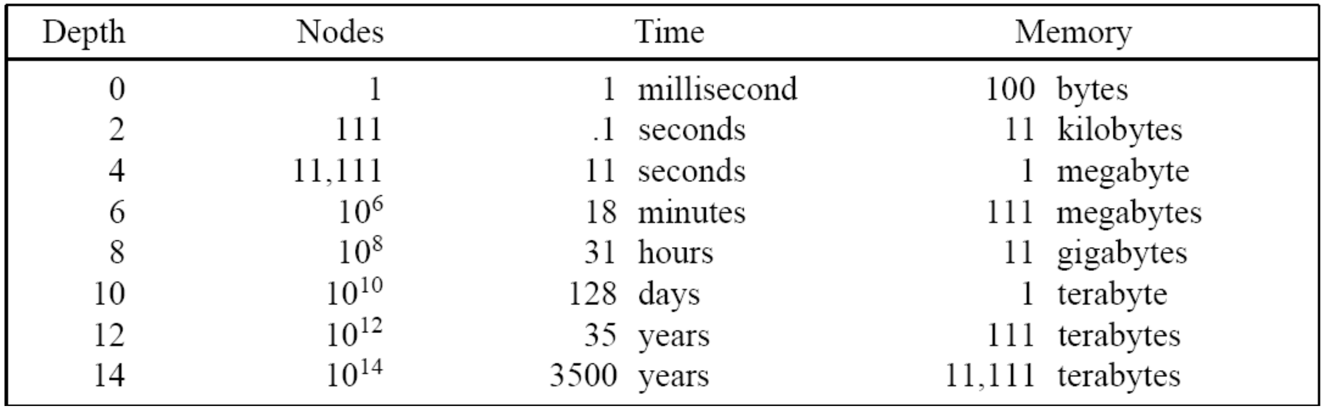
- ✧ **Completeness:** yes
- ✧ **Optimality:** yes, *provided the path cost is a non-decreasing function of the depth of the node*.
- ✧ **time and space:** O(**branching_factor**$^{\textbf{depth}}$)

  Branching factor *b*- the number of new states generated by the current state.

  The maximum number of nodes expanded to find a solution at depth **d** is:

$$\boldsymbol{1 + b + b^2 + b^3 + \ldots\ldots. + b^d}$$

The time and space required for a breadth-first search with branching factor ***b = 10*** is shown in (***Figure 12***) for various depth d.



Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

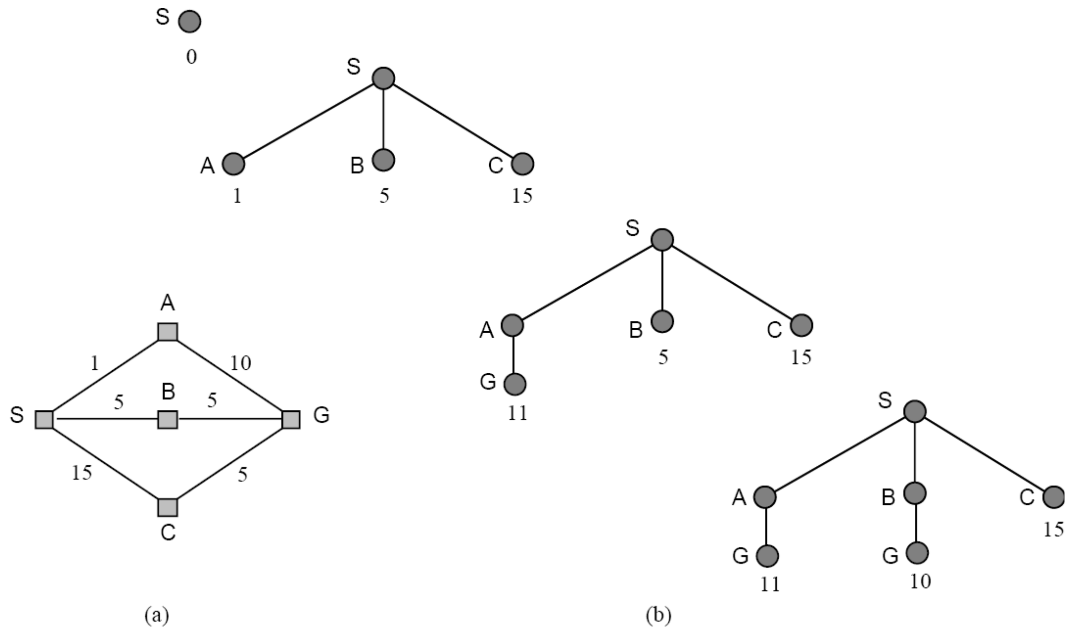
The conclusion:

1. *Always find the shallowest goal state, but this may not always the least-cost solution for a general path cost function.*
2. *Empirically the space requirement is a larger problem than the execution time.*
3. *In general, exponential complexity search problems cannot be solved for any but the smallest instances.*

※Uniform cost search

Modifies the breadth-first strategy by always expanding the lowest-cost fringe node first. It is a generalization of Breadth first search:

Breadth-first search is uniform cost search with *g(n)*=DEPTH(n).



The search will stop when a solution with the cheapest path cost in the queue is found.

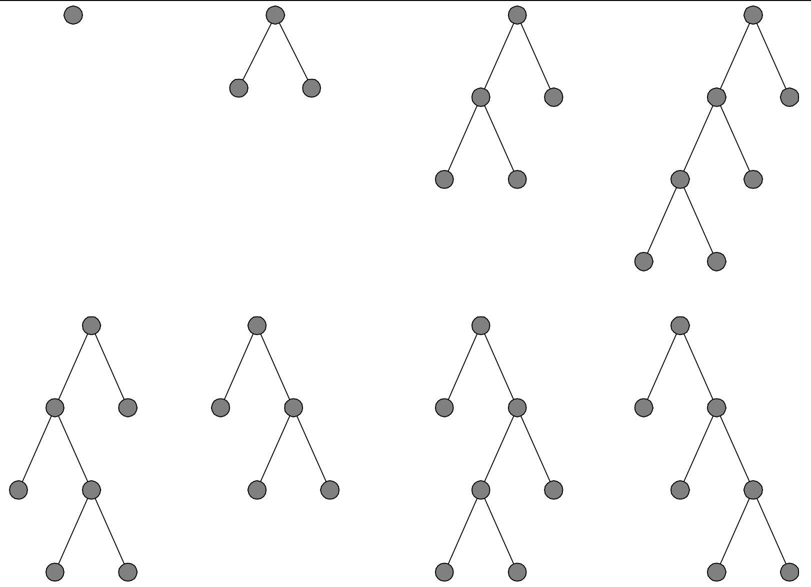
- ✧ **Completeness:** yes
- ✧ **Optimality:** yes, *provided the path cost is a non-decreasing function of the depth of the node*, i.e.

$$\forall \boldsymbol{n}\ g(\text{SUCCESSOR}(n)) \geq g(n)$$

※**Depth-first search** (stacking strategy)

Always expands one of the nodes at the deepest level of the tree.

```
function DEPTH-FIRST-SEARCH(problem)
returns a solution or failure
   return GENERAL-SEARCH(problem,
                         ENQUEUE-AT-FRONT)
```

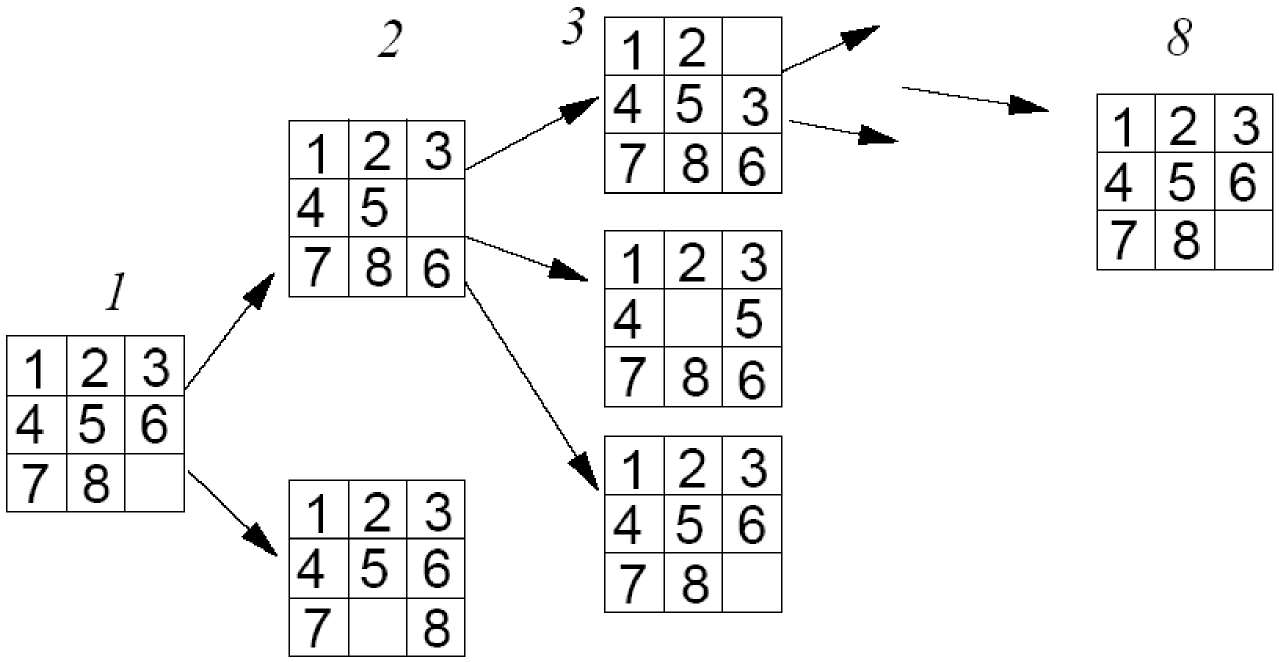


- ✧ **Completeness:** no! may go down and not come back
- ✧ **Time:** ***O*****(branch_factor^depth)**
- ✧ **Optimality:** no! returns the first found solution, not necessarily the ideal one.
- ✧ **Space:** ***O*****(depth)**

For a state space with branching factor ***b*** and maximum depth ***m***, depth-first search requires storage of only ***bm***

The conclusion:

1. *May get stuck going down an infinite search path, or find a solution path that is longer than a shallower optimal one.*
2. *Should be avoided for search trees with larger or infinite maximum depths.*



1
2
3
8

※**Depth-limited search** (DFS down to some extent)

Avoids the pitfalls of DFS by imposing a **cutoff** on the maximum depth of a path.

For example, if there are 20 cities, then the path must be of length 19 at the longest. The cutoff may be set to 19.

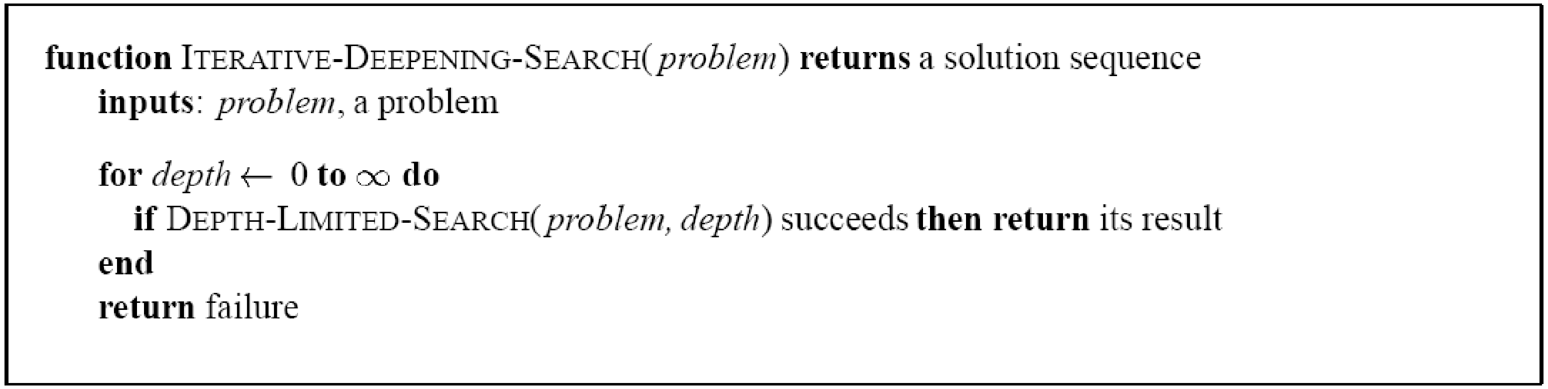
- ✧**Completeness:** yes, provided there are solutions exist before cutoff.
- ✧**time:** ***O*(branch_factor^depth_limit)**
- ✧**space:** ***O*(depth_limit)**
- ✧**Optimality:** no!

The conclusion:

1. *May find a solution path that is longer than a shallower optimal one.*
2. *Choosing a smaller depth limit may affect the completeness.*

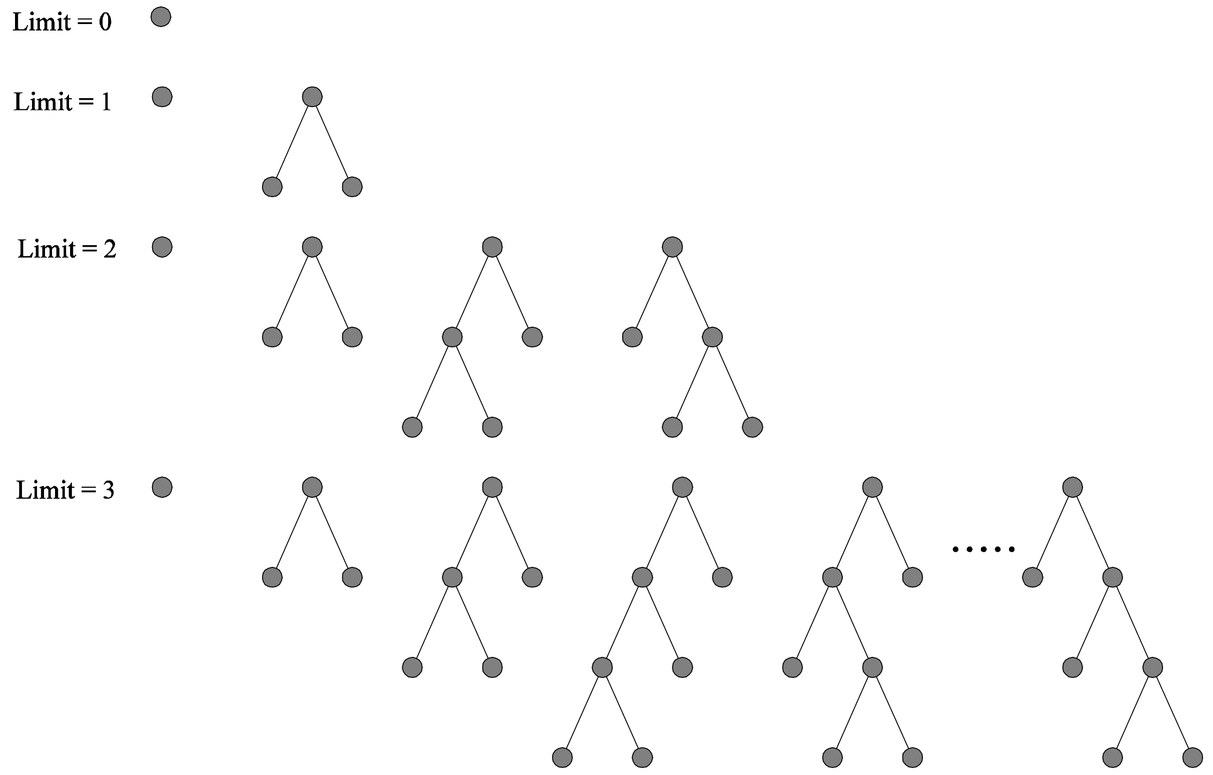
※**Iterative deepening search** (successively increasing depth-limit)

Combines the benefits of DFS and BFS, using a strategy of choosing the best depth limit by trying all possible depth limits.



```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
    end
    return failure
```



The **diameter** of state space: max anticipated path length for most problems.

- ✧ **Completeness:** yes (like Breadth-first).
- ✧ **Time: O(branch_factor^diameter)**
- ✧ **Optimality: yes (like Breadth-first).**
- ✧ **Space: O(diameter) (like Depth-first)**

It may seem wasteful to expand so many states multiple times, however, the overhead of the multiple expansion is actually rather small.

$$\boldsymbol{1 + b + b^2 + b^3 + \ldots\ldots. + b^{d-1} + b^d}$$

where $\boldsymbol{b^{d-1}}$ ***is usually*** $\boldsymbol{\ll}$ $\boldsymbol{b^d}$

The total number of expansions is:

$$(d+1)1+(d)b+(d-1)b^2+ \ldots +3b^{d-2}+2b^{d-1}+1b^d$$

For branching factor *b=10*, and *d=5* :

1+10+100+1,000+10,000+100,000 = 111,111
6+50+400+3,000+20,000+100,000 = 123,456

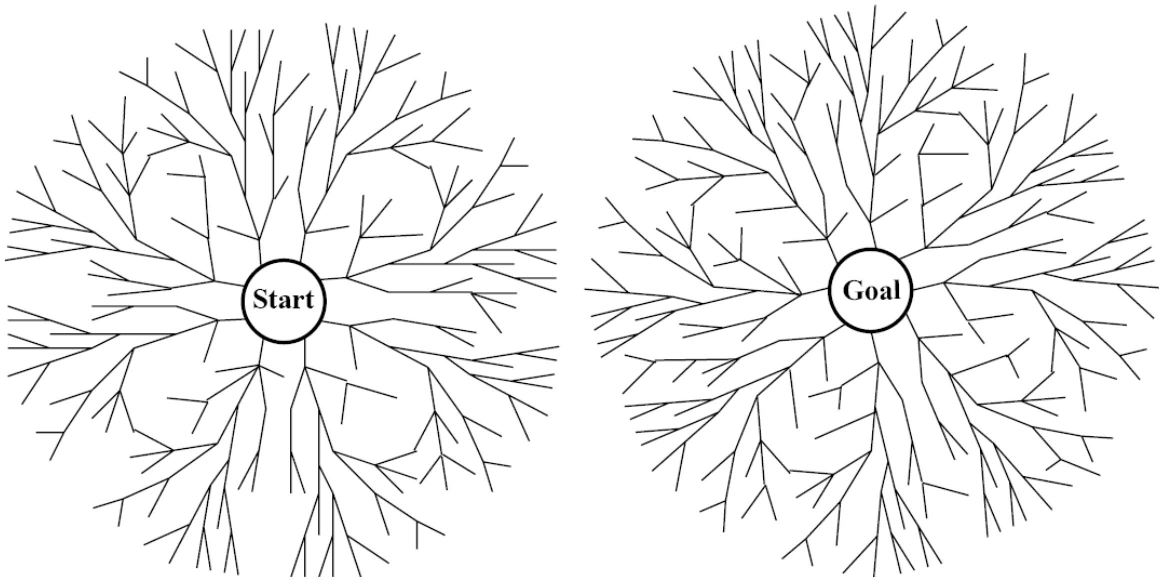
With a total overhead of ***11%***

The conclusion:

1. *The higher the branching factor, the lower the overhead of repeatedly expanded states*.
2. *In general, iterative deepening is the prefered search method **when there is a large search space and the depth of the solution is not known***.

※**Bidirectional search(**Faster but unpractical**)**

To simultaneously search both forward from the initial state and backward from the goal. The search stops when two searches meet in the middle.



Comparision with BFS for
branching factor *b=10*, and depth *d=6* :

1+10+100+1,000+10,000+100,000 = 1,111,111
2 * (1+10+100+1,000) = 2,222

- ✧ **Completeness:** yes (like Breadth-first).
- ✧ **Time:** ***O***(2***branch_factor^diameter/2**)
- ✧ **Optimality:** yes (like Breadth-first).
- ✧ **Space:** ***O***(2***branch_factor^diameter/2**)

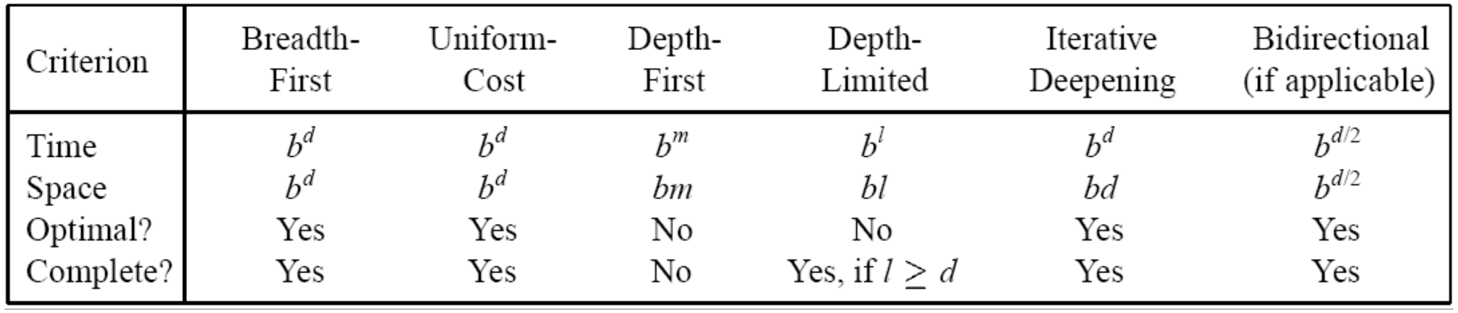
Some issues must be addressed:

1. How to define the **predecessors** of a node and to generate precedessors successively starting from the goal node.
2. Are the operators reversible(e.g. route finding, cannibals, etc.)
3. What if there are many possible goal states? and what if the goal state is just an description, e.g. checkmate?
4. What is the efficient way to check if a new node has been generated in the other half?
5. What kind of search is used for each half?

The conclusion:

1. *The idea sounds great, but in practical it is tricky to implement.*
2. *In order to check if the two searches have met, at least one of the searches must store all the generated nodes in the memory. This makes the space complexity* $\mathbf{O(b^{d/2})}$

Comparing search strategies



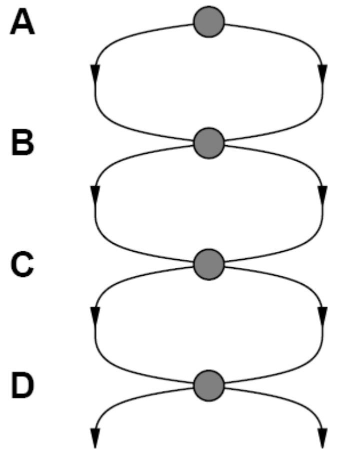
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

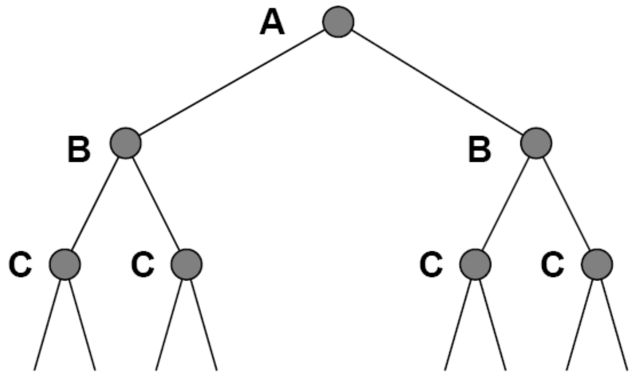
Avoiding Repeated States

Search processes must avoid expanding states that have already been encountered and expanded before.

Problems with **reversible operators** (route finding, missionaries and cannibals) tend to generate repeated states

- **Wasting time and memory**





How to deal with repeated states

- ✧ Do not generate a state the same to the state you just came from.
- ✧ Do not create paths with cycles in them(i.e. check if the new node will be the same as one of its ancestors).
- ✧ Do not generate any state that was ever generated before. This requires buffering all the generated states.

For efficiency, search algorithm of the last option can make use of a hash table to store all the generated nodes.