Foundations of AI

3. Solving Problems by Searching

Problem-Solving Agents, Formulating Problems, Search Strategies

Wolfram Burgard, Andreas Karwath, Bernhard Nebel, and Martin Riedmiller

Contents

- Problem-Solving Agents
- Formulating Problems
- Problem Types
- Example Problems
- Search Strategies

Problem-Solving Agents

→ Goal-based agents

Formulation: *goal* and *problem*

Given: *initial state*

Goal: To reach the specified goal (a state) through the *execution of appropriate actions.*

→ Search for a suitable action sequence and execute the actions

A Simple Problem-Solving Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  inputs: percept, a percept
  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 do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Properties of this Agent

- Static world
- Observable environment
- Discrete states
- Deterministic environment

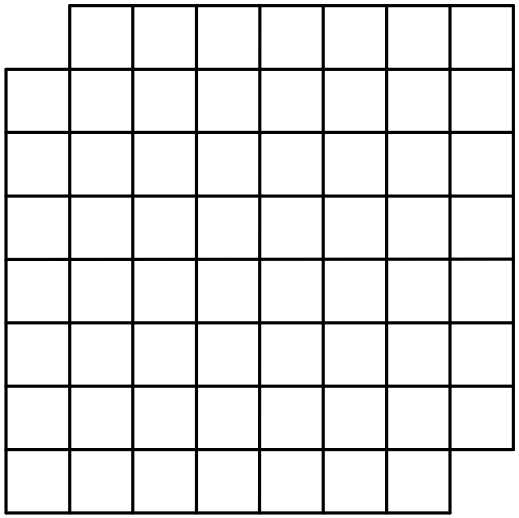
Problem Formulation

- Goal formulation
  World states with certain properties
- Definition of the state space
  (important: only the relevant aspects → abstraction)
- Definition of the actions that can change the world state
- Definition of the problem type, which depends on the knowledge of the world states and actions
  → states in the search space
- Specification of the search costs (search costs, offline costs) and the execution costs (path costs, online costs)

Note: The type of problem formulation can have a serious influence on the difficulty of finding a solution.

Example Problem Formulation

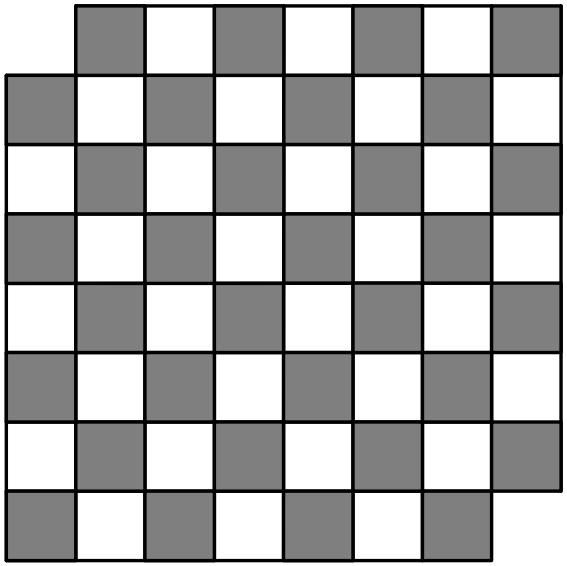
Given an nxn board from which two diagonally opposite corners have been removed (here 8x8):



Goal: Cover the board completely with dominoes, each of which covers two neighbouring squares.

→ Goal, state space, actions, search, …

Alternative Problem Formulation



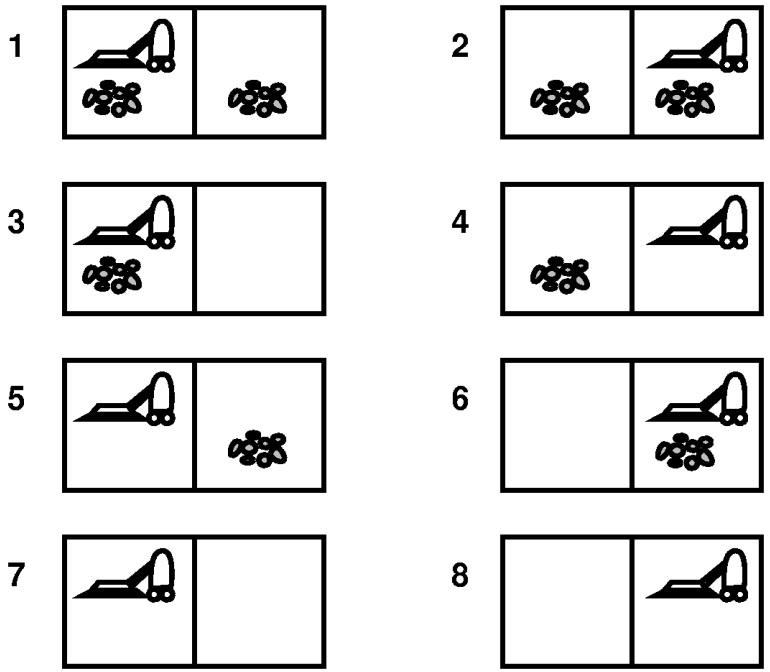
Question:

Can a chess board consisting of $n^2/2$ black and $n^2/2-2$ white squares be completely covered with dominoes such that each domino covers one black and one white square?

… clearly not.

Problem Formulation for the Vacuum Cleaner World

- World state space:
  2 positions, dirt or no dirt
  → 8 world states
- Actions:
  Left (L), Right (R), or Suck (S)
- Goal:
  no dirt in the rooms
- Path costs:
  one unit per action



Problem Types: Knowledge of States and Actions

- Single-state problem
  Complete world state knowledge
  Complete action knowledge
  → The agent always knows its world state

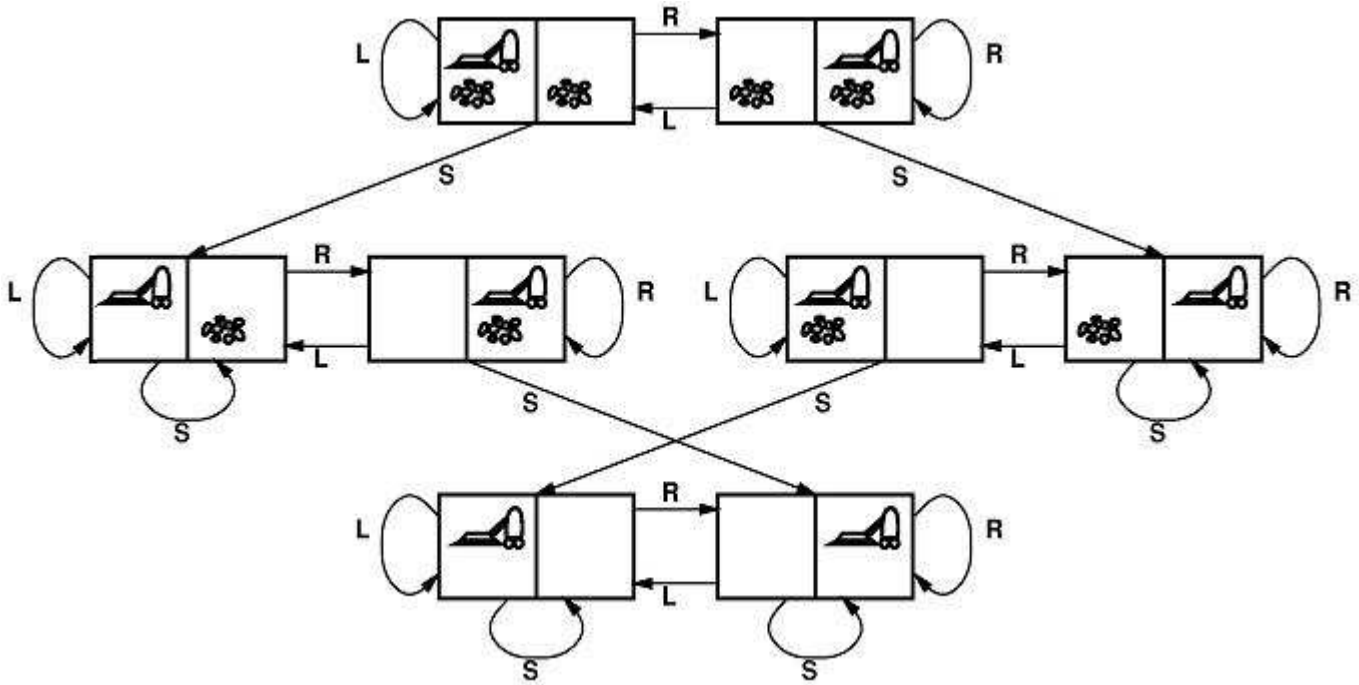
- Multiple-state problem
  Incomplete world state knowledge
  Incomplete action knowledge
  → The agent only knows which group of world states it is in

- Contingency problem
  It is impossible to define a complete sequence of actions that constitute a solution in advance because information about the intermediary states is unknown.

- Exploration problem
  State space and effects of actions unknown. Difficult!

The Vacuum Cleaner Problem as a One-State Problem

If the environment is completely accessible, the vacuum cleaner always knows where it is and where the dirt is. The solution then is reduced to searching for a path from the initial state to the goal state.



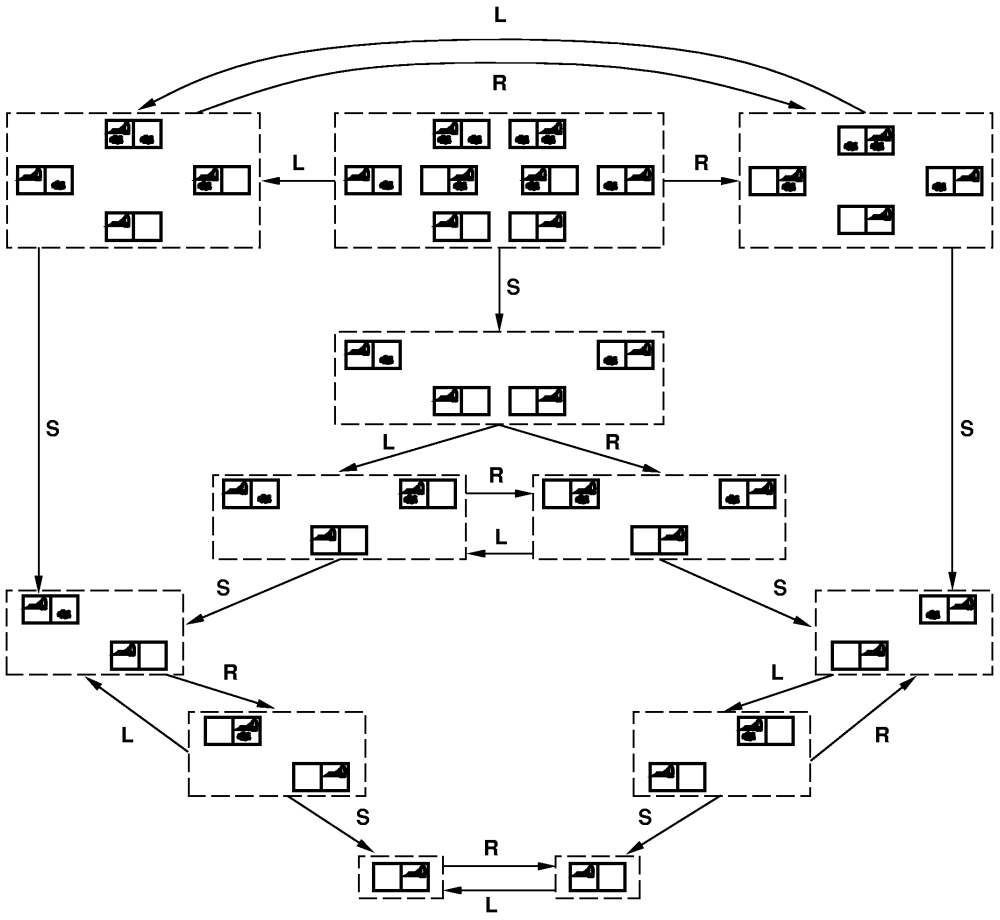
States for the search: The world states 1-8.

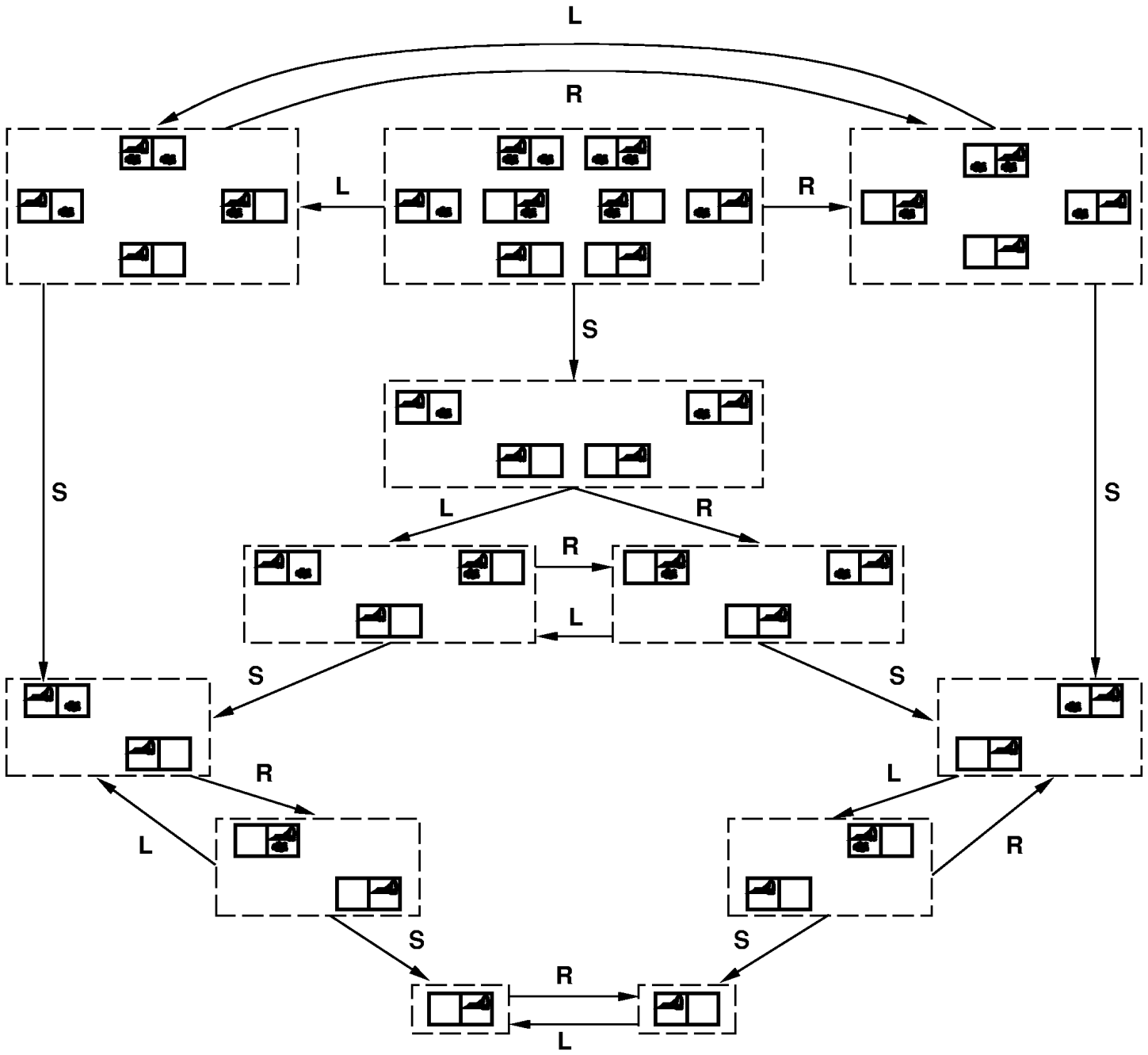
The Vacuum Cleaner World as a Multiple-State Problem

If the vacuum cleaner has no sensors, it doesn't know where it or the dirt is.

In spite of this, it can still solve the problem. Here, states are knowledge states.

States for the search: The power set of the world states 1-8.




L
R
L
R
S
S
S
L
R
R
L
S
S
R
L
L
R
S
S
R
L
S
S
R
L

Concepts (1)

Initial State

The state from which the agent infers that it is at the beginning

State Space

Set of all possible states

Actions

Description of possible actions and their outcome (successor function)

Goal Test

Tests whether the state description matches a goal state

Concepts (2)

Path

A sequence of actions leading from one state to another.

Path Costs

Cost function g over paths. Usually the sum of the costs of the actions along the path.

Solution

Path from an initial to a goal state

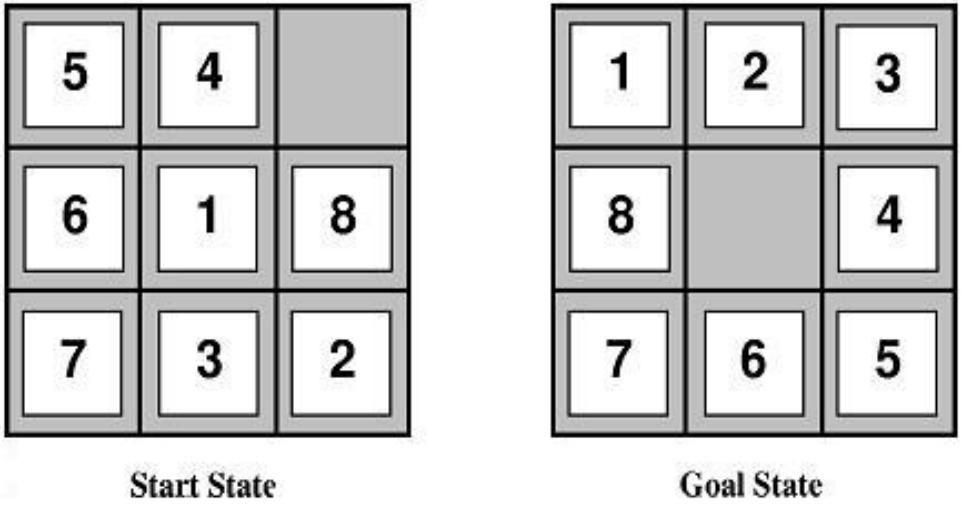
Search Costs

Time and storage requirements to find a solution

Total Costs

Search costs + path costs

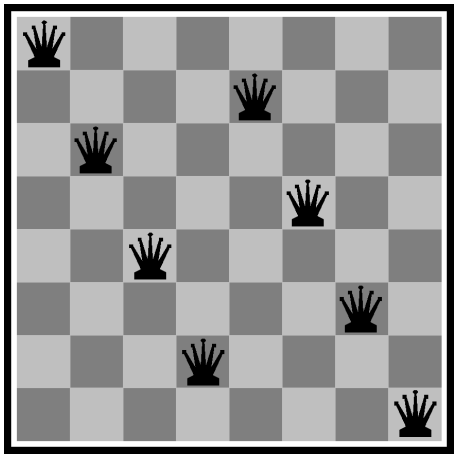
Example: The 8-Puzzle



- States:
  Description of the location of each of the eight tiles and (for efficiency) the blank square.
- Initial State:
  Initial configuration of the puzzle.
- Actions or Successor function:
  Moving the blank left, right, up, or down.
- Goal Test:
  Does the state match the configuration on the right (or any other configuration)?
- Path Costs:
  Each step costs 1 unit (path costs corresponds to its length).

Example: 8-Queens Problem

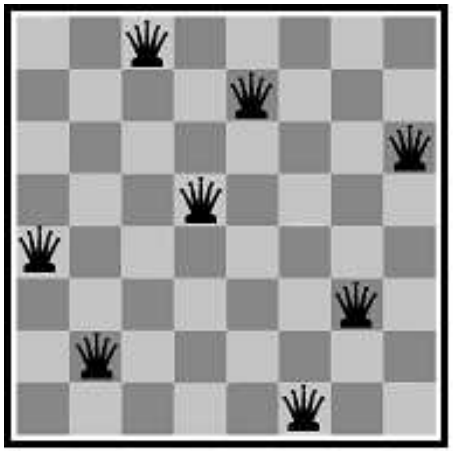
Almost a solution:



- States:
  Any arrangement of 0 to 8 queens on the board.
- Initial state:
  No queen on the board.
- Successor function:
  Add a queen to an empty field on the board.
- Goal test:
  8 queens on the board such that no queen attacks another
- Path costs:
  0 (we are only interested in the solution).

Example: 8-Queens Problem

A solution:



- States:
  Any arrangement of 0 to 8 queens on the board.
- Initial state:
  No queen on the board.
- Successor function:
  Add a queen to an empty field on the board.
- Goal test:
  8 queens on the board such that no queen attacks another
- Path costs:
  0 (we are only interested in the solution).

Alternative Formulations

- Naïve formulation
  - States: Any arrangement of 0-8 queens
  - Problem: $64 \cdot 63 \cdot \ldots \cdot 57 \approx 10^{14}$ possible states
- Better formulation
  - States: any arrangement of n queens ($0 \leq n \leq 8$) one per column in the leftmost n columns such that no queen attacks another.
  - Successor function: add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
  - Problem: 2,057 states
  - Sometimes no admissible states can be found.

Example: Missionaries and Cannibals

Informal problem description:

- Three missionaries and three cannibals are on one side of a river that they wish to cross.
- A boat is available that can hold at most two people.
- You must never leave a group of missionaries outnumbered by cannibals on the same bank.

→ Find an action sequence that brings everyone safely to the opposite bank.

Formalization of the M&C Problem

States: triple (x,y,z) with $0 \leq x,y,z \leq 3$, where x,y, and z represent the number of missionaries, cannibals and boats currently on the original bank.

Initial State: (3,3,1)

Successor function: from each state, either bring one missionary, one cannibal, two missionaries, two cannibals, or one of each type to the other bank.

Note: not all states are attainable (e.g., (0,0,1)), and some are illegal.

Goal State: (0,0,0)

Path Costs: 1 unit per crossing

Examples of Real-World Problems

- Route Planning, Shortest Path Problem

  Simple in principle (polynomial problem). Complications arise when path costs are unknown or vary dynamically (e.g., route planning in Canada)

- Travelling Salesperson Problem (TSP)

  A common prototype for NP-complete problems

- VLSI Layout

  Another NP-complete problem

- Robot Navigation (with high degrees of freedom)

  Difficulty increases quickly with the number of degrees of freedom. Further possible complications: errors of perception, unknown environments

- Assembly Sequencing

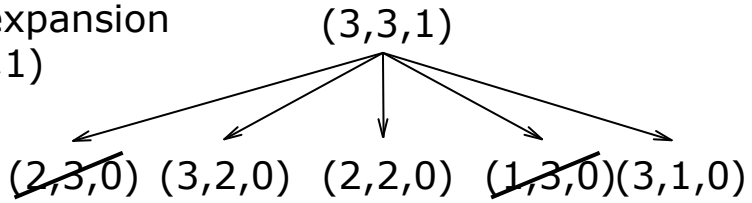
  Planning of the assembly of complex objects (by robots)

General Search

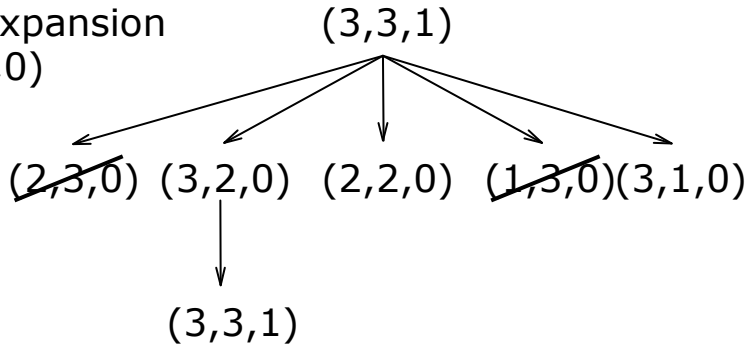
From the initial state, produce all successive states step by step → search tree.

(a) initial state (3,3,1)

(b) after expansion of (3,3,1)



(c) after expansion of (3,2,0)



Implementing the Search Tree

Data structure for nodes in the search tree:

State: state in the state space

Parent-Node: Predecessor nodes

Action: The operator that generated the node

Depth: number of steps along the path from the initial state

Path Cost: Cost of the path from the initial state to the node

Operations on a queue:

Make-Queue(Elements): Creates a queue

Empty?(Queue): Empty test

First(Queue): Returns the first element of the queue

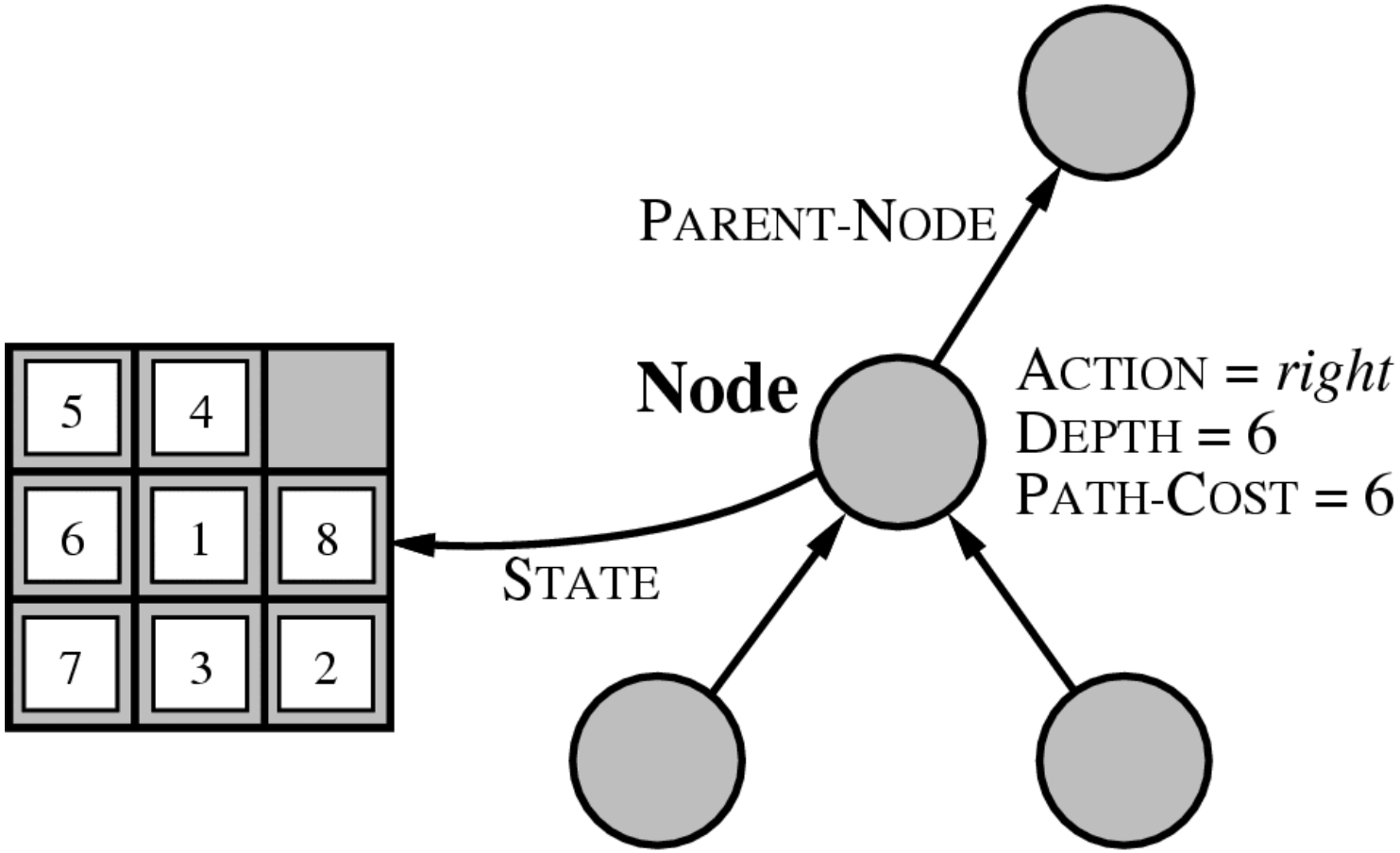
Remove-First(Queue): Returns the first element

Insert(Element, Queue): Inserts new elements into the queue

(various possibilities)

Insert-All(Elements, Queue): Inserts a set of elements into the queue

Nodes in the Search Tree



General Tree-Search Procedure

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure

  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem] applied to STATE[node] succeeds
      then return SOLUTION(node)
    fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

---------------------------------------------------------------

function EXPAND(node, problem) returns a set of nodes

  successors ← the empty set
  for each ⟨action, result⟩ in SUCCESSOR-FN[problem](STATE[node]) do
    s ← a new NODE
    STATE[s] ← result
    PARENT-NODE[s] ← node
    ACTION[s] ← action
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors
```

Criteria for Search Strategies

Completeness:

Is the strategy guaranteed to find a solution when there is one?

Time Complexity:

How long does it take to find a solution?

Space Complexity:

How much memory does the search require?

Optimality:

Does the strategy find the best solution (with the lowest path cost)?

Search Strategies

Uninformed or blind searches:

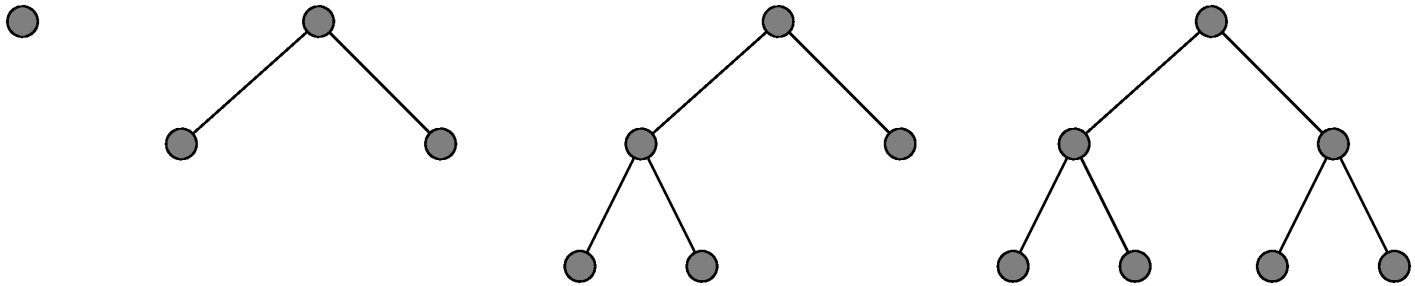
No information on the length or cost of a path to the solution.

- breadth-first search, uniform cost search, depth-first search,
- depth-limited search, Iterative deepening search, and
- bi-directional search.

In contrast: informed or heuristic approaches

Breadth-First Search (1)

Nodes are expanded in the order they were produced. *(fringe = FIFO-QUEUE()).*



- Always finds the shallowest goal state first.
- Completeness is obvious.
- The solution is optimal, provided every action has identical, non-negative costs.

Breadth-First Search (2)

The costs, however, are very high. Let b be the maximal branching factor and d the depth of a solution path. Then the maximal number of nodes expanded is

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

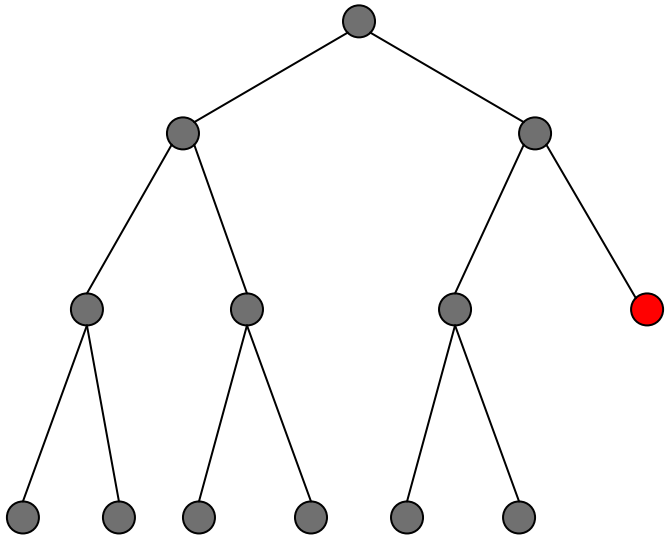
Example: b = 10, 10,000 nodes/second, 1,000 bytes/node:

Depth	Nodes	Time	Memory
2	1,100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Breadth-First Search (3)

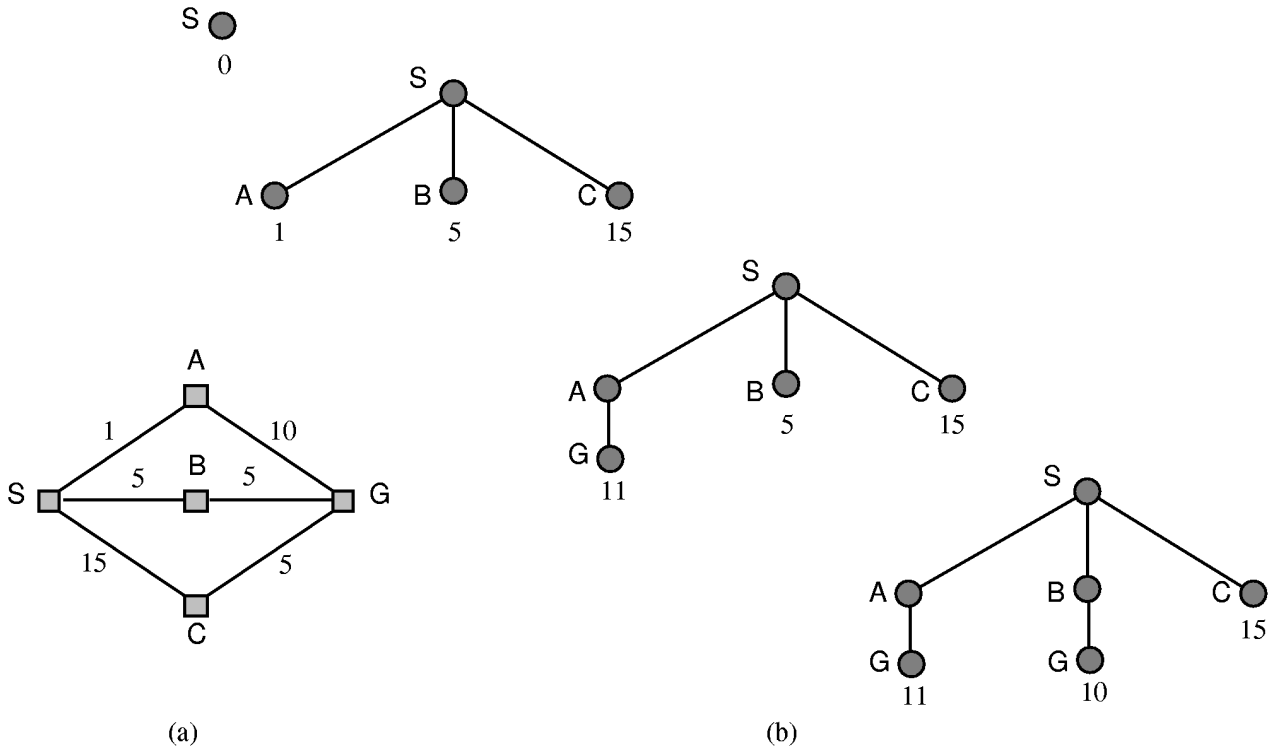
- The BFS implementation as shown is quite **inefficient**, because it always stores the final layer without using the nodes!
- Change the general search algorithm so that the goal test is performed **before** the nodes are inserted into the queue.
- This reduces the number of expanded nodes to:

$1 + b + b^2 + b^3 + \ldots + b^d \in O(b^d)$



Uniform Cost Search

Modification of breadth-first search to always expand the node with the lowest-cost *g(n)*.

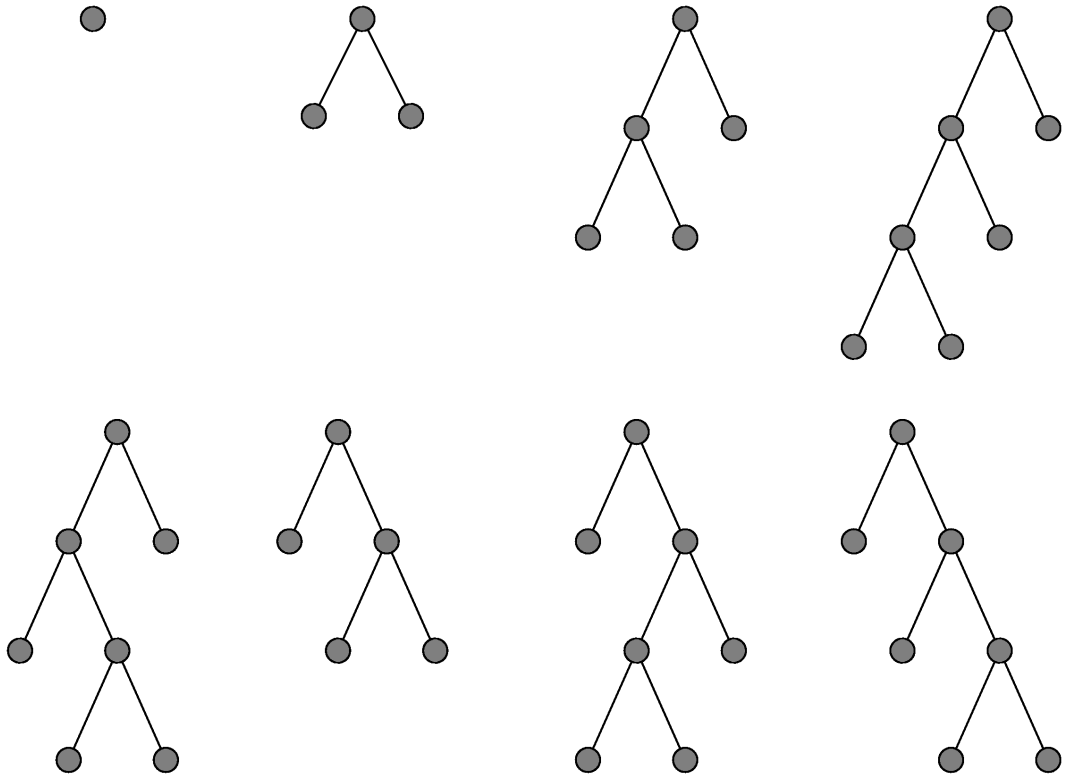


Always finds the cheapest solution, given that *g(successor(n)) >= g(n)* for all n.

Depth-First Search

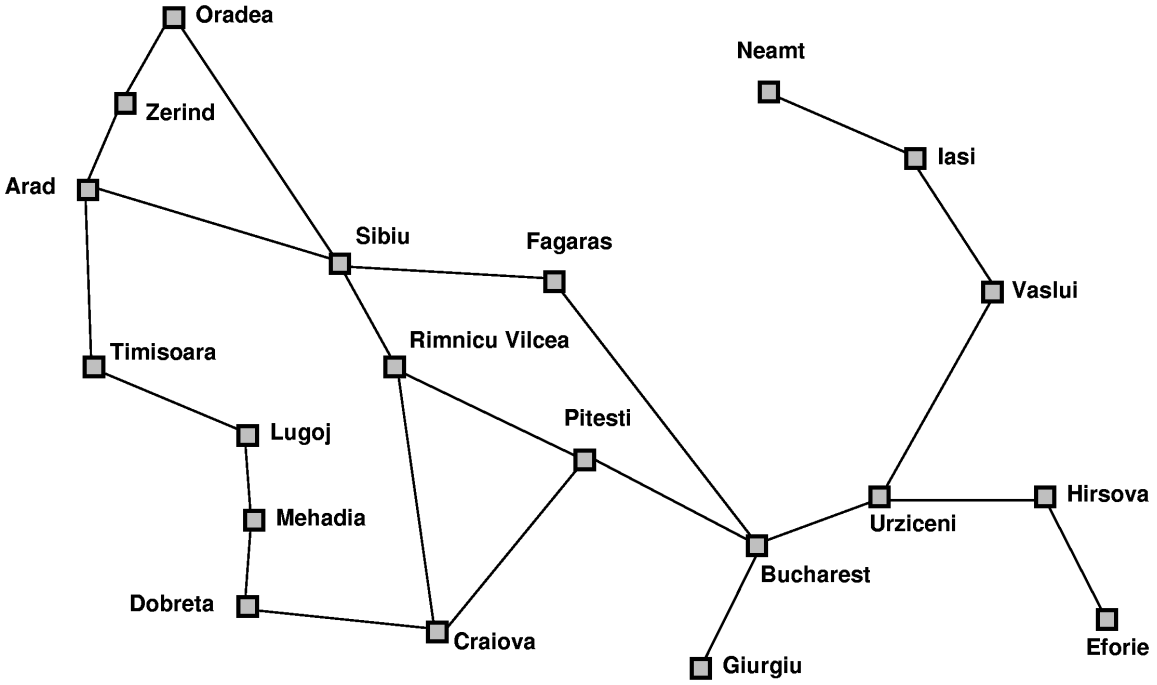
Always expands an unexpanded node at the greatest depth (Queue-Fn = Enqueue-at-front).

Example (Nodes at depth 3 are assumed to have no successors):



Depth-Limited Search

Depth-first search with an imposed cutoff on the maximum depth of a path. E.g., route planning: with *n* cities, the maximum depth is *n–1*.



Here, a depth of 9 is sufficient (diameter of the problem).

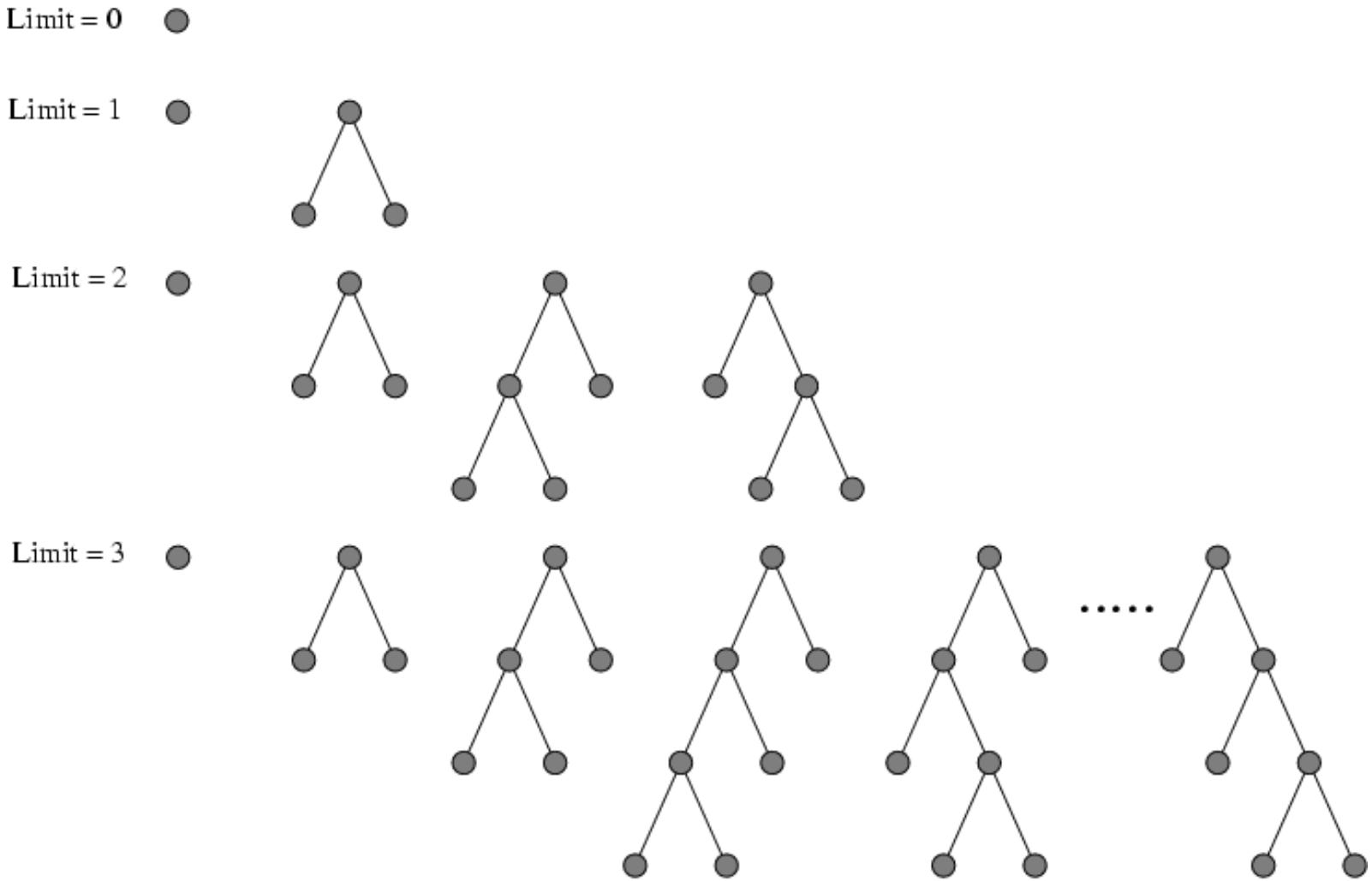
Iterative Deepening Search (1)

- Combines depth- and breadth-first searches
- Optimal and complete like breadth-first search, but requires less memory

```
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
```

Example



Iterative Deepening Search (2)

Number of expansions

Iterative Deepening Search	$(d)b + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$
Breadth-First-Search	$b + b^2 + \ldots + b^{d-1} + b^d + b^{d+1} - b$

Example: b = 10, d = 5

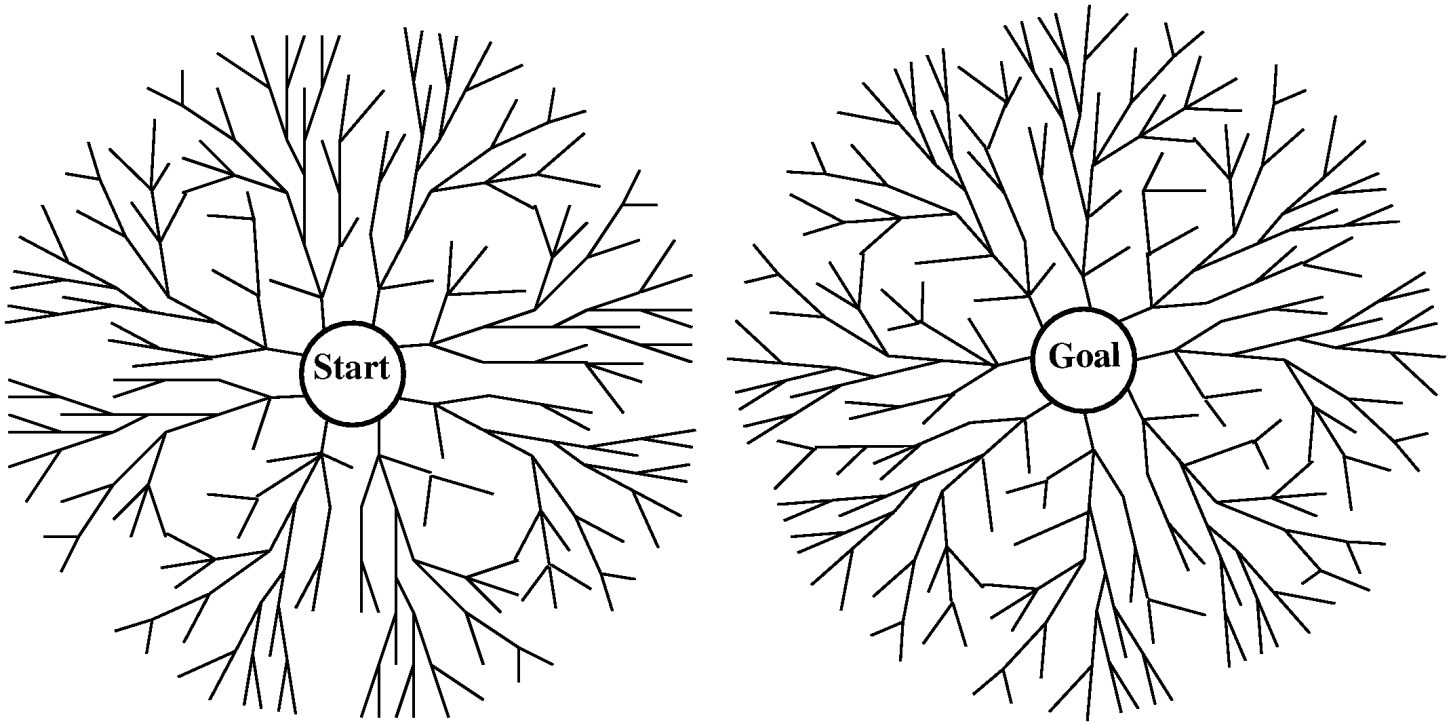
Breadth-First-Search	10 + 100 + 1,000 + 10,000 + 999,990 = 1,111,100
Iterative Deepening Search	50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

For b = 10, only 11% of the nodes expanded by breadth-first-search are generated, so that the memory requirement is considerably lower.

Time complexity: $O(b^d)$ Memory complexity: $O(b \cdot d)$

→ Iterative deepening in general is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

Bidirectional Searches



As long as forwards and backwards searches are symmetric, search times of $O(2 \cdot b^{d/2}) = O(b^{d/2})$ can be obtained.

E.g., for b=10, d=6, instead of 111111 only 2222 nodes!

Problems with Bidirectional Search

- The operators are not always reversible, which makes calculation the predecessors very difficult.
- In some cases there are many possible goal states, which may not be easily describable. Example: the predecessors of the checkmate in chess.
- There must be an efficient way to check if a new node already appears in the search tree of the other half of the search.
- What kind of search should be chosen for each direction (the previous figure shows a breadth-first search, which is not always optimal)?

Comparison of Search Strategies

Time complexity, space complexity, optimality, completeness

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

b branching factor
d depth of solution,
m maximum depth of the search tree,
l depth limit,
C* cost of the optimal solution,
$\in$ minimal cost of an action

Superscripts:
a) b is finite
b) if step costs not less than $\in$
c) if step costs are all identical
d) if both directions use breadth-first search

Summary

- Before an agent can start searching for solutions, it must formulate a goal and then use that goal to formulate a problem.
- A problem consists of five parts: The state space, initial situation, actions, goal test, and path costs. A path from an initial state to a goal state is a solution.
- A general search algorithm can be used to solve any problem. Specific variants of the algorithm can use different search strategies.
- Search algorithms are judged on the basis of completeness, optimality, time complexity, and space complexity.