

Lecture

Foundations of Artificial Intelligence

Part 4 – Uninformed Search

Dr. Mohsen Mesgar

Universität Duisburg-Essen

Recall...

- What's the **task/goal?** **PEAS description**
- **Performance measures**
- **Environment variables**
- **Agents**
 - **Simple reflex agent**
 - **Model-based agent**
 - **Goal-based agent**
 - **Utility-based agent**
 - **Learning-based agent**



Any other open questions?



Goal: At the end of this lecture, you learn about

Problem-solving agents

Tree-search algorithms

Breadth-first search

Depth-first search

Iterative deepening

Performance of a search algorithm

Problem-solving agents

Problem-solving agents

- special case of **goal-based agents**
- find sequences of actions that lead to desirable states

Goal-based agents

- can consider future actions and the desirability of their outcomes

Problem-solving agents

Uninformed problem-solving agents

- do **not have any information** except the problem definition

Informed problem-solving agents

- **have knowledge** where to look for solutions



Road trip through Romania

Task: find route from Arad to Bucharest

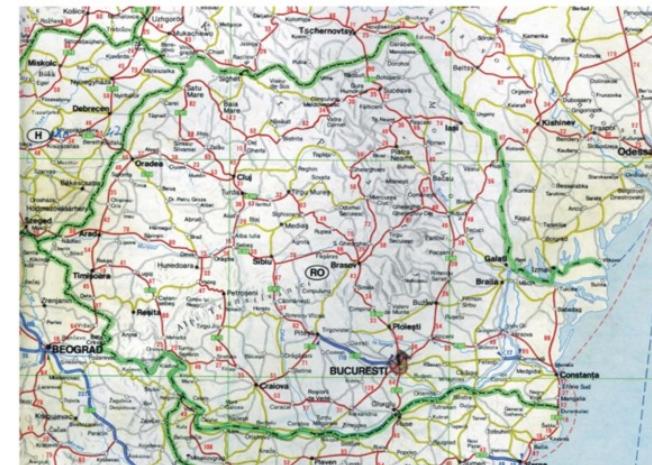
Assumption: agent has a map of Romania

1. Formulate goal

- be in Bucharest

2. Formulate environment (problem)

- **states:** various cities
- **actions:** drive between cities



3. Agent's program: Search

- Return a sequence of cities, e.g. [Arad, Sibiu, Rimnicu, Vilcea, Pitesti, Bucharest]

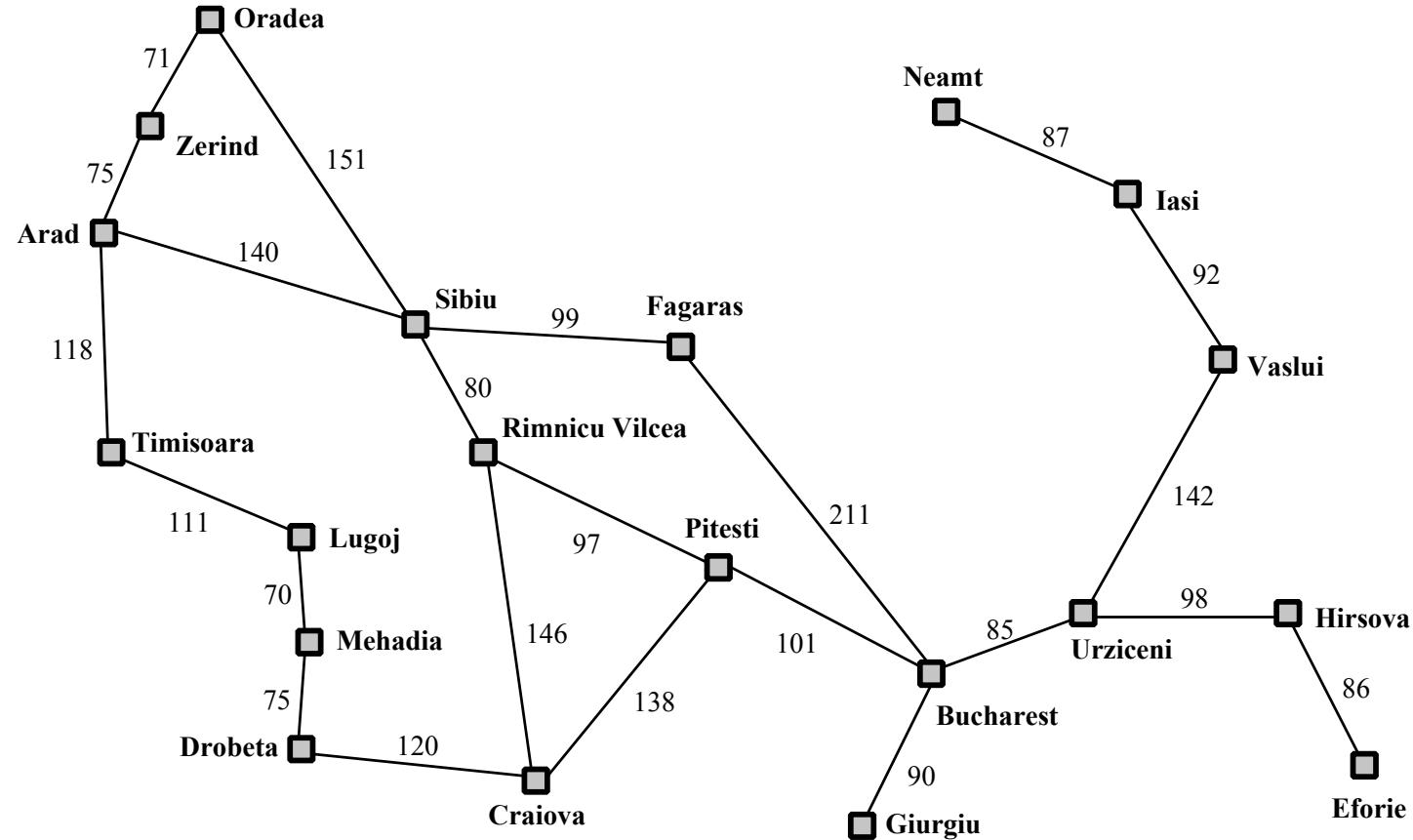
Environment

- environment is **static, fully observable, discrete**, and **deterministic**
- **easiest** possible environment
- once we have a plan, we can execute it “with eyes closed”
- the solution to any problem is a fixed sequence of actions

This is an Abstraction

- Real-world conditions are too complex.
 - Quality of the trip depends on
 - harmony with travel companions
 - weather
 - scenery
 - radio program
 - condition of roads
 - food supply
- Leads to infeasibly large search space.
- **For finding a rational solution, we need to operate on an abstraction of the problem.**

Abstracted formulation



Problem formulation

States

- $\text{in}(x)$, x corresponds to all possible cities

Initial state

- start in a specific city: e.g. $\text{in}(\text{Arad})$

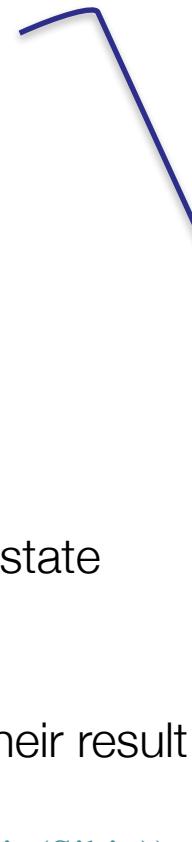
Actions

- $\text{go}(x)$
- only a limited set of actions is applicable in each state

Successor function

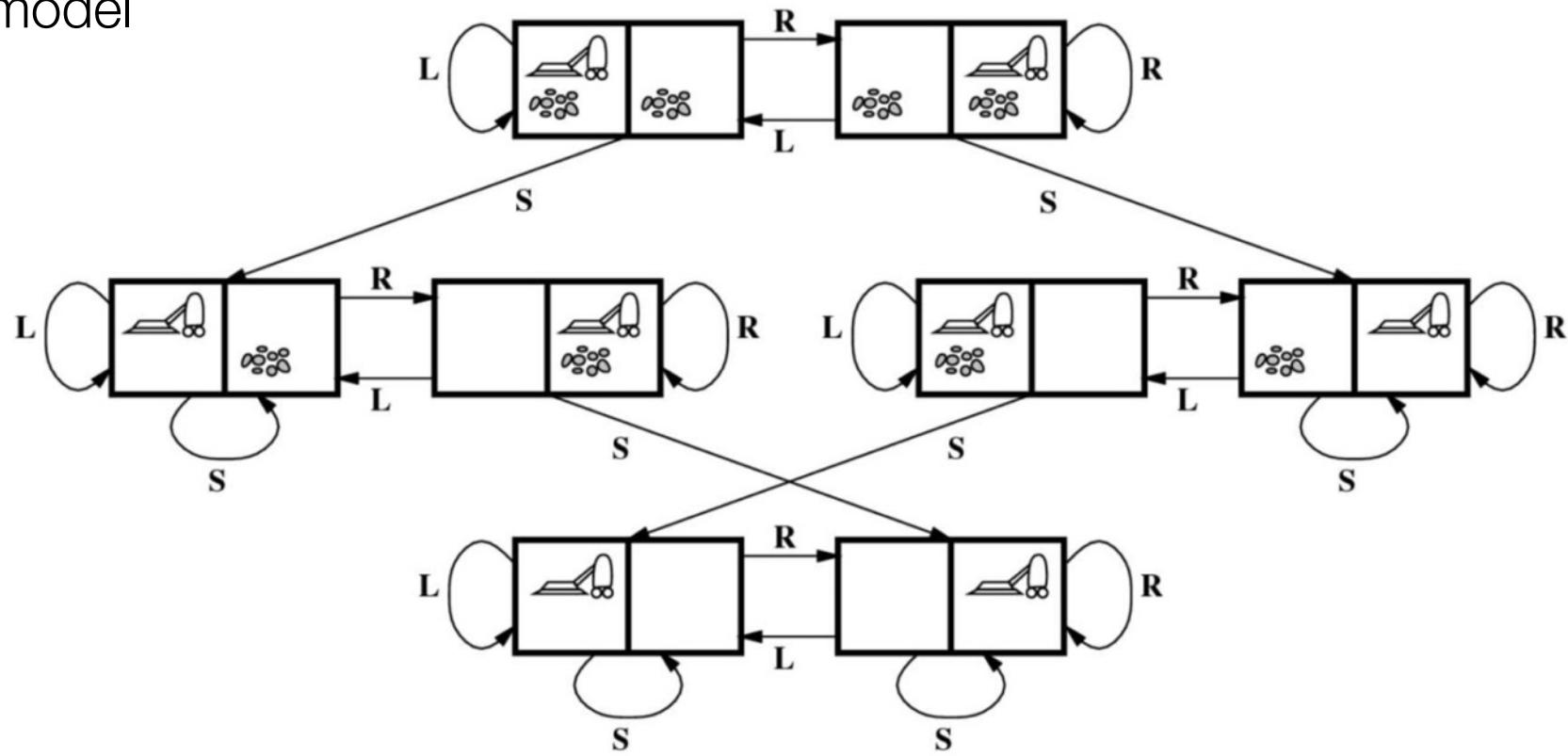
- models the applicable actions for each state and their result

$\text{Successors}(\text{in}(\text{Arad})) = \{\text{(go}(\text{Zerind}), \text{in}(\text{Zerind})), \text{(go}(\text{Sibiu}), \text{in}(\text{Sibiu})), \text{(go}(\text{Timisoara}), \text{in}(\text{Timisoara}))\}$



State space

- The **set of all states reachable from the initial state**
- Implicitly defined by the **initial state**, the actions, and the transition model



Search parameters

Goal test

- explicit goal state: `in(Bucharest)`
- abstract property: `checkmate`

Path

- a sequence of states connected by a sequence of actions

Path cost

- cumulated cost of actions connecting a sequence of states
- e.g.: `sum of distances`, `driving time`
- cost depends on performance measure

Solution

- a path that leads from the initial state to a goal state

Optimal solution

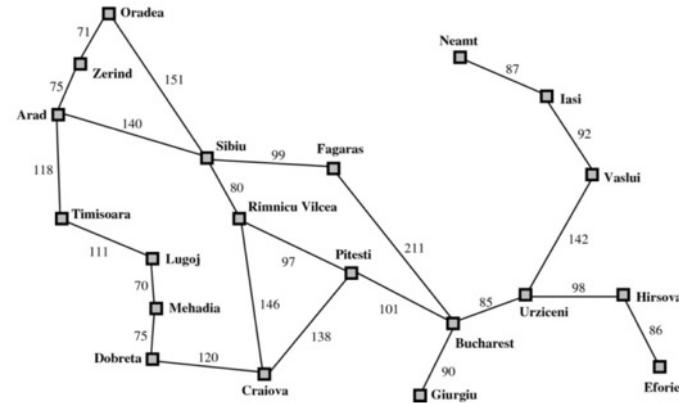
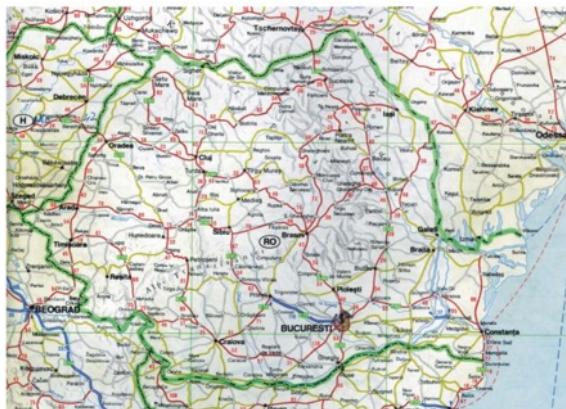
- solution with the minimum path cost

Recall!

Abstract actions correspond to a complex combination of real actions

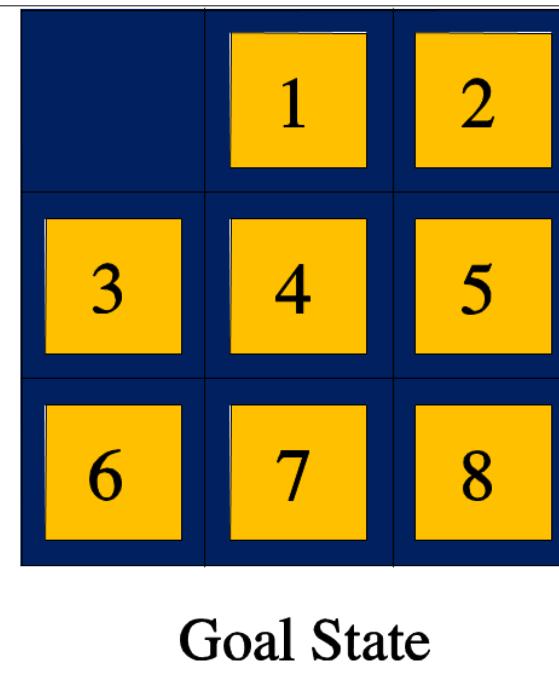
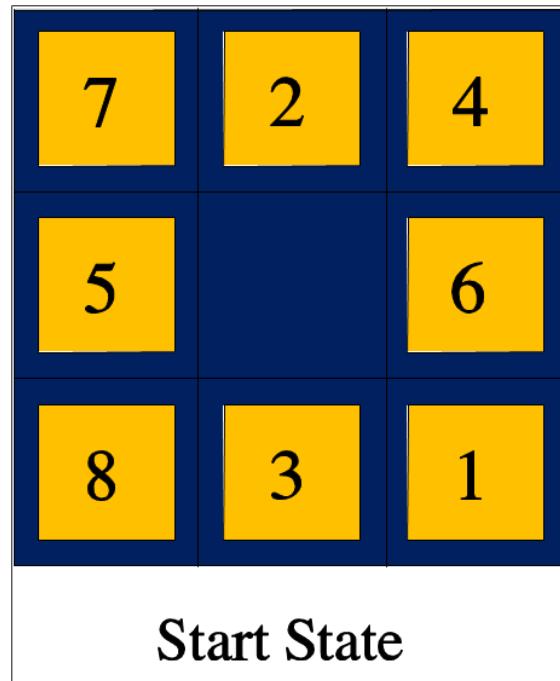
- e.g. go(Bucharest) represents a complex set of possible routes, detours, rest stops, etc.
- **abstract actions** are a simplification of the original actions

An **abstract solution** corresponds to a set of **real paths** that are **solutions in the real world**.



Toy Problems

Example: The 8-puzzle



Search configuration of the 8-puzzle

States

- location of tiles

Initial state

- any configuration of tiles

Successor function

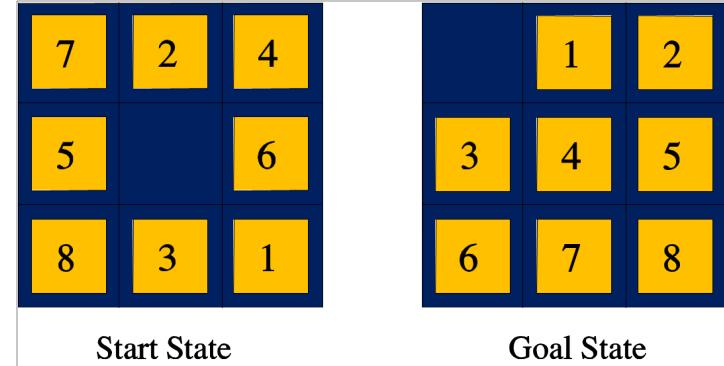
- **actions:** move blank tile left, right, up, down
- efficient, but counterintuitive: move blank left \leadsto move 5 right
- **result state:** new configuration of tiles

Goal test

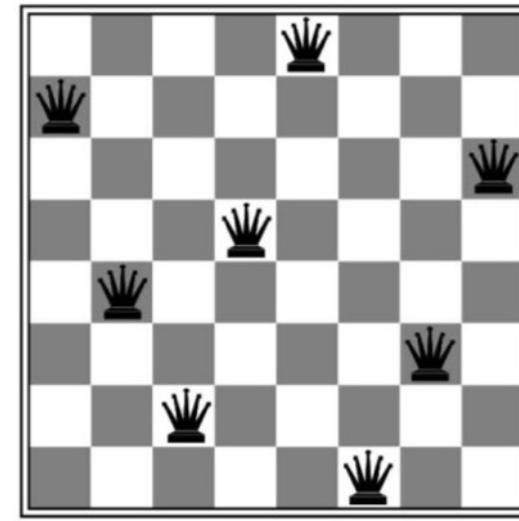
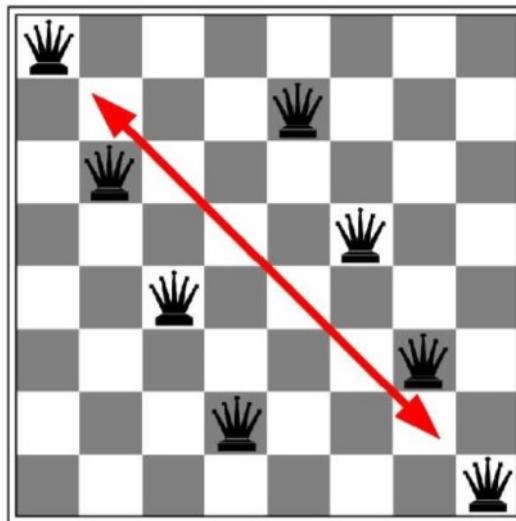
- configuration equals goal state

Path cost

- number of steps in path



Example: The 8-Queens Problem



Naïve incremental approach

States

- 0-8 queens are positioned on the board

Initial state

- no queens on board

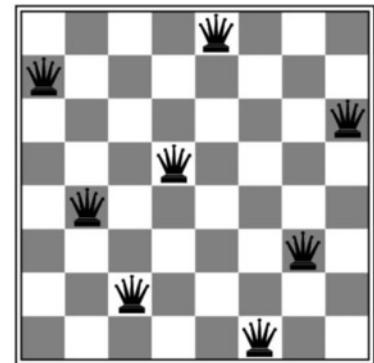
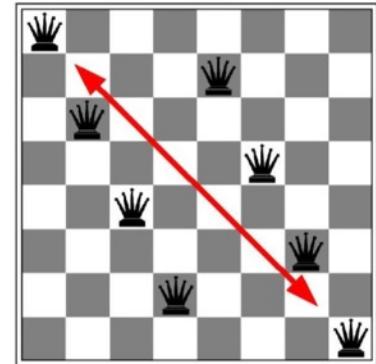
Successor function

- **action:** add queen to any empty square
- **result state:** new configuration of queens

Goal test

- 8 queens on board, none attacked

→ inefficient: $64 \cdot 63 \cdot \dots \cdot 57 \approx 1.8 \cdot 10^{14}$ states



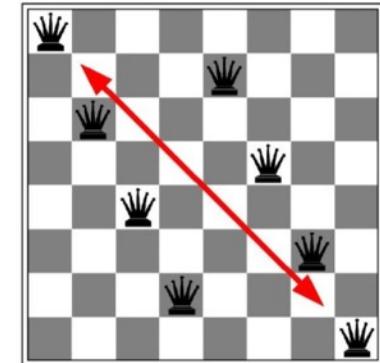
Less naïve approach

States

- any configuration of 0 to 8 queens in the leftmost columns,
no queen attacking each other

Initial state

- no queens on board

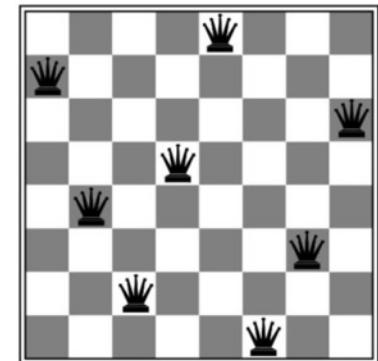


Successor function

- action: add queen to leftmost empty column such that it is not attacked**
- result state: new configuration of queens

Goal test

- 8 queens on board, none attacked
- reduction to 2057 possible states



Tree Search Algorithms

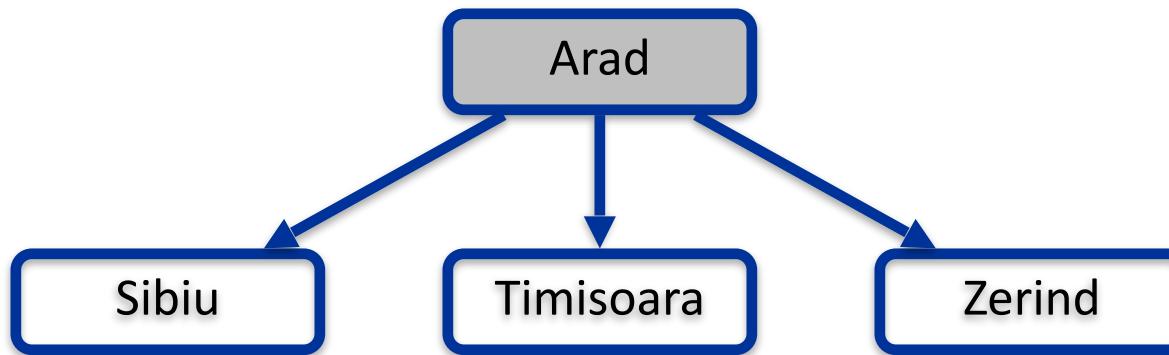
Search Tree

Treat the state-space graph as a tree

- **root:** initial state
- **successor nodes:** result states after applying possible actions

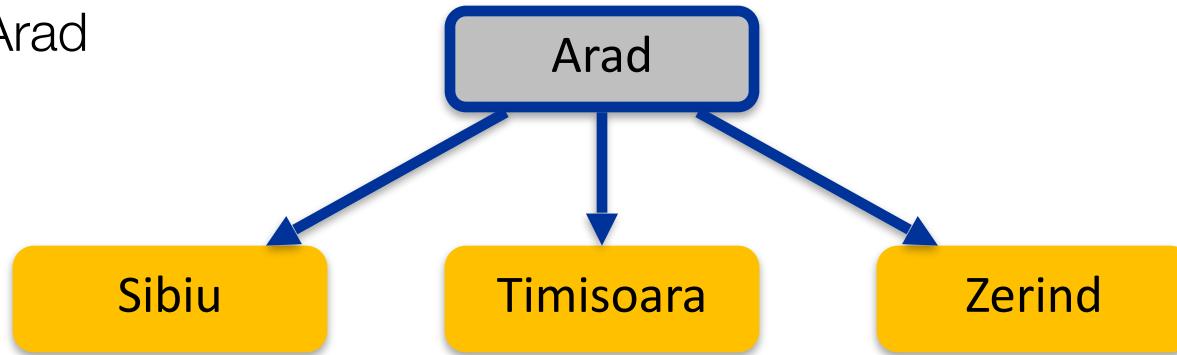
Search

- expand the successor nodes
- **search strategy:** decide which nodes to expand next



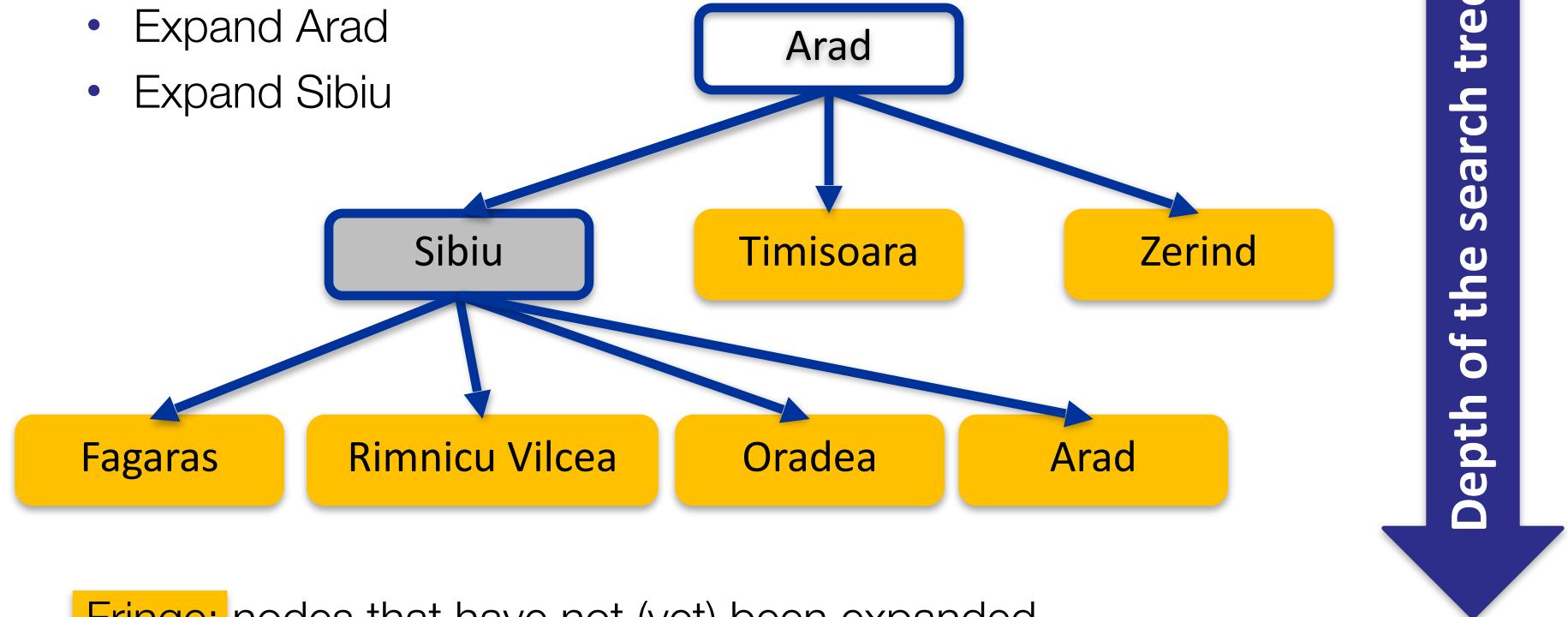
Tree search example

- Initial State: In(Arad)
- Expand Arad



Tree search example

- Initial State: In(Arad)
- Expand Arad
- Expand Sibiu



Fringe: nodes that have not (yet) been expanded
→ leave nodes of the current tree

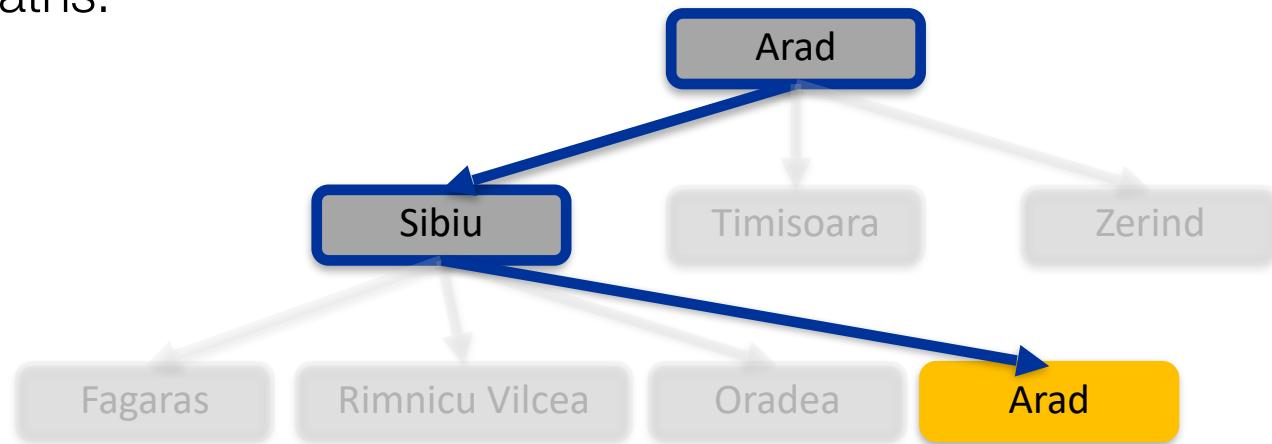
States vs. nodes

State

- (representation of) a problem configuration

Node

- data structure constituting a part of a search tree
- can have parent and successor nodes
- two different nodes can point to the same state which has been reached by different paths:



Search strategies

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

- Breadth-first search (BFS)
- Depth-first search (DFS)
- Iterative deepening search
- Bidirectional search

Informed search strategies have knowledge that allows to guide the search to promising regions → **heuristic search**

- Greedy search
- A* best-first search

Problem-solving agents

Tree-search algorithms

Breadth-first search

Depth-first search

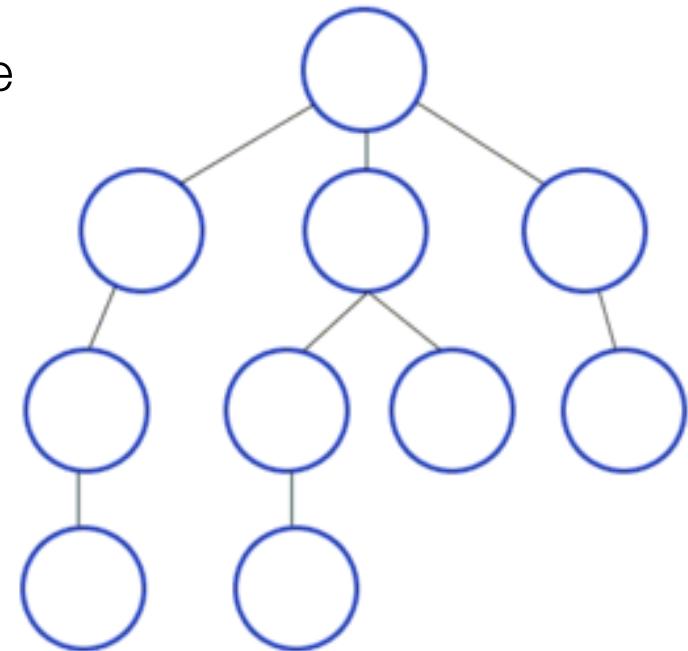
Iterative deepening

Performance of a search algorithm

Breadth-first strategy

Expand all neighbors of a node before any of its successors is expanded

- breadth before depth
- expand the shallowest unexpanded node

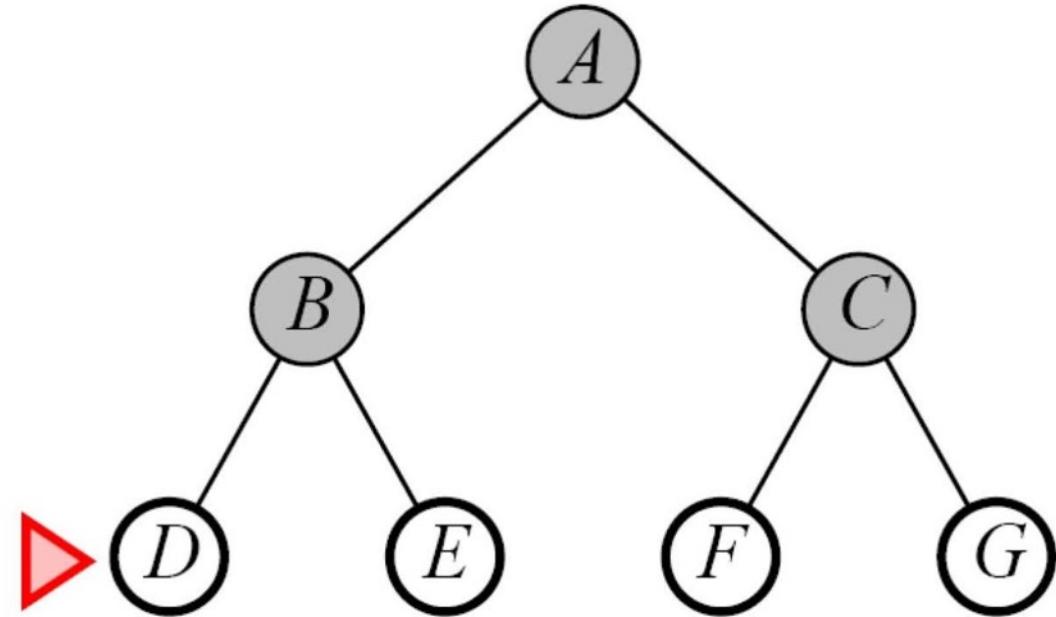


Check animation on: <https://de.wikipedia.org/wiki/Breitensuche>

Breadth-first implementation

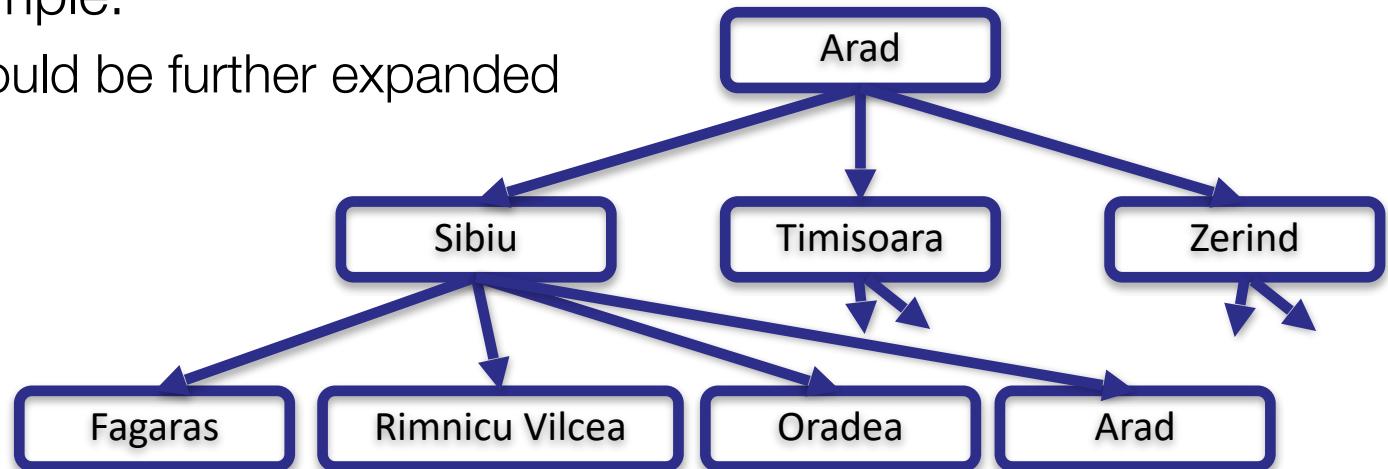
Breadth-first search is implemented as a FIFO queue:

- FIFO = First in, first out
- Expand start node A → Queue: B, C
- Expand node B → Queue: C, D, E
- Expand node C → Queue: D, E, F, G
- Expand node D



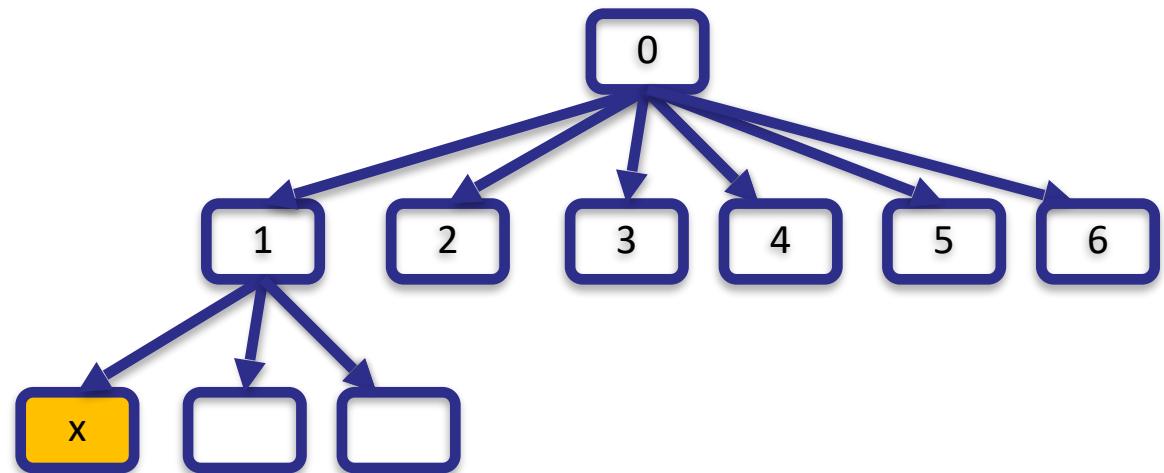
Tree Shapes

- d = depth (starting at 0)
- b = maximum branching factor = maximum number of successor nodes
- b_i = maximum branching factor at depth i
- Romania-Example:
 - partial tree, could be further expanded
 - $d = 2$
 - $b = 4$
 - $b_0 = 3$



Tree examples: Breadth-first

- $d = 4$
- $b = 1$
- Breadth-first search steps to $x = 4$
- Path cost = 4



◊ $d = 2$

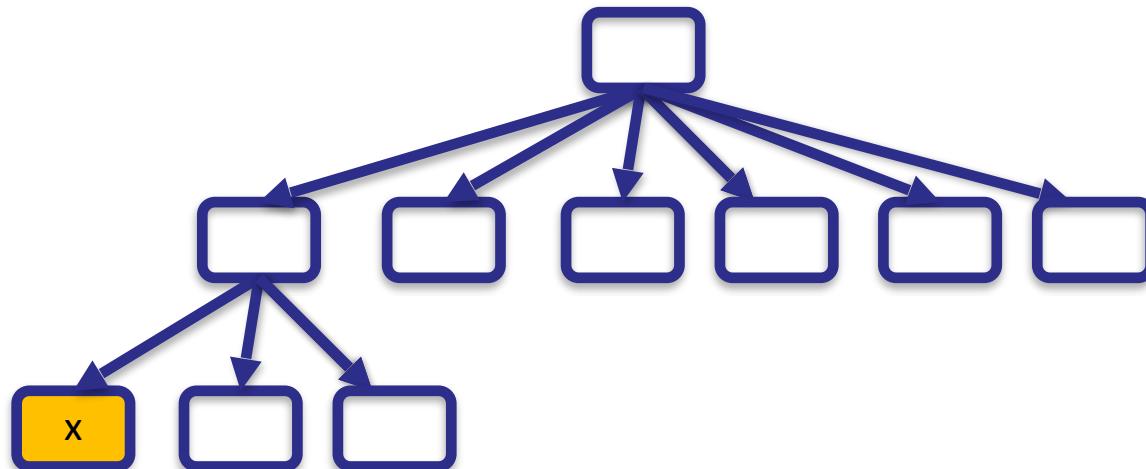
◊ $b = 6$

◊ Breadth-first search steps to $x = 7$

◊ Path cost = 2

Shortcomings of BFS

- Needs to keep track of all discovered nodes for later expansion
- → large memory consumption



Problem-solving agents

Tree-search algorithms

Breadth-first search

Depth-first search

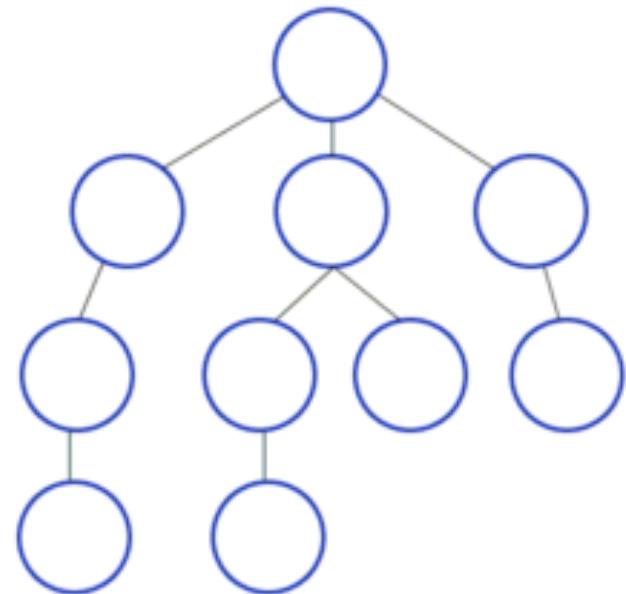
Iterative deepening

Performance of a search algorithm

Depth-first Search (DFS)

Expand all successors of a node before any of its neighbors is expanded

- depth before breadth
- expand the deepest unexpanded node



Check animation on: <https://de.wikipedia.org/wiki/Tiefensuche>

Depth-first implementation

Depth-first search is implemented as a LIFO stack

LIFO: Last in ↓, first out ↑



Stack

FIFO: → first in, → first out



Queue

Comparing Implementations

```
def bfs(graph, root):
    seen = set([root])
    queue = collections.deque([root])
    while queue:
        print("Queue: ", queue)
        vertex = queue.popleft()
        print("Expanding: ", vertex)
        for node in graph[vertex]:
            if node not in seen:
                seen.add(node)
                queue.append(node)
```

```
def dfs(graph, root):
    seen = set([root])
    stack = [root]
    while stack:
        print("Stack: ", stack)
        vertex = stack.pop()
        print("Expanding: ", vertex)
        for node in graph[vertex]:
            if node not in seen:
                seen.add(node)
                stack.append(node)
```

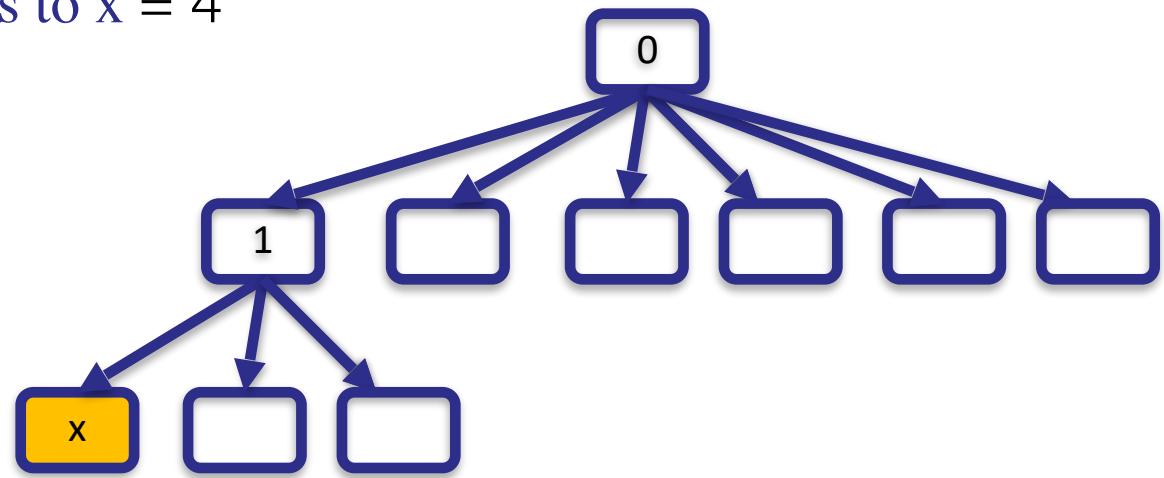
DFS implementation

```
graph = {
    "A": ["B", "C"],
    "B": ["D", "E"],
    "C": ["F", "G"],
    "D": ["H", "I"],
    "E": ["J", "K"],
    "F": [], "G": [], "H": [], "I": [], "J": [], "K": []
}
```

DFS
Stack: ['A']
Expanding: A
Stack: ['B', 'C']
Expanding: C
Stack: ['B', 'F', 'G']
Expanding: G
Stack: ['B', 'F']
Expanding: F
Stack: ['B']
Expanding: B
Stack: ['D', 'E']
Expanding: E
Stack: ['D', 'J', 'K']
Expanding: K
Stack: ['D', 'J']
Expanding: J
Stack: ['D']
Expanding: D
Stack: ['H', 'I']
Expanding: I
Stack: ['H']
Expanding: H

Tree Shapes

- $d = 4, b = 1$
- Breadth-first search steps to $x = 4$
- Depth-first search steps to $x = 4$
- Path cost = 4



◊ $d = 2, b = 6$

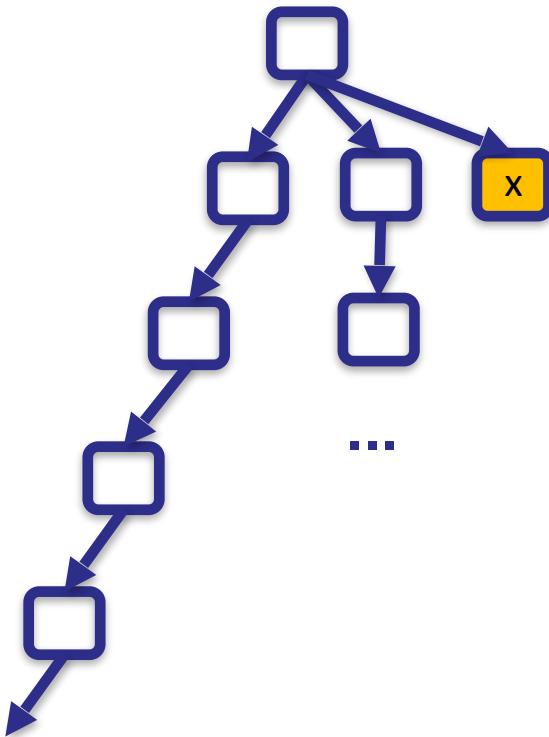
◊ Breadth-first search steps to $x = 7$

◊ Depth-first search steps to $x = 2$

◊ Path cost = 2

Shortcomings of DFS

- Might explore very deep path, before finding solution higher in tree.
- Could even get stuck in sub-tree with infinite depth.



Problem-solving agents

Tree-search algorithms

Breadth-first search

Depth-first search

Iterative deepening

Performance of a search algorithm

Limit the depth

Overcome shortcomings of DFS by setting a maximum depth L

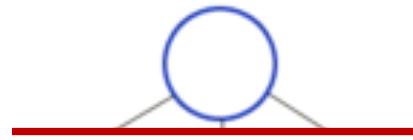
- How to set L ?
- If L is too small, the solution is not found.
- If L is too big, the search takes too long

Iterative deepening

- Try all possible $L = 0, 1, 2, 3, \dots$
 - The costs are dominated by the last iteration
- overhead of iterative process can be neglected

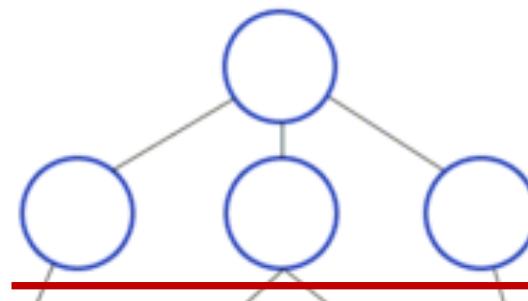
Iterative Deepening

- Limit = 0



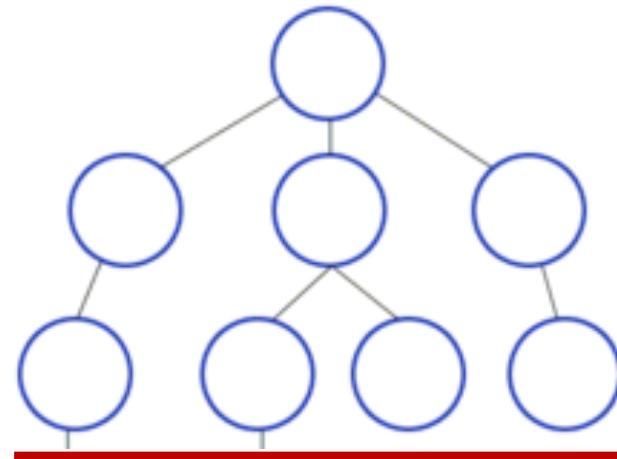
Iterative Deepening

- Limit = 1



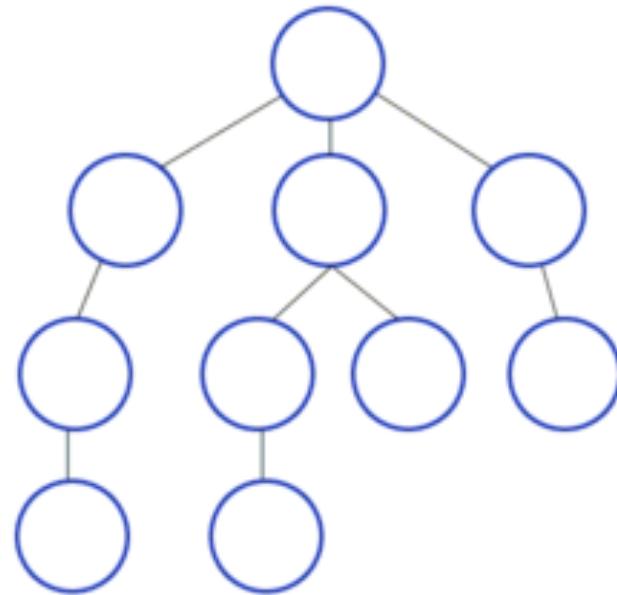
Iterative Deepening

- Limit = 2



Iterative Deepening

- Limit = 3

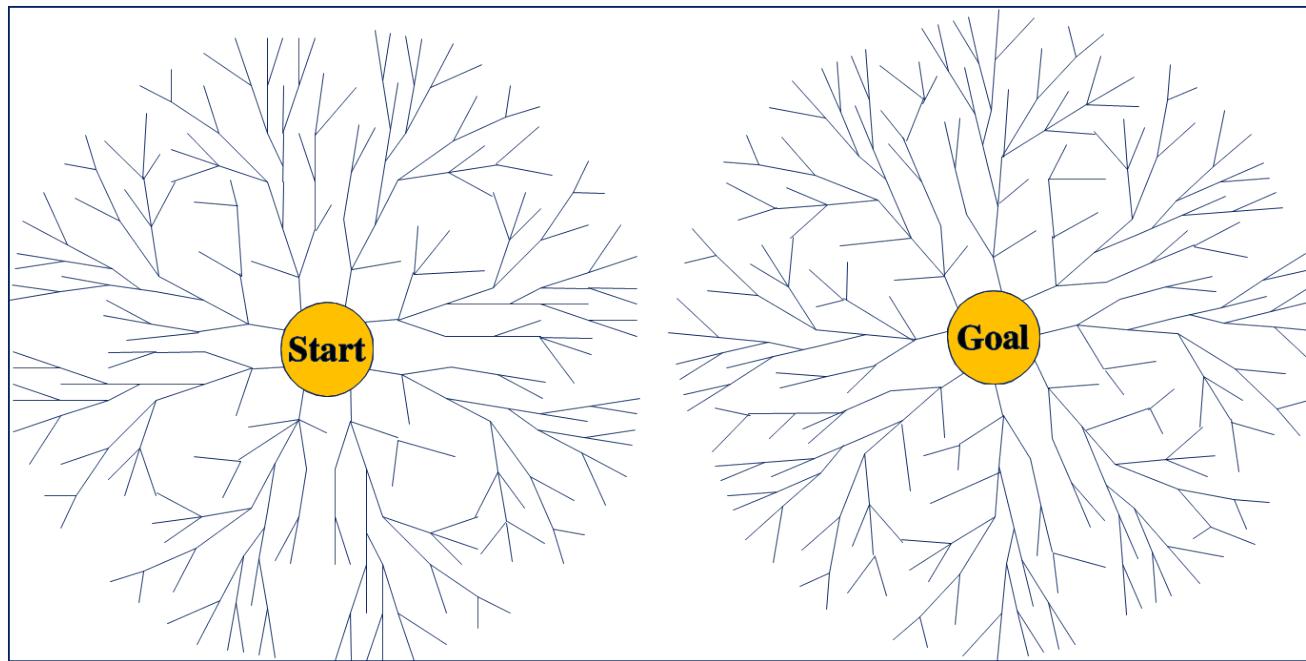


- In the last iteration, we have standard depth-first search again:

Bidirectional search

Perform two searches simultaneously

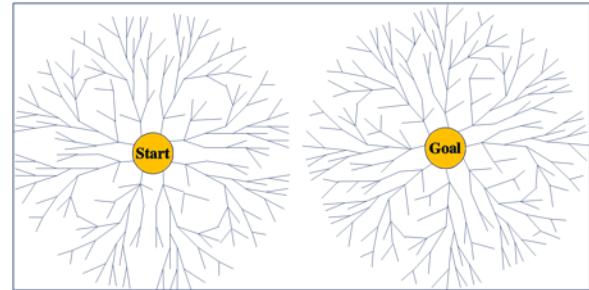
- ◊ forward: starting with initial state
- ◊ backward: starting with goal state



Bidirectional search

Perform two searches simultaneously

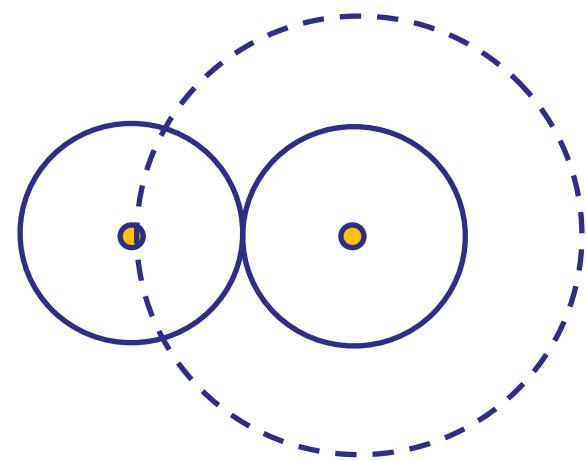
- ◊ forward starting with initial state
- ◊ backward starting with goal state



Check whether generated node is in fringe of the other search

Properties

- ◊ reduction in complexity
- ◊ only possible if actions can be reversed
- ◊ search paths may not meet for DFS \Rightarrow use BFS



Problem-solving agents

Tree-search algorithms

Breadth-first search

Depth-first search

Iterative deepening

Performance of a search algorithm

Search strategies

Different search strategies have different strength and weaknesses.

Search strategies are compared along the following dimensions:

- **Completeness:** Does it always find a solution if one exists?
- **Optimality:** Does it always find a least-cost solution?

Completeness & Optimality

Completeness:

Do we always find a solution if one exists?

- **Breadth-first:** yes
- **Depth-first:** no, fails in infinite-depth search spaces and spaces with loops
- **Iterative deepening:** yes

Optimality:

Do we always find a least-cost solution?

- **Breadth-first:** yes, for uniform costs
- **Depth-first:** no, solutions with longer paths may be found earlier than solutions with shorter path
- **Iterative deepening:** yes

Search strategies

Different search strategies have different strength and weaknesses.

Search strategies are compared along the following dimensions

- **completeness:** Does it always find a solution if one exists?
- **optimality:** Does it always find a least-cost solution?
- **time complexity:** How long does the search take?
- **space complexity:** How much memory does the search need?

→ Time and space complexity are presented by the O-notation.

O-Notation

Linear time complexity

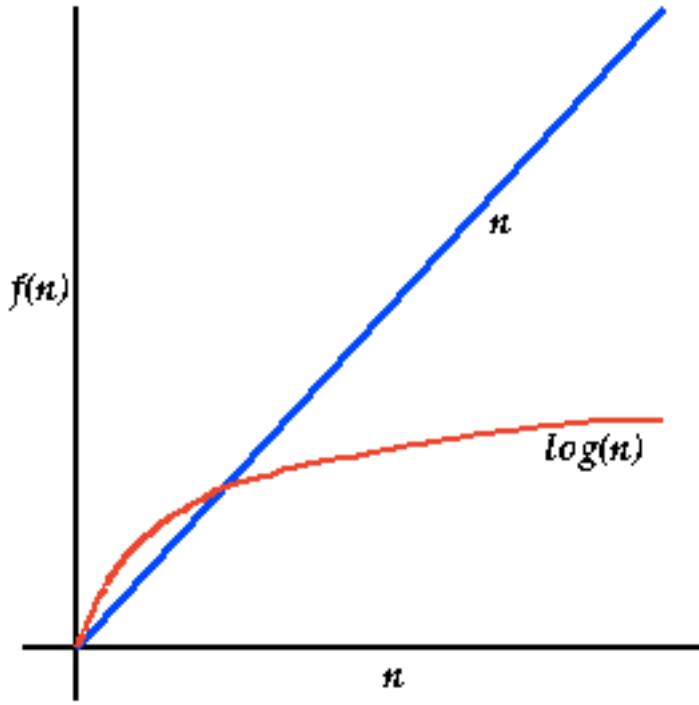
- Number of calculation steps that need to be performed
- Example: addition
 - 2 numbers, 1 step: $a + b$
 - 3 numbers, 2 steps: $(a + b) + c$
 - 4 numbers, 3 steps: $((a + b) + c) + d$
 - n numbers → $n - 1$ steps
 - $2n$ numbers → $2n - 1$ steps
- Number of calculation steps grows linearly with the number of input elements
- The constant -1 becomes less and less important for large numbers
- Order of magnitude → $O(n)$

Quadratic time complexity

- Example: check for duplicates in a list
 - $n = 2$, 1 step
input: [a, b], compare: a/b
 - $n = 3$, 3 steps
input: [a, b, c], compare: a/b, a/c, b/c,
 - $n = 4$, 6 steps
input: [a, b, c, d], compare: a/b, a/c, a/d, b/c, b/d, c/d
 - $n = 5$, 10 steps
input: [a, b, c, d, e], compare: a/b, a/c, a/d, a/e, b/c, b/d, b/e, c/d, c/e, d/e
 - n inputs $\rightarrow \frac{n * (n - 1)}{2}$ steps
 - $2n$ inputs $\rightarrow \frac{2n * (2n - 1)}{2}$ steps
- Number of calculation steps grows quadratically with number of input elements
- Order of magnitude $\rightarrow O(n^2)$

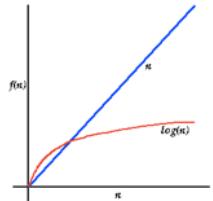
Logarithmic time complexity

- Recall logarithms:
 - $2^x = n$
 - $\log_2 n = x$
 - n = input, x = time steps



Logarithmic time complexity

- Searching a number in a phone book of n pages
- Start in the middle: $\frac{n}{2}$, check name
 - names on page are later in the alphabet → go to middle of left part: $\frac{n}{4}$
 -
 - names on page are earlier in the alphabet → go to middle of right part: $\frac{3}{4}n$
- If we double the number of pages to $2n$ pages, we only need to add one step:
 - Start in the middle: $\frac{2n}{2} = n$, check name
 - names on page are later in alphabet → go to middle of left part: $\frac{2n}{4} = \frac{n}{2}$
 -
- Number of calculation steps grows logarithmically with number of input elements
- Order of magnitude → $O(\log(n))$



O-notation

The complexity is calculated for the worst-case scenario.

- If I search for an element in a list of size n , I might find it at position 1 (best case) or position n (worst case) or somewhere in between
- If I search for an element in a list of size $2n$, I might find it at position 1 (best case) or position $2n$ (worst case) or somewhere in between
- best-case scenario remains constant → $O(1)$
- worst-case scenario grows linearly → $O(n)$



Determining the O-notation

Constants are ignored.

- $100000n^2 = O(n^2)$

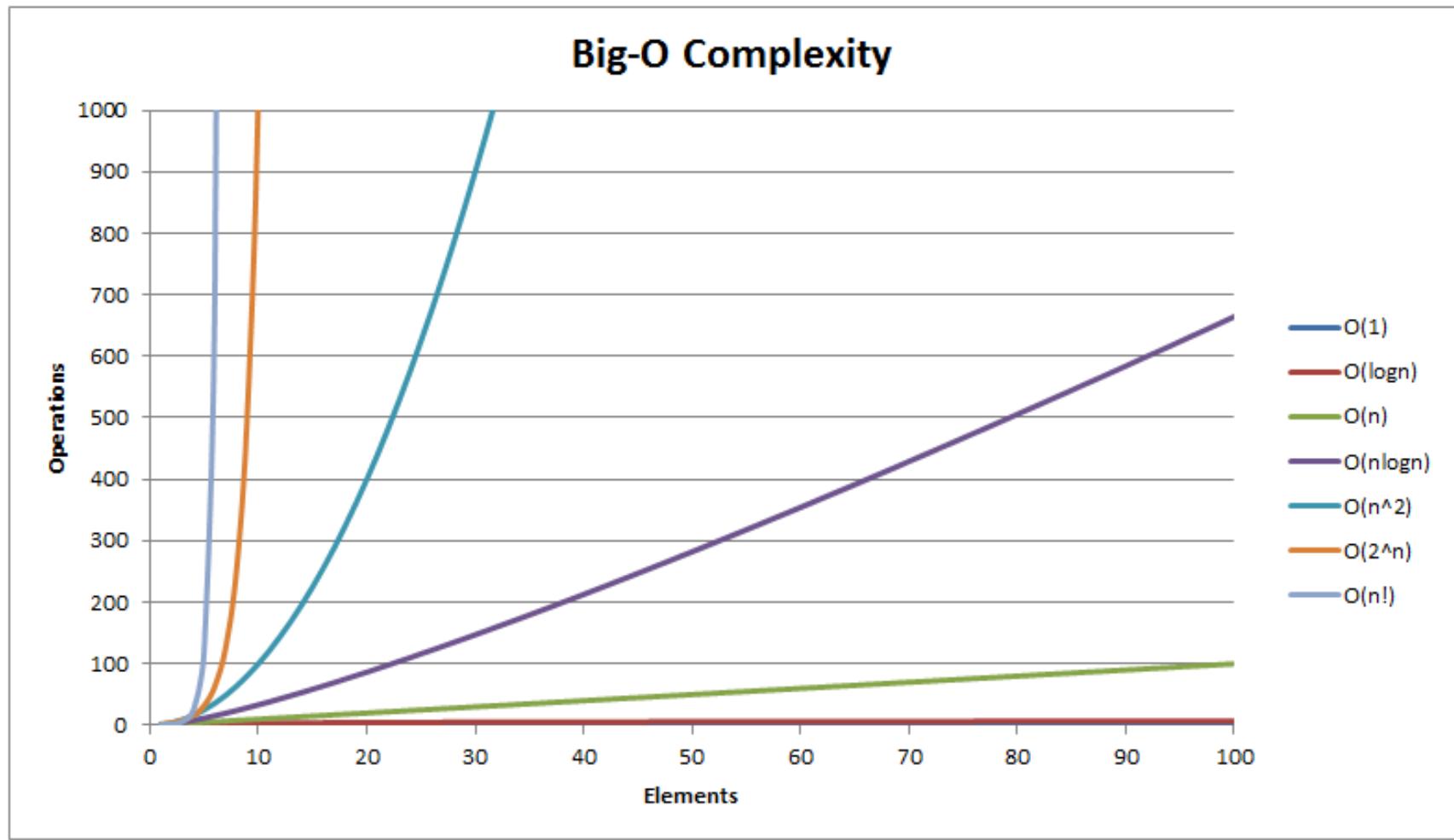
Order of magnitude is dominated by the highest-order term.

- $7n^3 + 2n^5 + 24n = O(n^5)$

Complexity should be expressed by minimal O.

- $2n$ is contained in $O(n^2)$ and $O(n^3)$,
but it is more accurate to say $2n = O(n)$

Function growth rates



<https://medium.com/@devontem/simple-guide-to-big-o-notation-time-complexity-space-complexity-9e906c60f7f9>

Performance Review Continued

Search strategies

Different search strategies have different strength and weaknesses.

Search strategies are compared along the following dimensions

- completeness: Does it always find a solution if one exists?
- optimality: Does it always find a least-cost solution?
- **time complexity:** How long does the search take?
- **space complexity:** How much memory does the search need?

Parameters:

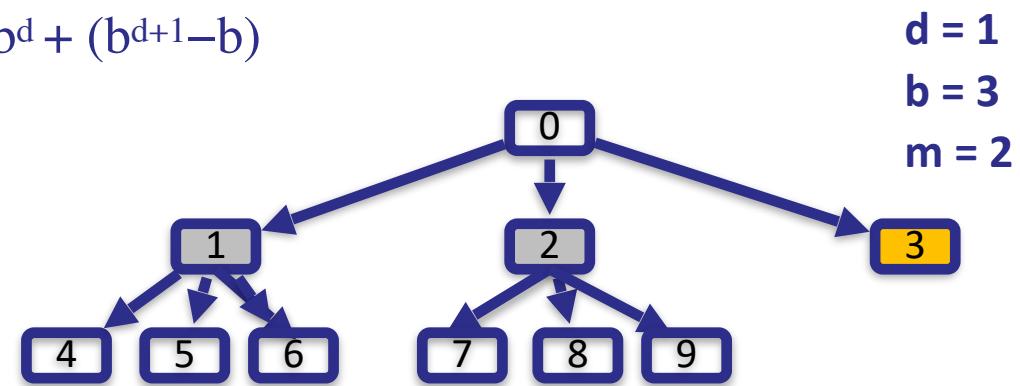
- b: maximum branching factor of the search tree
- d: depth of the least-cost solution
- m: maximum depth of the state space (may be ∞)

Time complexity

Time complexity: number of nodes that need to be generated

Breadth-first:

- depth 0 has 1 node
- depth 1 has max. $1 * b$ nodes
- depth 2 has max. $1 * b * b$ nodes
- goal is in level d
- worst case: goal is last node in level d
- search steps: $b + b^2 + b^3 \dots + b^d + (b^{d+1}-b)$
- $\rightarrow O(b^{d+1})$



Time complexity

Time complexity: number of nodes that need to be generated

Breadth-first:

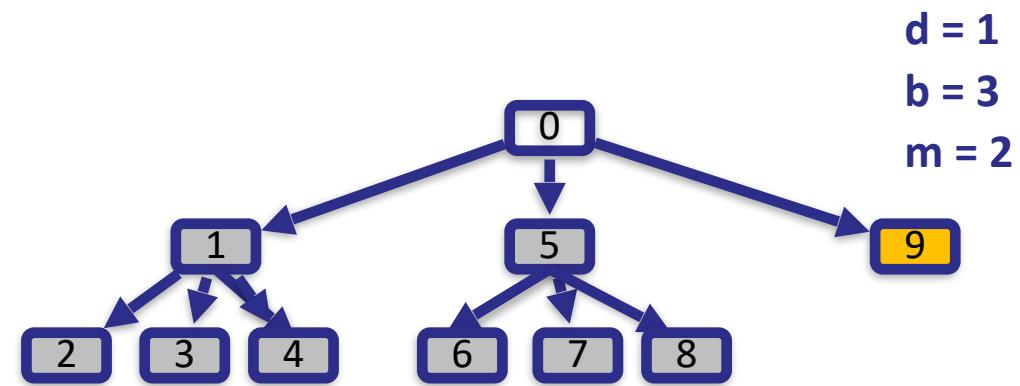
$$\rightarrow O(b^{d+1})$$

Depth-first:

- follows each path until maximum depth m

$$\rightarrow O(b^m)$$

- terrible if m much larger than d



Time complexity

Time complexity: number of nodes that need to be generated

Breadth-first:

$$\rightarrow O(b^{d+1})$$

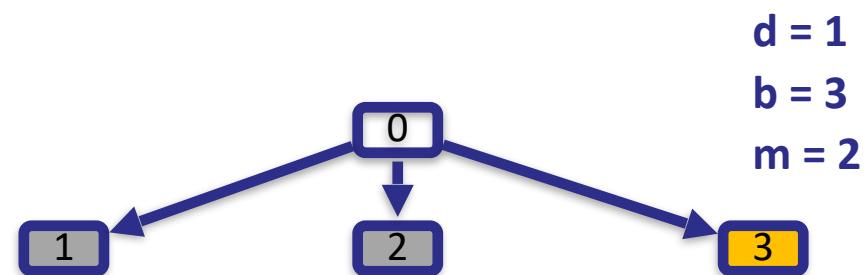
Depth-first:

$$\rightarrow O(b^m)$$

Iterative deepening:

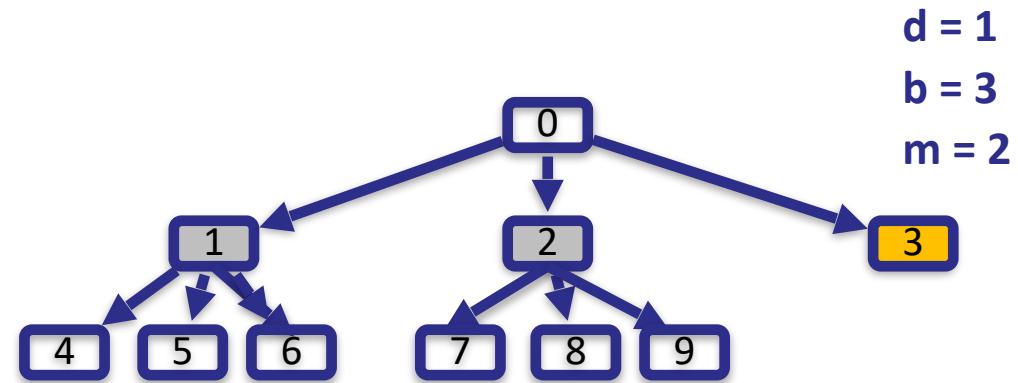
- m never goes beyond d

$$\rightarrow O(b^d)$$



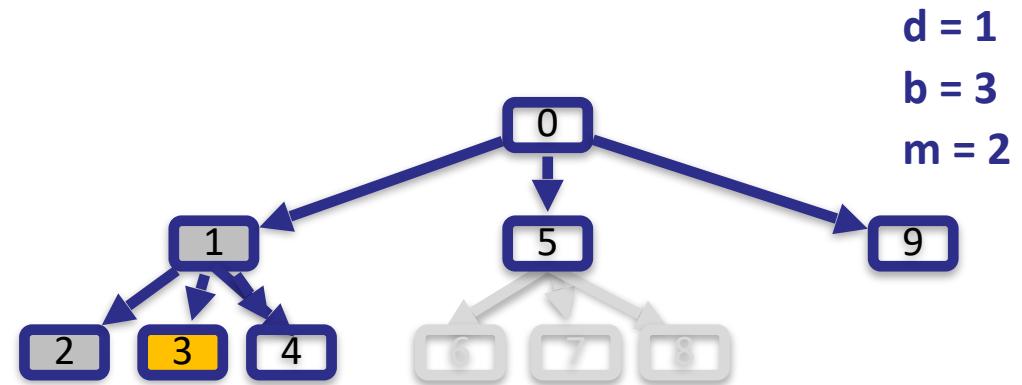
Space Complexity

- **Space complexity:** maximum number of nodes in memory
 - **Breadth-first:**
 - Every node must remain in memory because it is either a fringe node or an ancestor of a fringe node
 - In the end, the goal will be in the fringe, and its ancestors will be needed for the solution path
- $O(b^{d+1})$



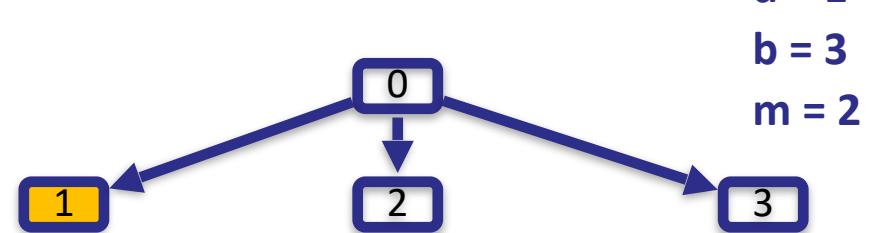
Space Complexity

- Space complexity: maximum number of nodes in memory
- Breadth-first:
→ $O(b^{d+1})$
- Depth-first:
 - only nodes in current path and their unexpanded siblings need to be stored
→ $O(bm)$



Space Complexity

- Space complexity: maximum number of nodes in memory
- Breadth-first:
→ $O(b^{d+1})$
- Depth-first:
→ $O(bm)$
- Iterative deepening:
 - only nodes in current path and their unexpanded siblings need to be stored
→ $O(bd)$



Performance review summary

- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms



	Breadth-first	Depth-first	Iterative deepening
Complete?	Yes	No	Yes
Optimal?	Yes	No	Yes
Time complexity	b^{d+1}	b^m	b^d
Space complexity	b^{d+1}	bm	bd

Complexity in Practice

Issue: Size of constant factor

Which algorithm would you choose?

Algorithm 1

- $2,000 n = O(n)$

Algorithm 2

- $2 n^2 = O(n^2)$

Depends on problem size

- for problem size with $n < 1,000$ Algorithm 2 is faster

Issue: Memory usage

Which algorithm would you choose?

Algorithm 1

- time-complexity $O(n \log n)$
- memory usage $O(2^n)$

Algorithm 2

- time-complexity $O(n^2)$
- memory usage $O(n)$

Depends on how much memory you have

- but Algorithm 1 will only fit in memory for very small n

Issue: Programming time

Which algorithm would you choose?

Algorithm 1

- time-complexity $O(n \log n)$
- programming time 1 month

Algorithm 2

- time-complexity $O(n^2)$
- programming time 1 hour

Depends on how much time you have ☺

- if you need to run it often or for large problems, might be worth the time
- but faster algorithms are usually also more complex and more error-prone

Issue: Quality of solution

Which algorithm would you choose?

Algorithm 1

- time-complexity $O(2^n)$
- finds the optimal answer

Algorithm 2

- time-complexity $O(n^2)$
- finds a good answer

Depends on whether you need the optimal answer

- often an **approximation** is good enough
 - can usually be found much, much faster than optimal solution

Summary

Tree search algorithms operate on an abstraction of the real problem.

Search strategies

- Breadth-First (BFS)
- Depth-First (DFS)
- Iterative Deepening (IDS)

Evaluation of search strategies

- completeness
 - optimality
 - time complexity
 - space complexity
-
- O-Notation



Readings

Mandatory

- Russell & Norvig, Section 3, *Solving Problems by Searching*, 3.1-3.4
- Reading Question:

What is uniform-cost search?

How does it differ from the other search strategies?

Optional

- Russell & Norvig, Rest of Section 3
- Joachim Ziegler: Tutorial on O-notation
<http://www.leda-tutorial.org/de/offiziell/ch02s02s03.html>

Next lecture: Find your way like Google

