INF5390 – Kunstig intelligens

Solving Problems by Searching

Roar Fjellheim

Outline

- Problem-solving agents
- Example problems
- Search programs
- Uninformed search
- Informed search
- Summary

AIMA Chapter 3: Solving Problems by Searching

Problem-solving agents

- Goal-based agents know their goals and the effect of their actions
- How do such agents determine the sequence of actions that lead to the goal?
- *Problem-solving agents* are goal-based agents that use *search* to find action sequences
- The agent must formulate the search problem in terms of goals and actions before solving it

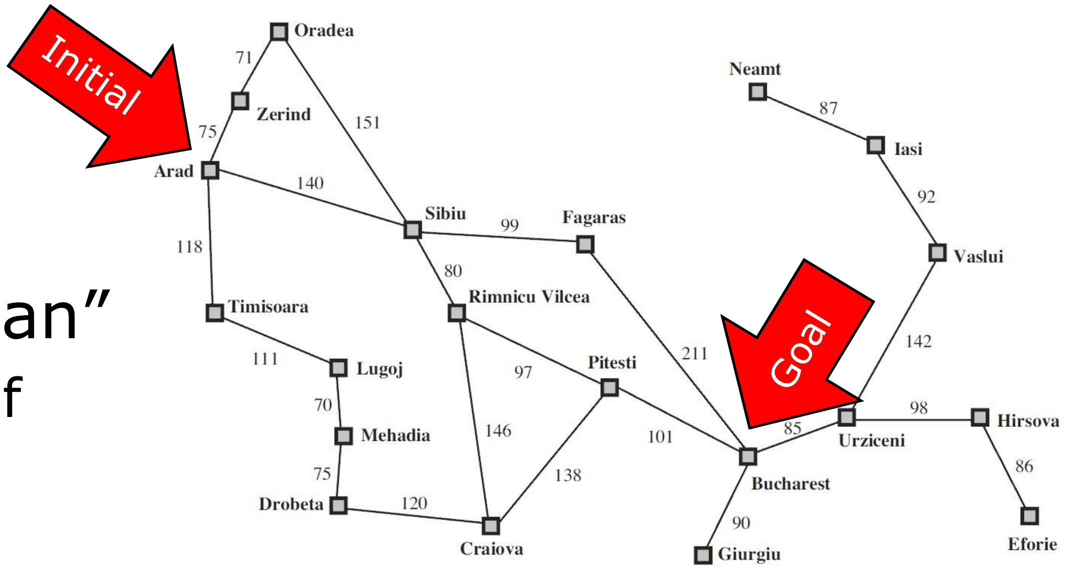
Formulation of a search problem

- Initial state
  - Initial state of environment
- Actions
  - Set of actions available to agent
- Path
  - Sequence of actions leading from one state to another
- Goal test
  - Test to check if a state is a goal state
- Path cost
  - Function that assigns cost to a path
- Solution
  - Path from initial state to a state that satisfies goal test

Defines the *state space*

Some real-world problems

- Route finding
  - E.g. airline or car travel planning
- "Traveling salesman"
  - E.g. movements of circuit board drills
- Robot navigation
  - Route finding in continuous space
- Automatic assembly sequencing
  - Synthesizing assembly operation sequences



Simple problem-solving agent

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **returns** an action

persistent: *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*)

    if *seq* = *failure* **then return** a null action

action <= FIRST(*seq*)

seq <= REST(*seq*)

return *action*

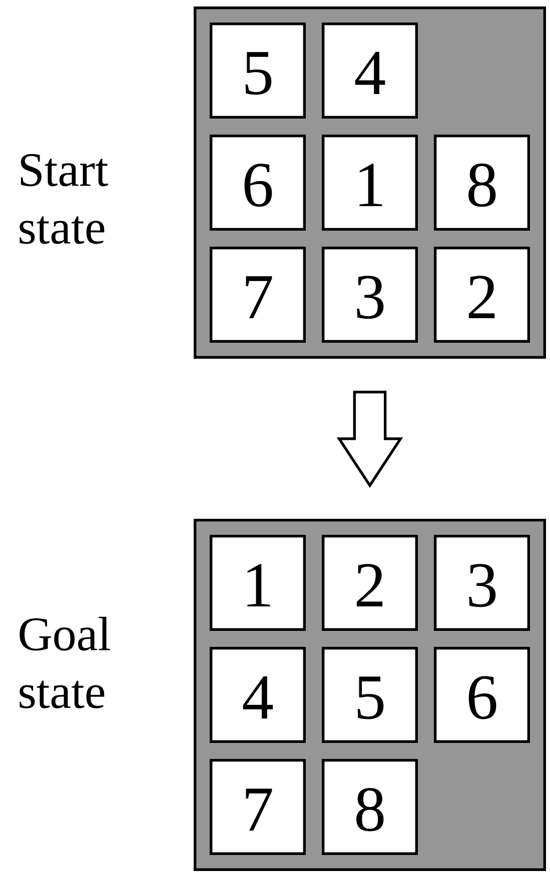
Implied environment properties

- Fully observable
  - √ Agent has full knowledge
- Deterministic
  - √ No surprises
- Static
  - √ No changes under deliberation
- Discrete
  - √ Discrete alternative actions

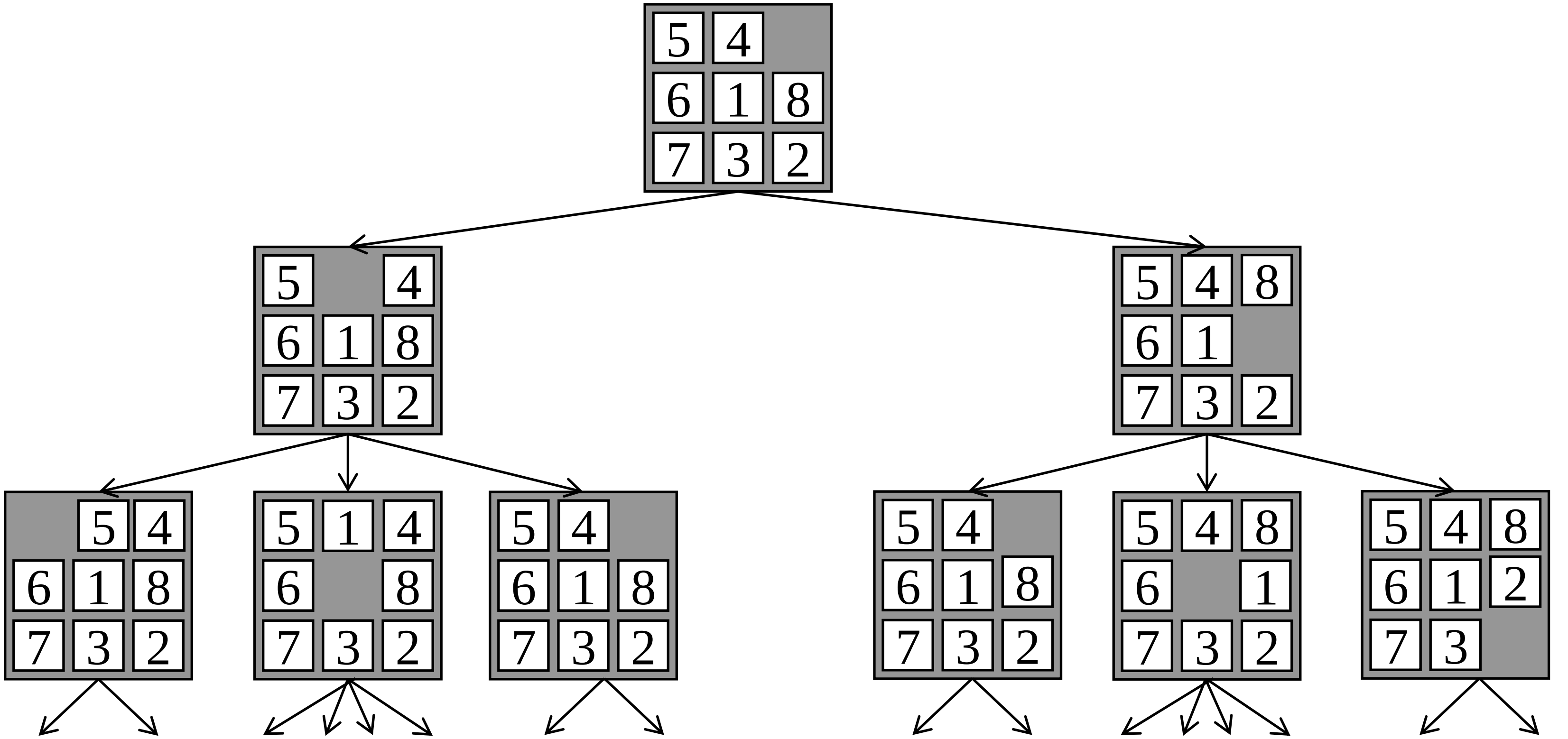
Simplest possible environment type!

Example toy problem: 8-puzzle

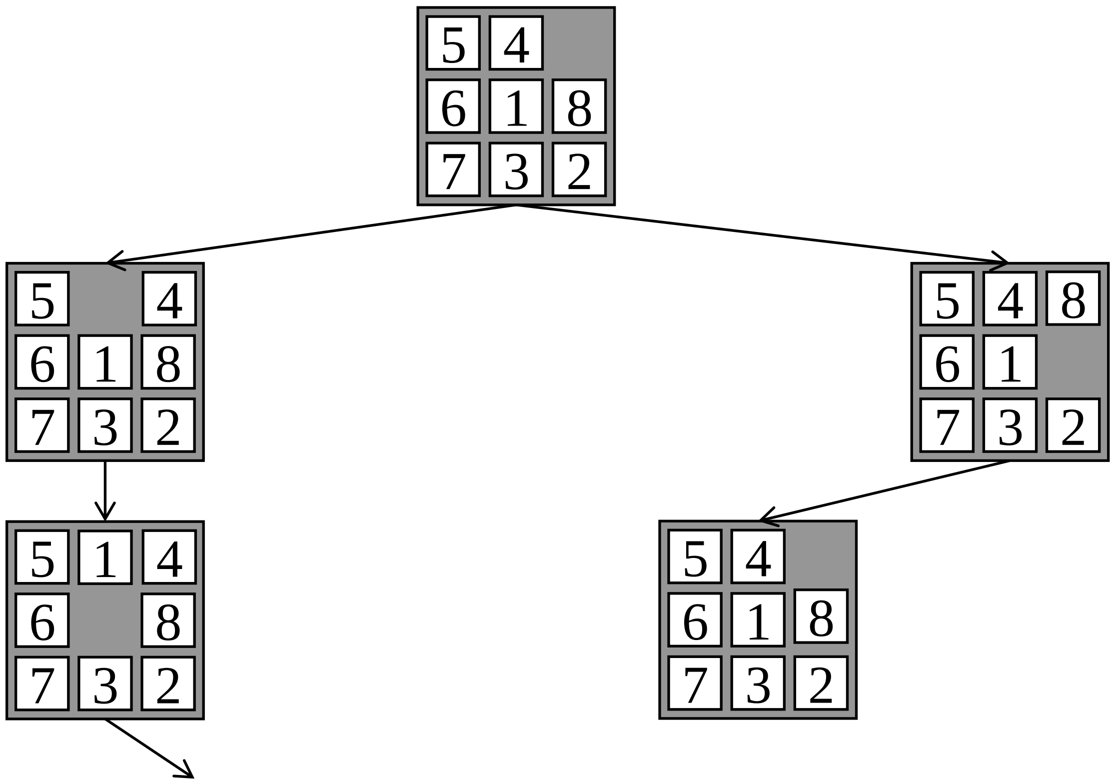
- States
  - Location of each tile
- Operators
  - Blank moves left, right, up, down
- Goal test
  - State matches goal configuration
- Path cost
  - Number of moves



Expanding a search tree fully …



… or partially or in different order



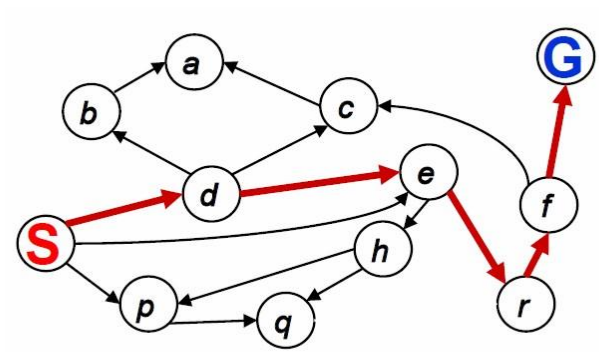
Search tree ≠ State space!

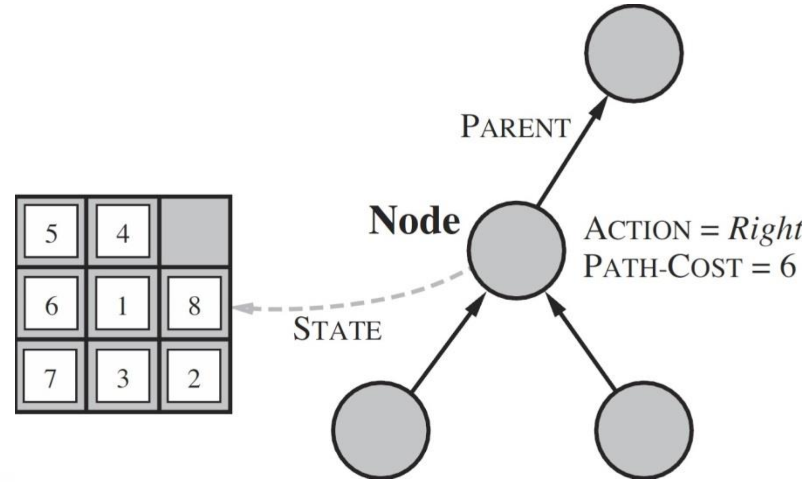
Searching for solutions

- The search starts in an *initial state*
- Thereafter, it iteratively explores the state space by selecting a state node and applying operators to generate *successor nodes*
- The choice of which node to expand at each level is determined by the *search strategy*
- The part of the state space that is explored is called the *search tree*

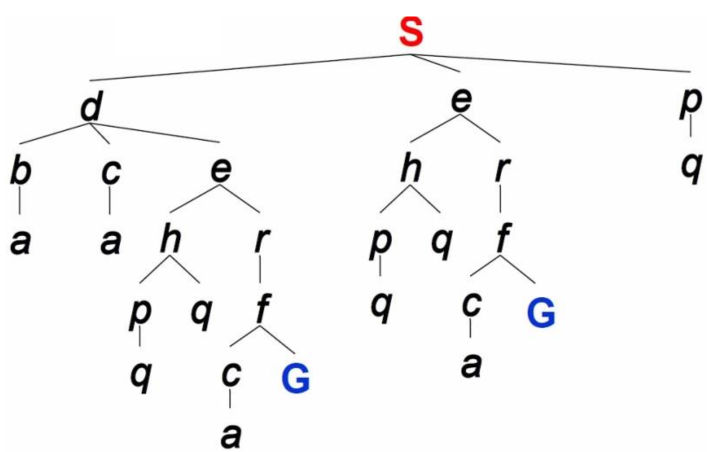
State space vs. search tree

State space





Search tree



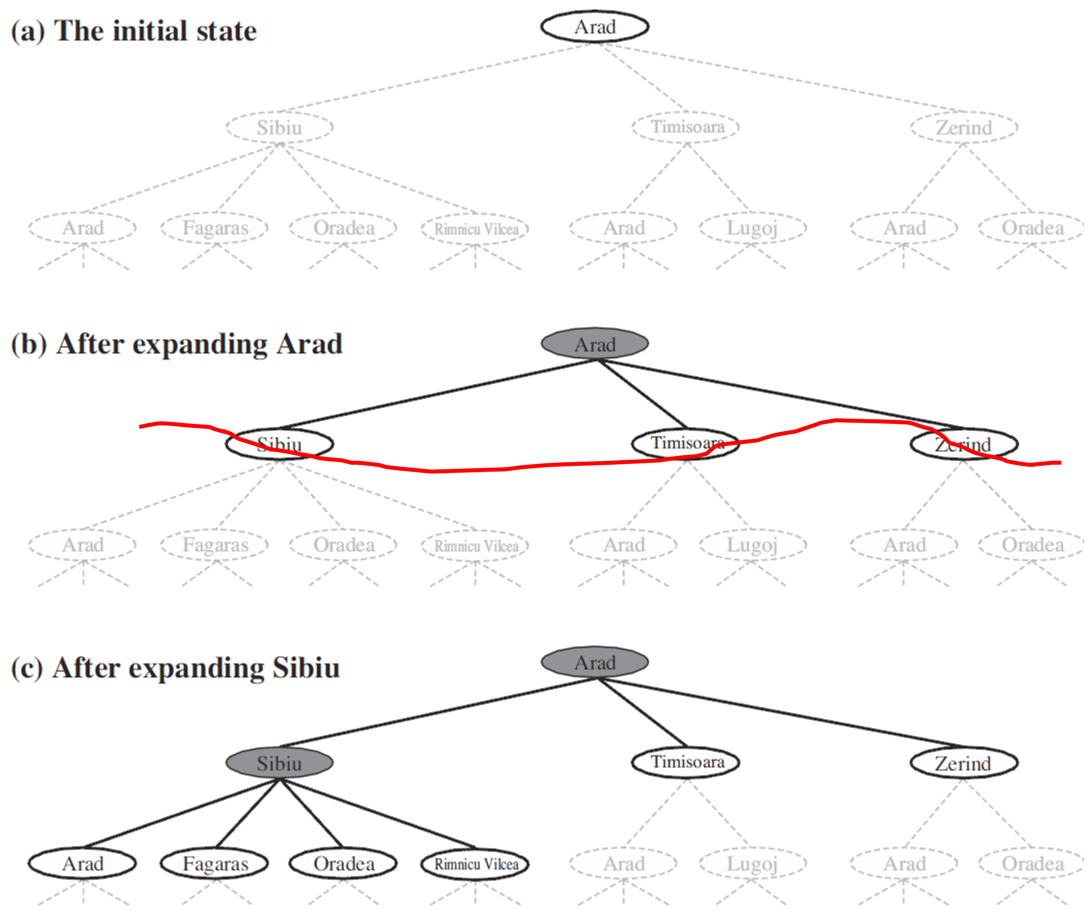
Node in a search tree points to a state in state space

Tree search vs. graph search

- The state space may contain *loops* (path back to earlier state) or *redundant paths* (more than one path between two states)
- Simple tree expansion will run infinitely or "explode" in such search spaces
- To avoid the problem, tree search can be replaced by generalized *graph search*
- In graph search, the algorithm keeps track and avoids expanding *already visited nodes*
- In the lecture, we will only study tree search

Tree search – General idea

- Start in initial state
- Expand possible nodes
- Keep a **frontier** of unexpanded nodes
- Select next node to expand according to strategy
- Continue until goal (or give up)



Uninformed search strategies

- Uninformed
  - √ No information on path cost from current to goal states
- Six uninformed strategies
  - √ Breadth-first
  - √ Uniform-cost
  - √ Depth-first
  - √ Depth-limited
  - √ Iterative deepening
  - √ Bidirectional
- Differ by *order* in which nodes are expanded

Evaluation of search strategies

- Completeness
  - √ Guaranteed to find a solution when there is one?
- Optimality
  - √ Finds the best solution when there are several different possible solutions?
- Time complexity
  - √ How long does it take to find a solution?
- Space complexity
  - √ How much memory is needed?

Data structures for search trees

- Datatype *node* with components:
  - √ STATE - search space state corresponding to the node
  - √ PARENT-NODE - node that generated this node
  - √ ACTION - action that was applied to generate this node
  - √ PATH-COST - cost of path from initial node (called *g*)
  - √ DEPTH - number of nodes on path from initial node
- Search tree nodes kept in a *queue* with operators:
  - √ MAKE-QUEUE(*Elements*) - create queue with given elements
  - √ EMPTY?(*Queue*) - true if no more elements in queue
  - √ FIRST(*Queue*) – returns first element of the queue
  - √ REMOVE-FIRST(*Queue*) - removes and returns first element
  - √ INSERT(*Element*, *Queue*) - inserts an element into queue
  - √ INSERT-ALL(*Elements*, *Queue*) - inserts set of elements into queue

General tree-search algorithm

function TREE-SEARCH(*problem*, *frontier*) **returns** a solution, or failure
frontier <= INSERT(MAKE-NODE*(problem*.INITIAL-STATE), *frontier*)
loop do
  if EMPTY?(*frontier*) **then return** failure
  node <= REMOVE-FIRST(*frontier)*
  if *problem*.GOAL-TEST applied to *node*.STATE succeeds
  then return SOLUTION(*node*)
  frontier <= INSERT-ALL(EXPAND(*node*,*problem*), *frontier*)

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

-*frontier* is an initially empty queue of a certain type (FIFO, etc.)

-SOLUTION returns sequence of actions back to root

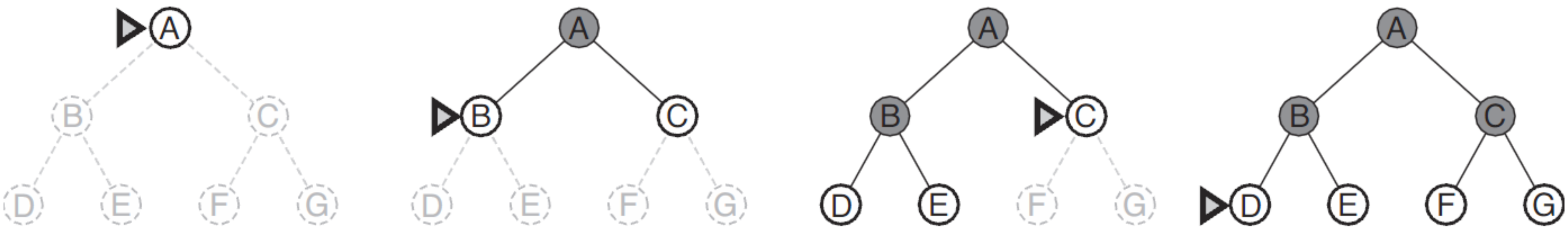
-EXPAND generates all successors of a node

Breadth-first search

function BREADTH-FIRST-SEARCH(*problem*)
returns a solution or failure
return TREE-SEARCH(*problem*, FIFO-QUEUE())

- FIFO – First In First Out (add nodes as last)
- Expands all nodes at a certain depth of search tree before expanding any node at next depth
- Exhaustive method - if there is a solution, breadth-first will find it (completeness)
- Will find the shortest solution first (optimal)

Breadth-first search illustrated



- All nodes on one level are explored before moving to next level

Complexity of breadth-first search

- Branching factor (b) - number of successors of each node (average)
- If solution is found at depth d, then max. number of nodes expanded is

$$1 + b + b^2 + b^3 + .. + b^d$$

- Exponential complexity ($O(b^d)$)
  - For b=10, 1000 nodes/sec, 100 bytes/node problem, time/memory increases from 1ms/100 bytes at depth 0 to 35 years/10 petabytes at depth 12 (10^{13} nodes)
- In general, we wish to avoid exponential search

Uniform-cost search

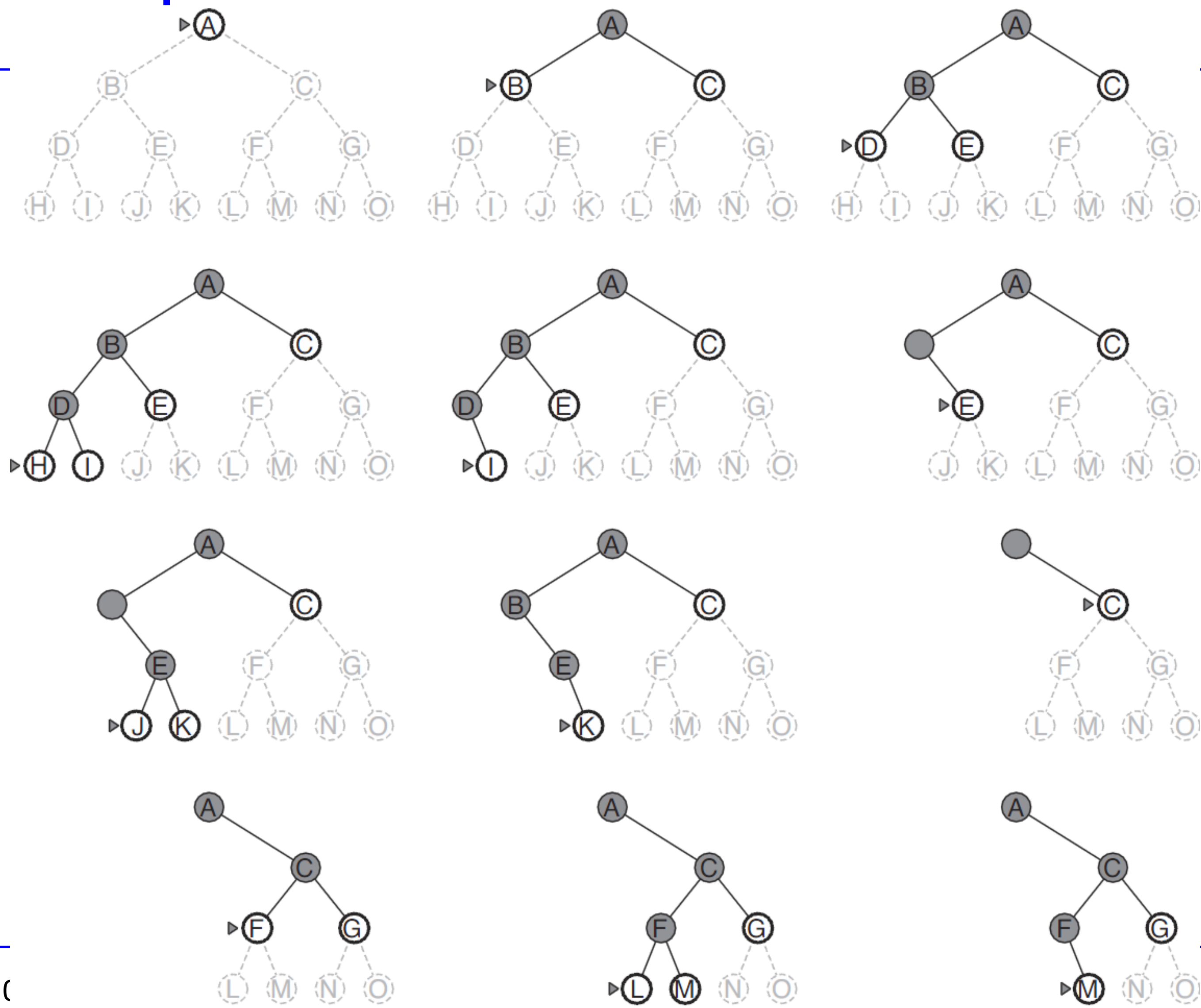
- Breadth-first is optimal because it always expands the *shallowest* unexpanded node
- Uniform-cost search expands the node *n* with *lowest path cost g(n)*
- This is done by storing the frontier as a priority queue ordered by *g*
- Uniform-cost search is optimal since it always expands the node with the lowest cost so far
- Completeness is guaranteed if all path costs>0

Depth-first search

function DEPTH-FIRST-SEARCH(problem)
returns a solution or failure
return TREE-SEARCH(*problem*, LIFO-QUEUE())

- LIFO – Last In First Out (add nodes as first)
- Always expands a node at deepest level of the tree, backtracks if it finds node with no successor
- May never terminate if it goes down an infinite branch, even if there is a solution (not complete)
- May return an early found solution even if a better one exists (not optimal)

Depth-first search illustrated



Complexity of depth-first search

- Depth-first has very low memory requirements, only needs to store one path from the root
- With branching factor *b* and depth *m*, space requirement is only *bm*.
  - √ For b=10, 100 bytes/node problem, memory increases from 100 bytes at depth 0 to 12 Kilobytes at depth 12
- Worst case time complexity is $O(b^m)$, but depth-first may find solution much quicker if there are many solutions (*m* may be much larger than *d* – the depth of the shallowest solution)

Depth-limited search

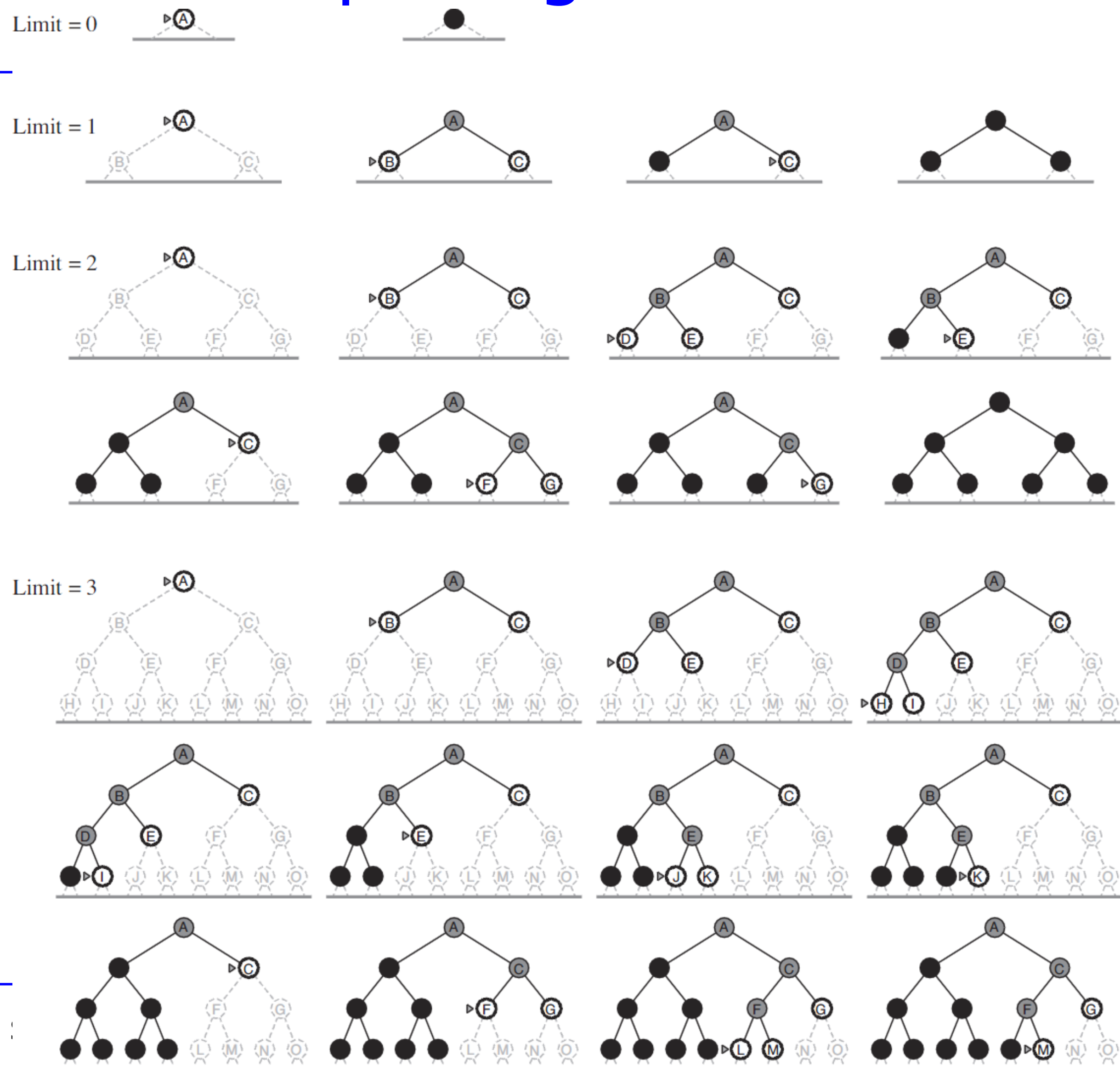
- Modifies depth-first search by imposing a cutoff on the maximum depth of a path
- Avoids risk of non-terminating search down an infinite path
- Finds a solution if it exists within cutoff limit (not generally complete)
- Not guaranteed to find shortest solution (not optimal)
- Time and space complexity as for depth-first

Iterative deepening search

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

- Modifies depth-limited search by iteratively trying all possible depths as the cutoff limit
- Combines benefits of depth-first and breadth-first

Iterative deepening search illustrated



Complexity of iterative deepening search

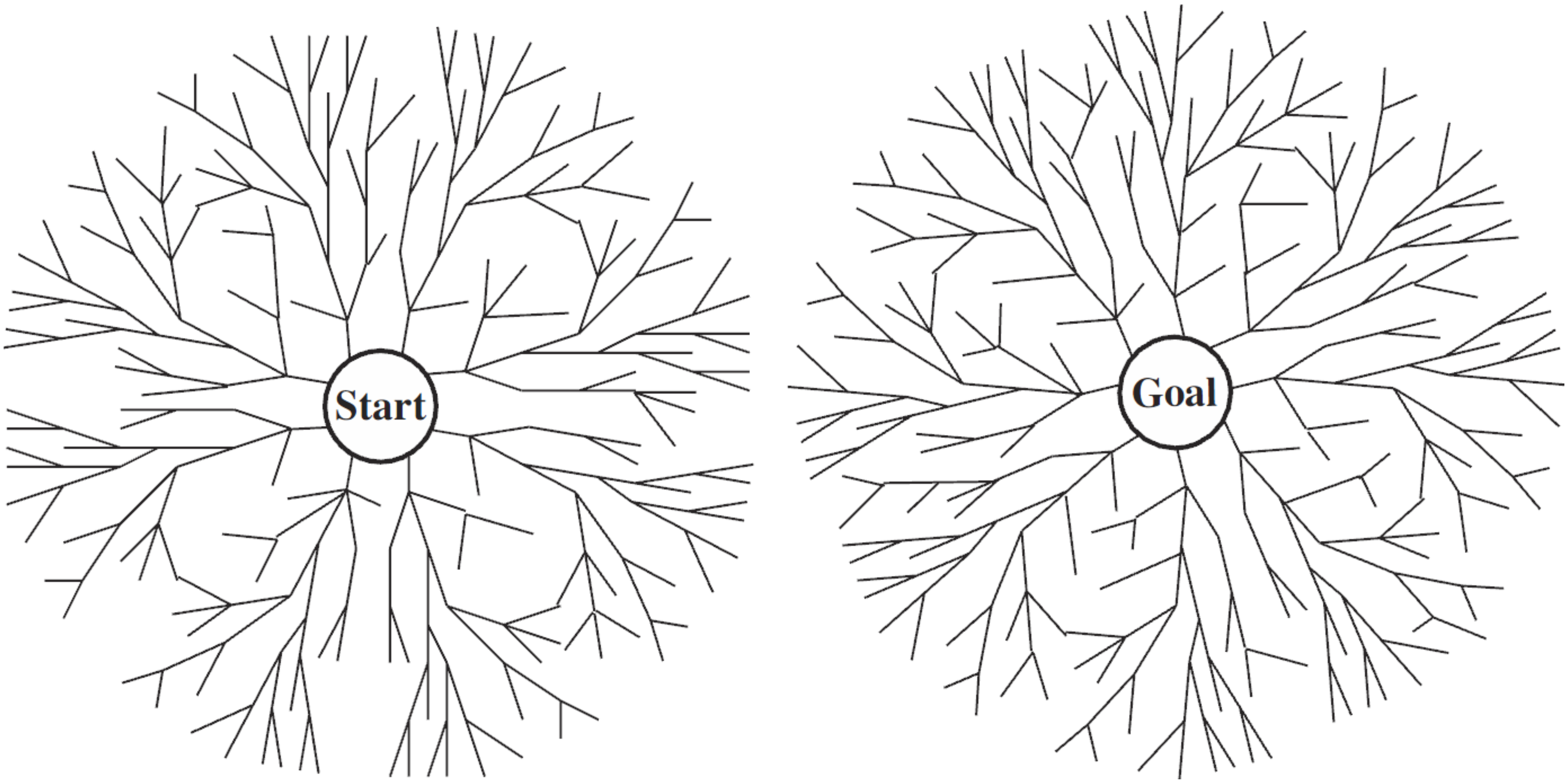
- May seem wasteful, since many states are expanded multiple times (for each cutoff limit)
- In exponential search trees most nodes are at lowest level, so multiple expansions at shallow depths do not matter much
- Time complexity is $O(b^d)$, space complexity $O(bd)$

Iterative deepening is the preferred (uninformed) search strategy when there is a large search space and the solution depth is unknown

Bidirectional search

- Searches simultaneously both forward from initial state and backward from goal state
- Time complexity reduced from $O(b^d)$ to $O(b^{d/2})$
  - E.g. for b=10, d=6, reduction from 1.1 mill nodes to 2.200
- But …
  - Does the node *predecessor* function exist?
  - What if there are many possible goals?
  - Must check a new node if it exists in other tree
- Must keep at least one tree, space complexity $O(b^{d/2})$

Bidirectional search illustrated



Comparing uninformed search strategies

Criterion	Breadth-first	Uniform-cost	Depth-first	Depth-limited	Iterative deepening	Bi-directional
Complete	Yes	Yes	No	No	Yes	Yes
Time	b^d	$b^{1+c/e}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	$b^{1+c/e}$	bm	bl	bd	$b^{d/2}$
Optimal	Yes	Yes	No	No	Yes	Yes

b - branching factor
d - depth of solution
c – cost of solution

m - maximum depth of tree
l - depth limit
e – cost of action

Informed search methods

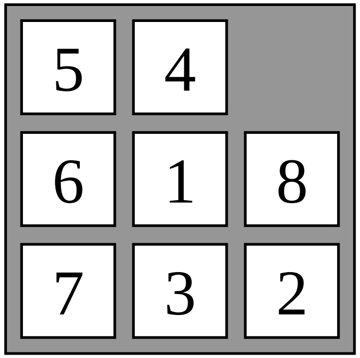
- Search can be improved by applying *knowledge* to better select which node to expand (*best-first*)
- An function to estimate the cost to reach a solution is called a *search heuristic* (h)
- *Greedy search*: Minimizes $h(n)$ - the estimated cost of the cheapest path from n to the goal
- Greedy search reduces search time compared to uninformed search, but is neither optimal nor complete

A* search

- A* - most widely known informed search method
- Identical to Uniform-Cost except that it minimizes *f(n)* instead of *g(n)*:
  - $g(n)$ - the cost of the path so far
  - $h(n)$ - the estimated cost of the remaining path to goal
  - $f(n) = g(n) + h(n)$
- Restriction: *h* must *never overestimate* the actual cost – i.e. *h* is “optimistic” (*admissible*)
- Properties of A*
  - Optimal (and optimally efficient)
  - Complete
  - Time/space exponential (space most severe problem)

Heuristic functions

- Some admissible *h* for 8-puzzle
  - *h1* – number of misplaced tiles
  - *h2* – sum of distances of tiles from their goal positions
  - Neither overestimate true cost



- Branching factor *b* of 8-puzzle approx. 3
- *Effective* branching factor *b** using A* depends on chosen heuristic function *h*
  - *h1* – effective *b** 1.79-1.48 (depending on *d*)
  - *h2* – effective *b** 1.79-1.26 (always better than *h1*)
- Dramatic reduction of search time/space compared to uninformed search

Summary

- An agent can use *search* when it is not clear which action to take
- The problem environment is represented by a *state space*
- A search problem consists of an *initial state*, a set of *actions*, a *goal test*, and a *path cost*
- A *path* from the initial to the goal state is a *solution*
- Search algorithms treat states and actions as *atomic* – do not consider internal structure
- General *tree search* considers all possible paths, while *graph search* avoids redundant paths

Summary (cont.)

- Properties of search algorithms
  - √ *completeness* – finds a solution if there is one
  - √ *optimality* – finds the best solution
  - √ *time complexity*
  - √ *space complexity*
- *Uninformed* search strategies have no information on cost to reach goal and include
  - √ *breadth-first* search
  - √ *uniform-cost* search
  - √ *depth-first* search
  - √ *depth-limited* search
  - √ *iterative-deepening* search
  - √ *bidirectional* search

Summary (cont.)

- *Informed search* uses knowledge on remaining cost to goal (*search heuristics)* to improve performance
- A* is a complete and optimal informed search algorithm that uses search heuristics
- Heuristic function *h* in A* must be *admissible,* and can greatly improve search performance