

# Foundations of Artificial Intelligence

## 3. Solving Problems by Searching

Problem-Solving Agents, Formulating Problems, Search Strategies

Joschka Boedecker and Wolfram Burgard and  
Frank Hutter and Bernhard Nebel and  
Michael Tangermann



Albert-Ludwigs-Universität Freiburg

Apr 26, 2019

- 1 Problem-Solving Agents
- 2 Formulating Problems
- 3 Problem Types
- 4 Example Problems
- 5 Search Strategies

# Lecture Overview

- 1 Problem-Solving Agents
- 2 Formulating Problems
- 3 Problem Types
- 4 Example Problems
- 5 Search Strategies

# Problem-Solving Agents

→ Goal-based agents

Formulation: *problem* as a *state-space* and *goal* as a *particular condition on states*

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

**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*  $\leftarrow$  UPDATE-STATE(*state*, *percept*)

**if** *seq* is empty **then**

*goal*  $\leftarrow$  FORMULATE-GOAL(*state*)

*problem*  $\leftarrow$  FORMULATE-PROBLEM(*state*, *goal*)

*seq*  $\leftarrow$  SEARCH(*problem*)

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

*action*  $\leftarrow$  FIRST(*seq*)

*seq*  $\leftarrow$  REST(*seq*)

**return** *action*

# Properties of this Agent

- Fully observable environment
- Deterministic environment
- Static environment
- Discrete states
- Single-agent setting

# Lecture Overview

- 1 Problem-Solving Agents
- 2 Formulating Problems
- 3 Problem Types
- 4 Example Problems
- 5 Search Strategies

# Problem Formulation

- Goal formulation  
World states with desired 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)



# Problem Formulation

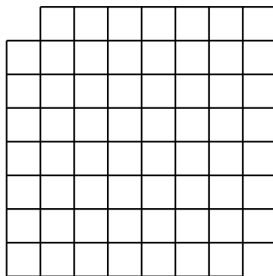
- Goal formulation  
World states with desired properties
- Definition of the state space  
(important: only the relevant aspects → abstraction)
- Definition of the actions that can change the world state
- Definition of the problem type, which depends on the knowledge of the world states and actions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

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

*Nuller ist machbar ist einfacher.*

# Example Problem Formulation

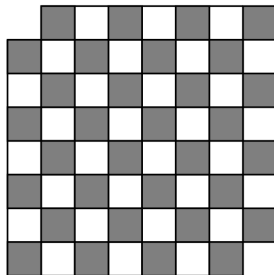
Given an  $n \times n$  board from which two diagonally opposite corners have been removed (here  $8 \times 8$ ):



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

→ Goal, state space, actions, search, ...

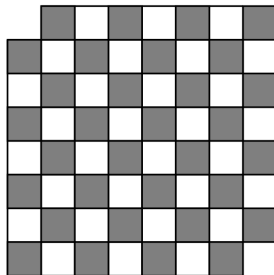
# Alternative Problem Formulation



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?

# Alternative Problem Formulation



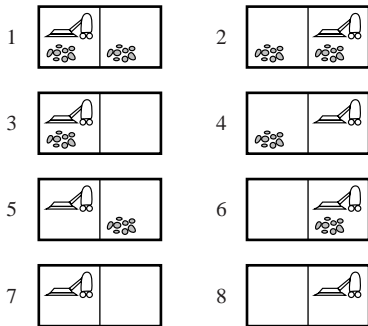
Question:

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



# Lecture Overview

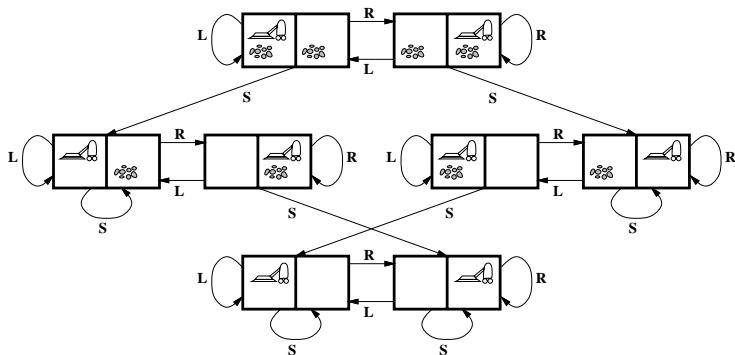
- 1 Problem-Solving Agents
- 2 Formulating Problems
- 3 Problem Types**
- 4 Example Problems
- 5 Search Strategies

# Problem Types: Knowledge of States and Actions

- State is completely observable  
Complete world state knowledge  
Complete action knowledge  
→ The agent always knows its world state
- State is partially observable  
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 the choice of the right action depends on information that will only be available at execution time.  
→ We need branching on observations
- Exploration problem  
State space and effects of actions unknown. Difficult!

# The Vacuum Cleaner Problem

If the environment is completely observable, 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 a goal state.



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

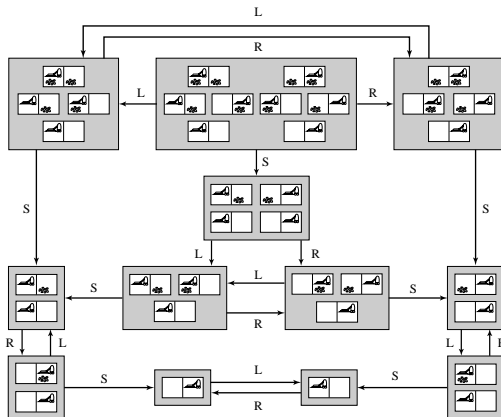


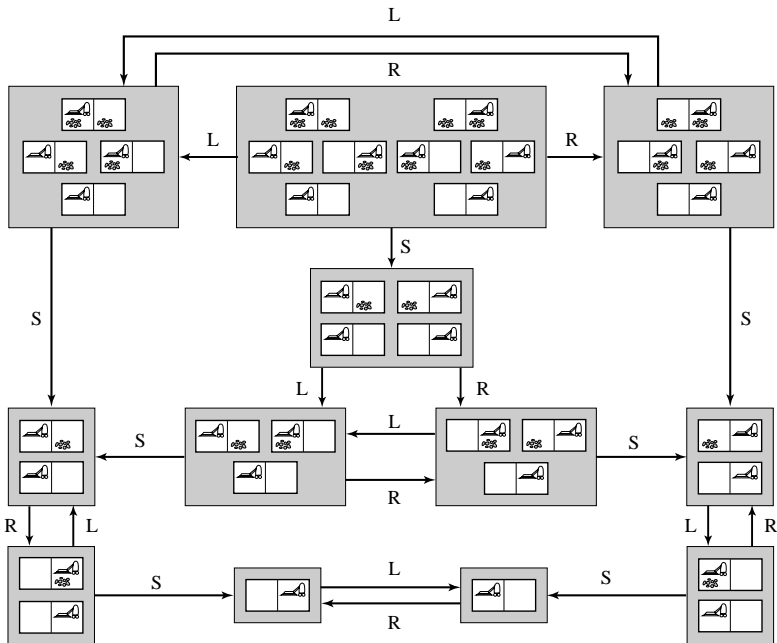
# The Vacuum Cleaner World as a Partially Observable State Problem

If the vacuum cleaner has no sensors, it does not 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.





# Concepts (1)

**Initial State:** The state the agent is in at the beginning

**State Space:** Set of all possible states

**Actions:** Description of possible actions. Available actions might be a function of the state.

**Transition Model:** Description of the outcome of an action (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 the initial to a goal state

**Search Costs:** Time and storage requirements to find a solution

**Total Costs:** Search costs + path costs

# Lecture Overview

- 1 Problem-Solving Agents
- 2 Formulating Problems
- 3 Problem Types
- 4 Example Problems**
- 5 Search Strategies

# Example: The 8-Puzzle

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

# Example: The 8-Puzzle

7	2	4
5		6
8	3	1

Start State

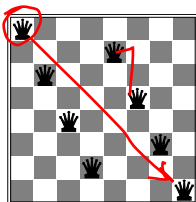
	1	2
3	4	5
6	7	8

Goal State

- **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** (transition model defined accordingly): 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

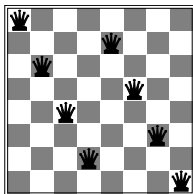
Almost a solution:





# Example: 8-Queens Problem

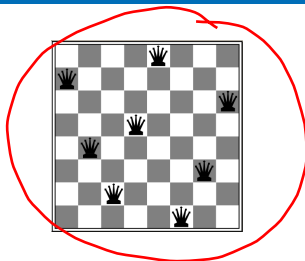
Almost a solution:



- **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 Formulation

- Naïve formulation

- States: any arrangement of the 0–8 queens
- State space:  $64 \times 63 \times \dots \times 57 \approx 10^{14}$  possible states

- Naïve formulation

- States: any arrangement of the 0–8 queens
- State space:  $64 \times 63 \times \dots \times 57 \approx 10^{14}$  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.
- State space: 2057 states

## 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:

$(3, 3, 1)$

$(0, 3, 0)$

Successor function: from each state, either bring one missionary and 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)

# Lecture Overview

- 1 Problem-Solving Agents
- 2 Formulating Problems
- 3 Problem Types
- 4 Example Problems
- 5 Search Strategies



From the initial state, produce all successive states step by step  $\rightarrow$  search tree.

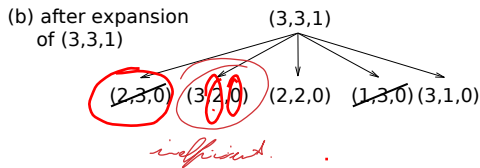
(a) initial state

(3,3,1)

# General Search

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

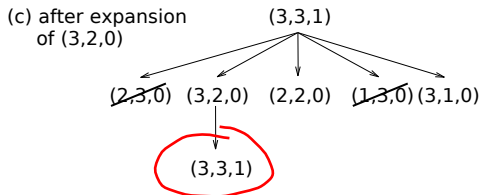
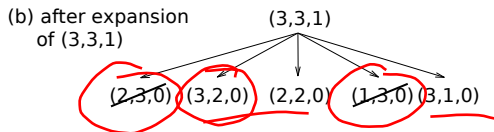
(a) initial state  $M \subset D$   
 $(3,3,1)$  *left bank*



# General Search

From the initial state, produce all successive states step by step  $\rightarrow$  search tree.

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



# Some notations

- **node expansion**  
generating all successor nodes considering the available actions
- **frontier**  
set of all nodes available for expansion
- **search strategy**  
defines which node is expanded next
- **tree-based search**  
it might happen, that within a search tree a state is entered repeatedly, leading even to infinite loops. To avoid this,
- **graph-based search** keeps a set of already visited states, the so-called explored set.

# Implementing the Search Tree

Data structure for each node  $n$  in the search tree:

$n$ .STATE: the state in the state space to which the node corresponds

$n$ .PARENT: the node in the search tree that generated this node

$n$ .ACTION: the action that was applied to the parent to generate the node

$n$ .PATH-COST: the cost, traditionally denoted by  $g(n)$  of the path from the initial state to the node, as indicated by the parent pointers

# Implementing the Search Tree

## Data structure for each node $n$ in the search tree:

- $n$ .STATE: the state in the state space to which the node corresponds
- $n$ .PARENT: the node in the search tree that generated this node
- $n$ .ACTION: the action that was applied to the parent to generate the node
- $n$ .PATH-COST: the cost, traditionally denoted by  $g(n)$ , of the path from the initial state to the node, as indicated by the parent pointers

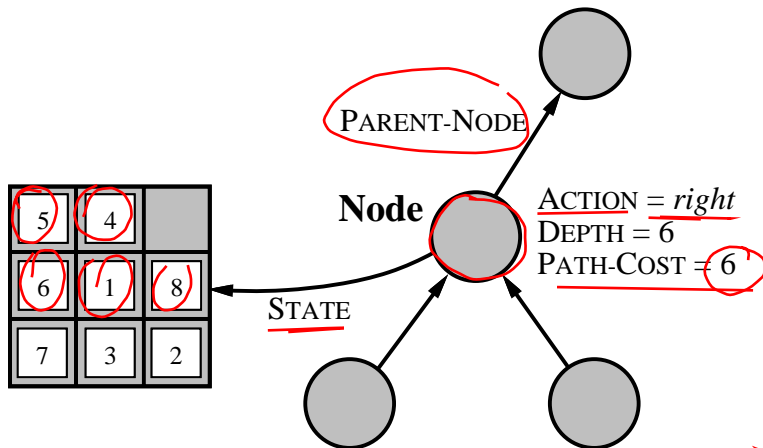
## Operations on a queue:

EMPTY?(*queue*): returns true only if there are no more elements in the queue

POP(*queue*): removes the first element of the queue and returns it

INSERT(*element*, *queue*): inserts an element (various possibilities) and returns the resulting queue

# Nodes in the Search Tree



# General Tree-Search Procedure

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

    initialize the frontier using the initial state of *problem*

**loop do**

**if** the frontier is empty **then return** failure

        choose a leaf node and remove it from the frontier

**if** the node contains a goal state **then return** the corresponding solution

        expand the chosen node, adding the resulting nodes to the frontier



# General Graph-Search Procedure

**function** GRAPH-SEARCH(*problem*) **returns** a solution, or failure

initialize the frontier using the initial state of *problem*

*initialize the explored set to be empty*

**loop do**

**if** the frontier is empty **then return** failure

→ choose a leaf node and remove it from the frontier

**if** the node contains a goal state **then return** the corresponding solution

*add the node to the explored set*

expand the chosen node, adding the resulting nodes to the frontier

→ *only if not in the frontier or explored set*

# 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)?

- **problem describing quantities**

b: branching factor

d: depth of shallowest goal node

m: maximum length of any path in the state space

## 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.

## Uninformed or blind searches

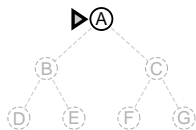
No information on the length or cost of a path to the solution.

- 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  
(*frontier*  $\leftarrow$  a FIFO queue).



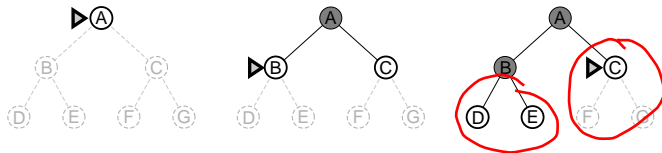
# Breadth-First Search (1)

Nodes are expanded in the order they were produced (*frontier*  $\leftarrow$  a FIFO queue).



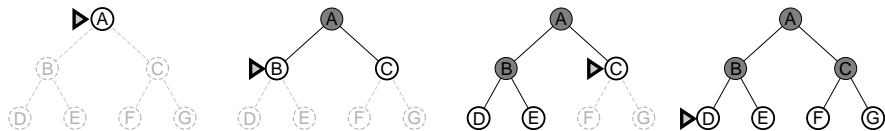
# Breadth-First Search (1)

Nodes are expanded in the order they were produced (*frontier*  $\leftarrow$  a FIFO queue).



# Breadth-First Search (1)

Nodes are expanded in the order they were produced (*frontier*  $\leftarrow$  a 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)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure  
node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0  
if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)  
frontier ← a FIFO queue with node as the only element  
explored ← an empty set  
loop do  
  if EMPTY?(frontier) then return failure  
  node ← POP(frontier) /* chooses the shallowest node in frontier */  
  add node.STATE to explored  
  for each action in problem.ACTIONS(node.STATE) do  
    child ← CHILD-NODE(problem, node, action)  
    if child.STATE is not in explored or frontier then  
      if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)  
      frontier ← INSERT(child, frontier)
```

# Breadth-First Search (3)

## Time Complexity:

Let  $b$  be the maximal branching factor and  $d$  the depth of a solution path. Then the maximal number of nodes expanded is

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

(Note: If the algorithm were to apply the goal test to nodes when selected for expansion rather than when generated, the whole layer of nodes at depth  $d$  would be expanded before the goal was detected and the time complexity would be  $O(b^{d+1})$ )

# Breadth-First Search (3)

## Time Complexity:

Let  $b$  be the maximal branching factor and  $d$  the depth of a solution path. Then the maximal number of nodes expanded is

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

(Note: If the algorithm were to apply the goal test to nodes when selected for expansion rather than when generated, the whole layer of nodes at depth  $d$  would be expanded before the goal was detected and the time complexity would be  $O(b^{d+1})$ )

## Space Complexity:

Every node generated is kept in memory. Therefore space needed for the frontier is  $O(b^d)$  and for the explored set  $O(b^{d-1})$ .

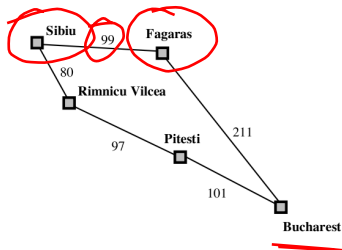
# Breadth-First Search (4)

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

Depth	Nodes	Time		Memory	
2	110	.11	milliseconds	107	kilobyte
4	11,110	11	milliseconds	10,6	megabytes
6	$10^6$	1,1	seconds	1	gigabytes
8	$10^8$	2	minutes	103	gigabyte
10	$10^{10}$	3	hours	10	terabytes
12	$10^{12}$	13	days	1	petabytes
14	$10^{14}$	3.5	years	99	petabyte

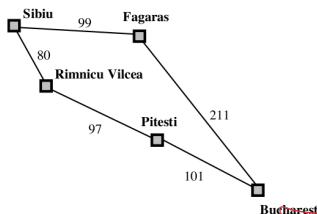
# Uniform-Cost Search

- If step costs for doing an action are equal, then breadth-first search finds path with the optimal costs.
- If step costs are different (e.g., map: driving from one place to another might differ in distance), then uniform-cost search is a means to find the optimal solution.
- Uniform-cost search expands the node with the lowest path costs  $g(n)$ .  
Realization: priority queue.



# Uniform-Cost Search

- If step costs for doing an action are equal, then breadth-first search finds path with the optimal costs.
- If step costs are different (e.g., map: driving from one place to another might differ in distance), then uniform-cost search is a means to find the optimal solution.
- Uniform-cost search expands the node with the lowest path costs  $g(n)$ . Realization: priority queue.



*Cannot have negative values.*

Always finds the cheapest solution, given that  $g(\text{successor}(n)) \geq g(n)$  for all  $n$ .

# Depth-First Search (1)

Always expands an unexpanded node at the greatest depth  
(*frontier*  $\leftarrow$  a LIFO queue).

It is common to realize depth-first search as a recursive function

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

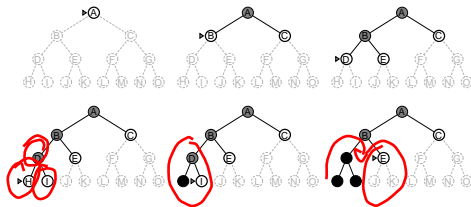


# Depth-First Search (1)

Always expands an unexpanded node at the greatest depth  
(*frontier*  $\leftarrow$  a LIFO queue).

It is common to realize depth-first search as a recursive function

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





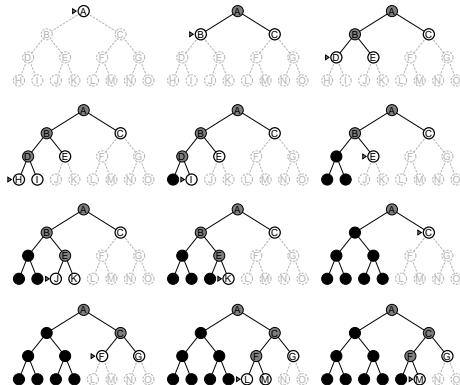


# Depth-First Search (1)

Always expands an unexpanded node at the greatest depth  
( $frontier \leftarrow$  a LIFO queue).

It is common to realize depth-first search as a recursive function

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



## Depth-First Search (2)

- In general, the solution found is not optimal.
- **Completeness** can be guaranteed only for graph-based search and finite state spaces.
- Algorithm: see later (depth-limited search)

# Depth-First Search (3)

## Time Complexity:

- in graph-based search bounded by the size of the state space (might be infinite!);
- in tree-based search, algorithm might generate  $O(b^m)$  nodes in the search tree which might be much larger than the size of the state space. ( $m$  is the maximum length of a path in the state space).

*maximal path length*

# Depth-First Search (3)

## Time Complexity:

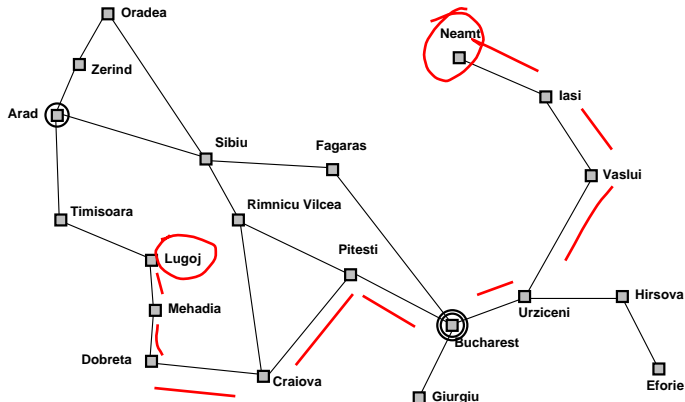
- in graph-based search bounded by the size of the state space (might be infinite!);
- in tree-based search, algorithm might generate  $O(b^m)$  nodes in the search tree which might be much larger than the size of the state space. ( $m$  is the maximum length of a path in the state space).

## Space Complexity:

- tree-based search: needs to store only the nodes along the path from the root to the leaf node. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored. Therefore, memory requirement is only  $O(bm)$ . This is the reason, why it is practically so relevant despite all the other shortcomings!
- graph-based search: in worst case, all states need to be stored in the explored set (no advantage over breadth-first)

# Depth-Limited Search (1)

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  $n - 1$ .



Sometimes, the search depth can be refined. E.g., here, a depth of 9 is sufficient (you can reach every city in at most 9 steps).

## Depth-Limited Search (2)

**function** DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a solution, or failure/cutoff  
**return** RECURSIVE-DLS(MAKE-NODE(*problem*.INITIAL-STATE), *problem*, *limit*)

**function** RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution, or failure/cutoff  
**if** *problem*.GOAL-TEST(*node*.STATE) **then** **return** SOLUTION(*node*)

**else if** *limit* = 0 **then** **return** cutoff

**else**

*cutoff\_occurred?*  $\leftarrow$  false

**for each** *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

*child*  $\leftarrow$  CHILD-NODE(*problem*, *node*, *action*)

*result*  $\leftarrow$  RECURSIVE-DLS(*child*, *problem*, *limit* - 1)

**if** *result* = cutoff **then** *cutoff\_occurred?*  $\leftarrow$  true

**else if** *result*  $\neq$  failure **then** **return** *result*

**if** *cutoff\_occurred?* **then** **return** cutoff **else** **return** failure

# Iterative Deepening Search (1)

- Idea: use depth-limited search and in every iteration increase search depth by one.
- Looks a bit like a waste of resources (since the first steps are always repeated), but complexity-wise it is not so bad as it might seem.
- Combines depth- and breadth-first searches.
- Optimal and complete like breadth-first search, but requires much less memory:  $O(b d)$ .
- Time complexity is only little bit worse than breadth-first search (see later).

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result
```

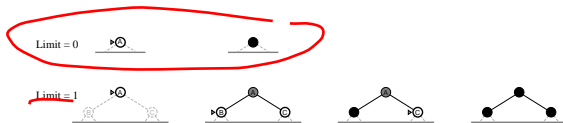


# Example

Limit = 0



# Example



# Example

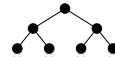
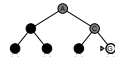
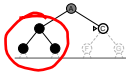
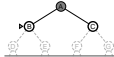
Limit = 0



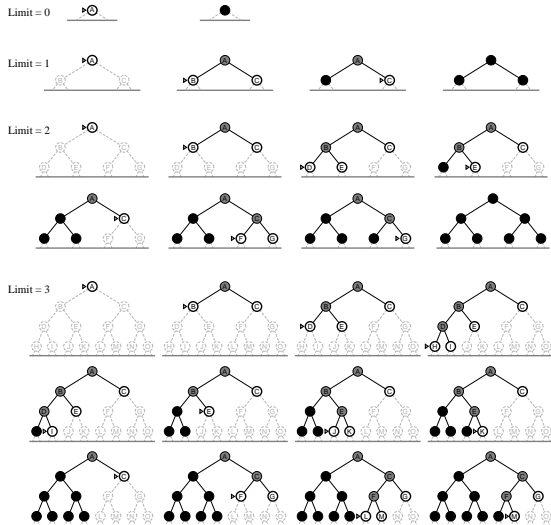
Limit = 1



Limit = 2



# Example



# Iterative Deepening Search (2)

Number of expansions

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

Example:  $b = 10, d = 5$

<u>Breadth-First-Search</u>	$10 + 100 + 1,000 + 10,000 + 100,000$ $= 111,110$
Iterative Deepening Search	$50 + 400 + 3,000 + 20,000 + 100,000$ $= 123,450$

For  $b = 10$ , IDS expands only 11% more than the number of nodes expanded by (optimized) breadth-first-search.

# Iterative Deepening Search (2)

Number of expansions

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

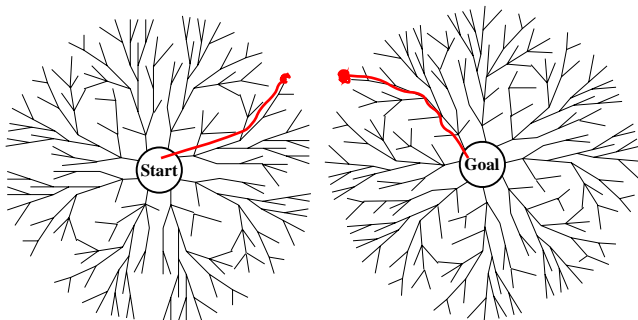
Example:  $b = 10$ ,  $d = 5$

Breadth-First-Search	$10 + 100 + 1,000 + 10,000 + 100,000$ $= 111,110$
Iterative Deepening Search	$50 + 400 + 3,000 + 20,000 + 100,000$ $= 123,450$

For  $b = 10$ , IDS expands only 11% more than the number of nodes expanded by (optimized) breadth-first-search.

→ *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 forward and backward 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 1,111,110 only 2,220 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 <sup>a</sup>	Yes <sup>a,b</sup>	No	No	Yes <sup>a</sup>	Yes <sup>a,d</sup>
Time	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes <sup>c</sup>	Yes	No	No	Yes <sup>c</sup>	Yes <sup>c,d</sup>

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

### Superscripts:

- <sup>a</sup>  $b$  is finite  
<sup>b</sup> if step costs not less than  $\epsilon$   
<sup>c</sup> if step costs are all identical  
<sup>d</sup> 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.