Artificial Intelligence 1

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Problem Solving: Uninformed Search

Introduction

Simple reflex agent „only“ transfers the current percept to one action.

Goal-based agent can plan action sequences that achieve agent’s goals.

- We will now explore one kind of goal-based agent – a **problem-solving agent**.
- What is a **problem** and what is its **solution**?
- How to **find a solution**?
- **Search** algorithms
  BFS, DFS, DLS, IDS, BiS

Note:

First, we will explore “uninformed” search algorithms, that is, algorithms that do not exploit any problem specific information (and use an atomic representation).

Intelligent agents are supposed to maximize their performance measure. This can be simplified if the agent can adopt a **goal** and aim at satisfying it.

A model problem and a possible way to solve it

- an agent is in the city of Arad (Romania)
- the performance measure contains many factors (improve its suntan, take in the sights, enjoy the nightlife, …), which make decisions hard
- now, suppose the agent has a non-refundable ticket to fly out of Bucharest the following day
- **goal formulation** – getting to Bucharest – greatly simplifies the agent's decision problem
- a goal is a set of world states, where the goal is satisfied, but we also need to assume other world states and actions for moving between – **problem formulation**
  - states could correspond to being in major towns (the exact location is too fine)
  - actions could correspond to driving from one major town to another (acceleration and breaking are too fine)
- How to find a path to Bucharest, if three roads lead out from Arad, one toward Sibiu, one to Timisoara, and one to Zerind?
  - Suppose the agent has a map of Romania and can explore possible journeys (**search**), select the best one, and then finally **execute** the actions.

Problem solving

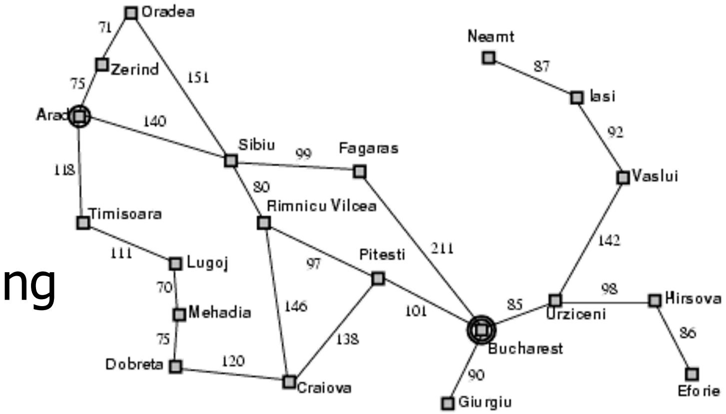
Problem solving consists of four steps:

- **goal formulation**
  - What are the desired states of the world?
- **problem formulation**
  - What are the states and actions assumed to reach the goal?
- **problem solving**
  - How to find the best sequence of states reaching the goal?
- **solution execution**
  - If we know the actions, how to execute them?

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH( problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Problem formulation



Well-defined problems consist of:

- **the initial state**
  - ***in(Arad)***
- a description of possible **actions** can be done via a **transition model** describing for each state possible actions and states to which the action leads
  - ***SUCCESSOR-FN(in(Arad)) = 〈 go(Sibiu), in(Sibiu) 〉***
  - implicitly defines the **state space** (the set of all states reachable from the initial state by any sequence of actions)
  - a **path** is a sequence of states connected by actions.
- **the goal test**
  a function determining whether a given state is a goal state or not
  - ***{ in(Bucharest) }***
- **a path cost**
  a function that assigns a numeric cost to each path (reflects the performance measure)
  - We assume that the cost of a path can be described as a sum of the costs of individual actions along the path.

A **solution to a problem** is an action sequence that leads from the initial state to a goal state.

An optimal solution is a solution that has the lowest path cost among all solutions.

Abstraction

In the problem formulation, we used

- **abstraction of world states**
  - we ignored weather, traffic conditions, …
- **abstraction of actions**
  - we ignored turning on the radio, looking out of the window, …

Abstraction is the process of removing detail from representation.

What is the **appropriate level of abstraction**?

- the abstraction is **valid**
  - we can expand any abstract solution into a solution in the more detailed world
  - we can find a way from any place in Arad to any place in Sibiu
- the abstraction is **useful**
  - carrying out each of the actions in the solution is easier than the original problem
  - the path from Arad to Sibiu can be executed by average driving agent

Intelligent agents would be completely swamped by the real world without using abstractions!

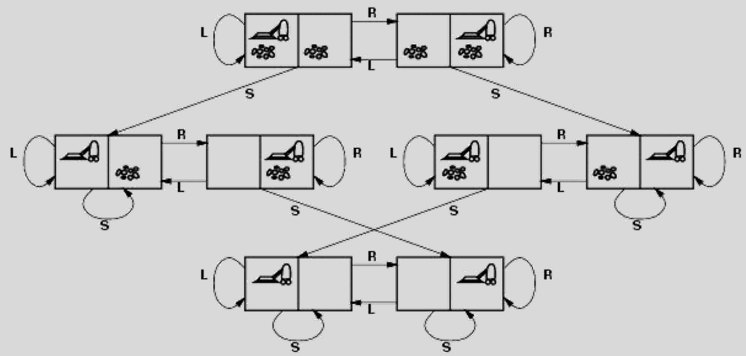
Toy problems are used to illustrate and to compare the solving techniques.

The vacuum world

states (location × dirt)

start, goal

actions (L, R, S)

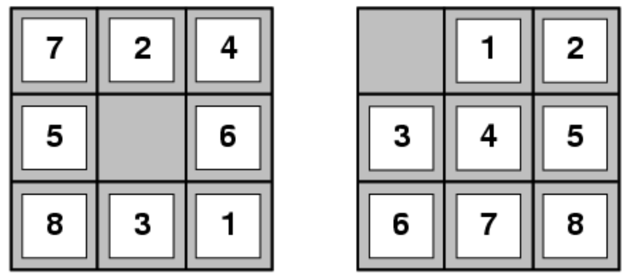


The 8-puzzle

states (blocks' positions)

start, goal

actions (L, R, U, D)

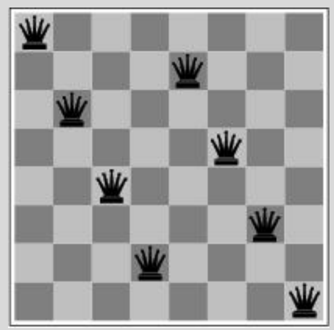


Start State Goal State

The 8-queens

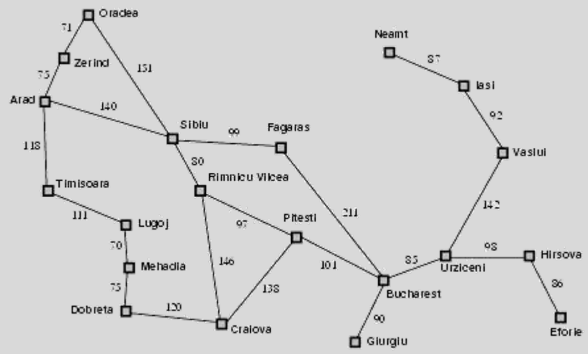
place queens on a chessboard without attacks

- incremental formulation (adding queens)
- complete-state formulation (moving queens)



That is what matters!

Route-finding problems

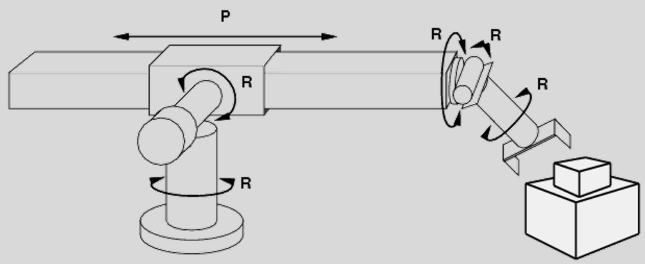


states (places)
start (here and now), goal (there, on-time)
successor
cost function (time, cost,…)

Touring problems

goal (to visit each place at least once)
states (current and visited places)
TSP (Travelling Salesman Problem), NP-hard

Product assembly problems

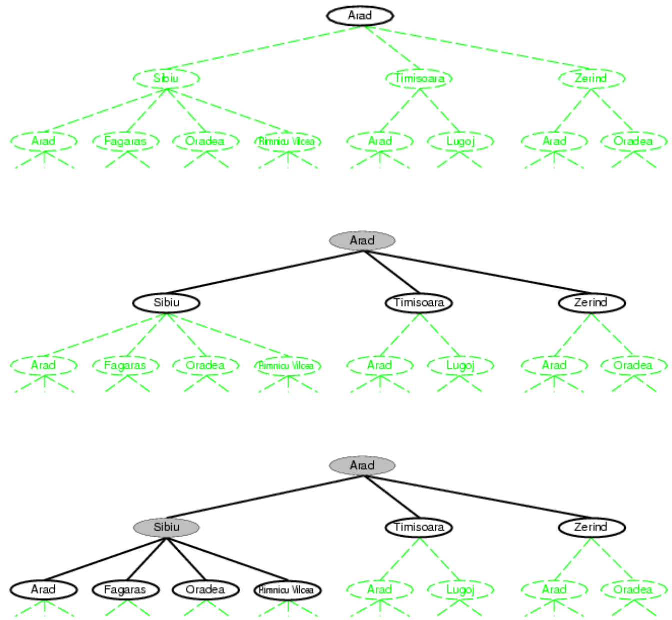


state (arm location × components)
goal (assembled product)
successor (movement of „hinges“)
cost (total assembly time)
protein design from a sequence of amino acids

Having formulated the problem, how do we solve it?

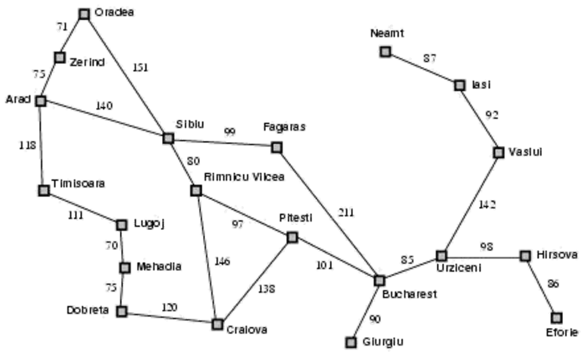
State space search

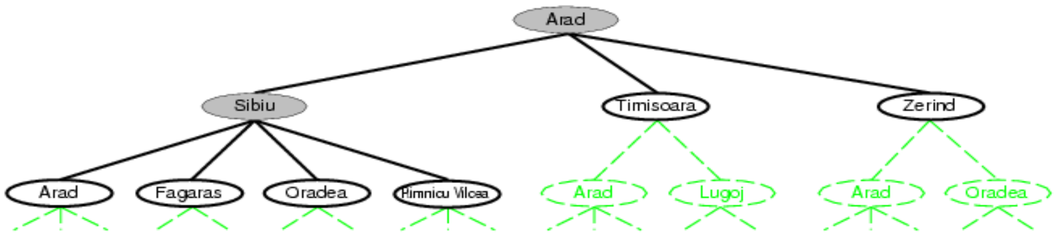
- start in the **initial state** (root node)
- **check** whether **the initial state** is **a goal state**
- if the state is not a goal state then **expand the state**, it generates a set of new states
- **select a next** state using a **search strategy**




```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

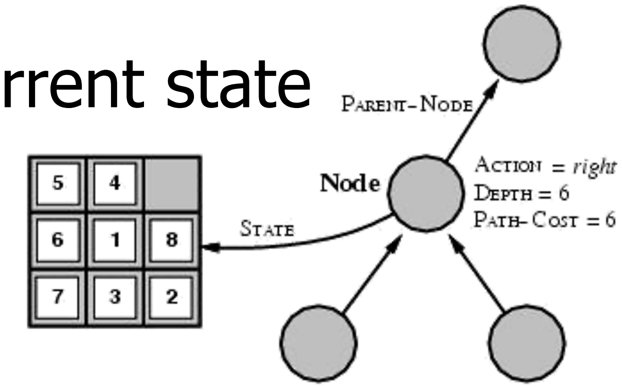
State space is different from search tree – world state is different from search node.



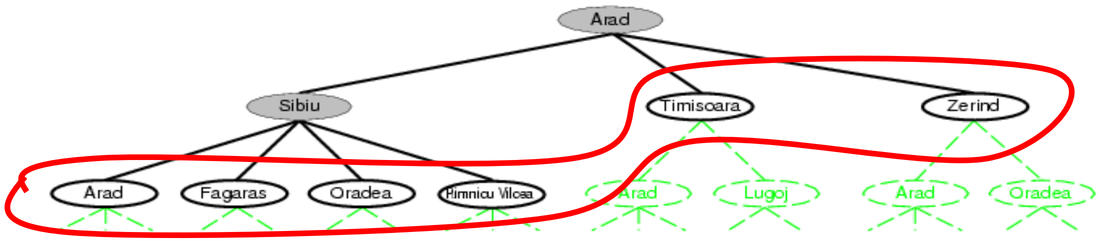


Node of the search tree consists of:

- a current **state**
- a link to its **parent**
- **action** leading from parent to the current state
- **cost** of path from the root g(n)
- **depth** (the number of steps from the root)



Fringe (**frontier**) is a set of nodes not yet expanded.



Nodes from the frontier are called **leafs**.

The algorithms can be described using the following operations over the fringe represented as a **queue**:

- MAKE-QUEUE(element,…)
- EMPTY?(queue)
- FIRST(queue)
- REMOVE-FIRST(queue)
- INSERT(element, queue)
- INSERT-ALL(elements, queue)

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(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
        PARENT-NODE[s] ← node;   ACTION[s] ← action;   STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

We can evaluate an **algorithm's performance** in four ways:

- **completeness**
  - Does the algorithm guarantee to find a solution when there is one?
- **optimality**
  - Does the strategy find the optimal solution?
- **time complexity**
  - How long does it take to find a solution?
- **space complexity**
  - How much memory is needed to perform search?
- Time and space complexity are always considered with respect to some measure of problem difficulty.
  - **branching factor b** (maximum number of successors of any node) – useful for implicit representation of world states (the initial state and successors)
  - **depth d** (the path length from the root to a shallowest goal node)
  - **path length m** (the maximum length of any path in the state space)

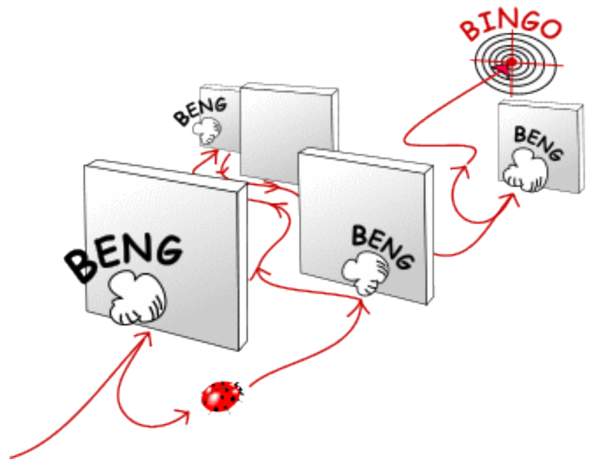
- **search cost** (how much time and space we need to find a solution)
- **total cost** (combines the search cost and the path cost of the solution found)

Today we will cover **uninformed (blind) search**

– no additional information about states beyond that provided in the problem definition
– it can only generate successors and distinguish a goal state from a non-goal state

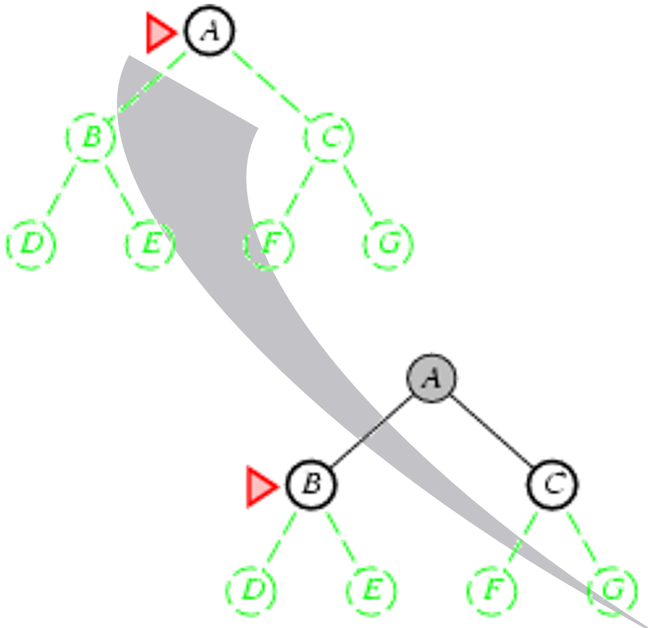
Next lecture will cover **informed** (**heuristic**) **search**

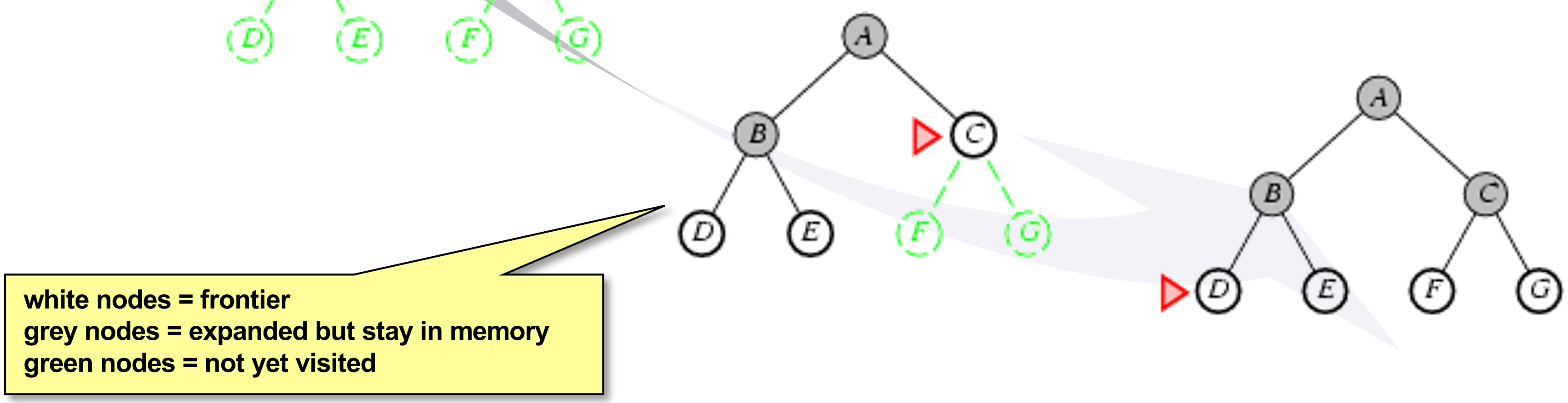
– information to distinguish "**more promising**" non-goal states from other states
– for example by estimating distance to some goal state



All nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

- the **shallowest unexpanded node** is chosen for expansion
- achieved by using a **FIFO** (first-in-first-out) for the frontier





It is a **complete method** (provided that the branching factor is finite).

The shallowest goal state is not necessarily the optimal one!

- BFS is optimal if the path cost is a non-decreasing function of the depth of the node

Time complexity (the number of visited nodes for the goal at depth d)

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

Space complexity

- store every expanded node in the explored set
- $\mathbf{O(b^{d+1})}$

The memory requirements are a bigger problem than is the execution time.

depth	nodes	time	memory
2	1100	0.11 sec.	1 megabyte
4	111100	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}	3523 years	1 exabyte

b = 10
10.000 nodes/sec.
1000 bytes/node

Modifying BFS for **finding an optimal solution**.
Expand the node n with the **lowest path cost** g(n).
Beware of zero-cost steps – they can cause cycling!

Completeness (and optimality) can be guaranteed if the cost of each step is lower-bounded by some ε.

Time (and space) complexity

- depends on the path cost rather than on the depth
- $\mathbf{O(b^{1+\lfloor C^*/\varepsilon \rfloor})}$, where C* is the cost of the optimal solution
- **Could be much worse than for BFS**, because the algorithm prefers long paths with "cheap" steps over shorter paths with more expensive steps.

Also known as Dijkstra's algorithm.

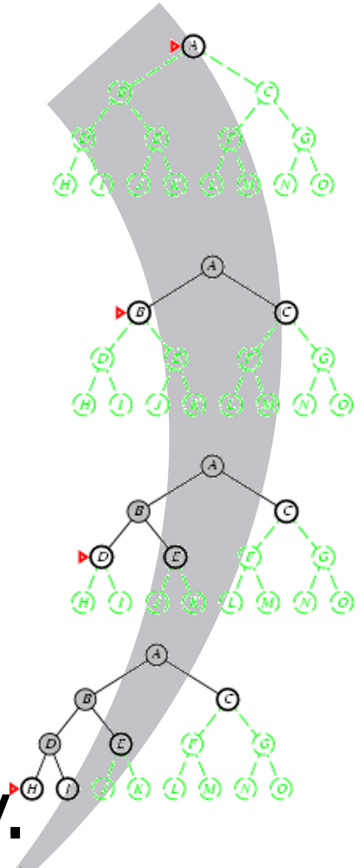
Depth-first search

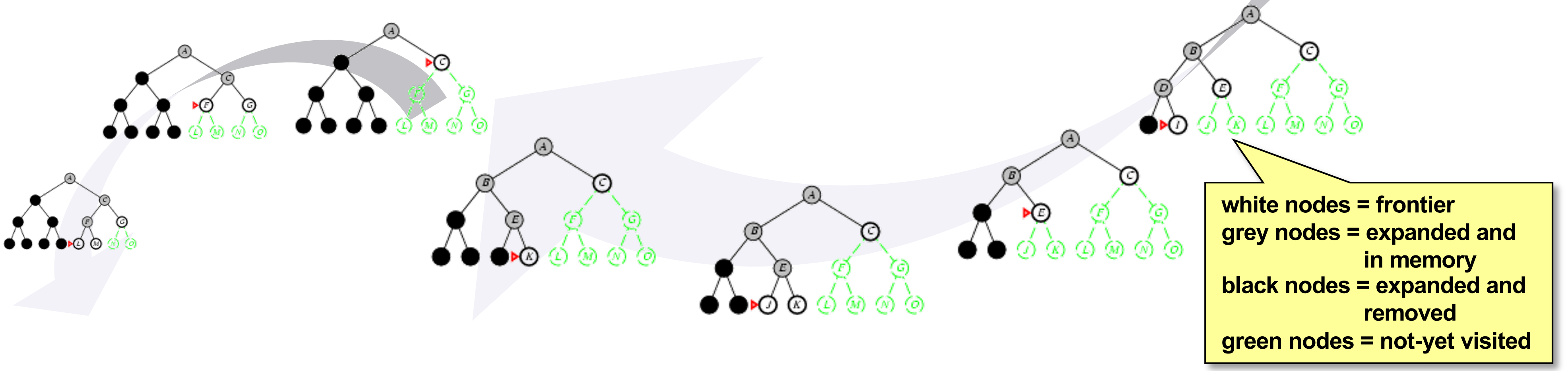
Always **expand the deepest node**

until reaching the level, where the nodes have no successors and then the search "backs up" to the node at the shallower level

achieved by using a **LIFO** (last-in-first-out) stack for the frontier

Frequently implemented in a recursive way.





If the algorithm selects a wrong path, it may not find a goal during tree search (it is **not complete**) and of course it may **not find optimum**.

Time complexity

- **O(b^m)**, where m is the maximum depth
- it may happen that d << m, where d is the depth of the optimum

Space complexity

- needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes
- expanded node can be removed as soon as all its descendants have been fully explored
- **O(bm)**, where m is the maximum visited depth
- Space complexity can be decreased via **backtracking**!
  - generate one successor (instead of all) $\rightarrow$ **O(m)** states
  - modify the current state rather than copying it (we must be able to undo each modification when going back to generate the next successor $\rightarrow$ **O(1)** states and **O(m)** actions

Depth-limited search

The failure of DFS in infinite state spaces can be alleviated by supplying DFS with a **predetermined depth limit l**.

- nodes at depth l are treated as if they have no successors
- terminates with a solution, with a failure, or with cut-off
- **time** complexity **$O(b^l)$**, **space** complexity **O(bl)**
- if l < d, then the algorithm is not complete (d is a depth of solution)
- if d << l, then the algorithm explores many useless nodes

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
   RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
   cutoff-occurred? ← false
   if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
   else if DEPTH[node] = limit then return cutoff
   else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
   if cutoff-occurred? then return cutoff else return failure
```

How to decide the depth limit?

- using knowledge of the problem
- for example, for path finding for 20 cities, we can use the depth limit 19
- if we study the map carefully, we can discover that any city can be reached from any other city in at most 9 steps (the **diameter** of the state space)

How to make the depth limited search complete?

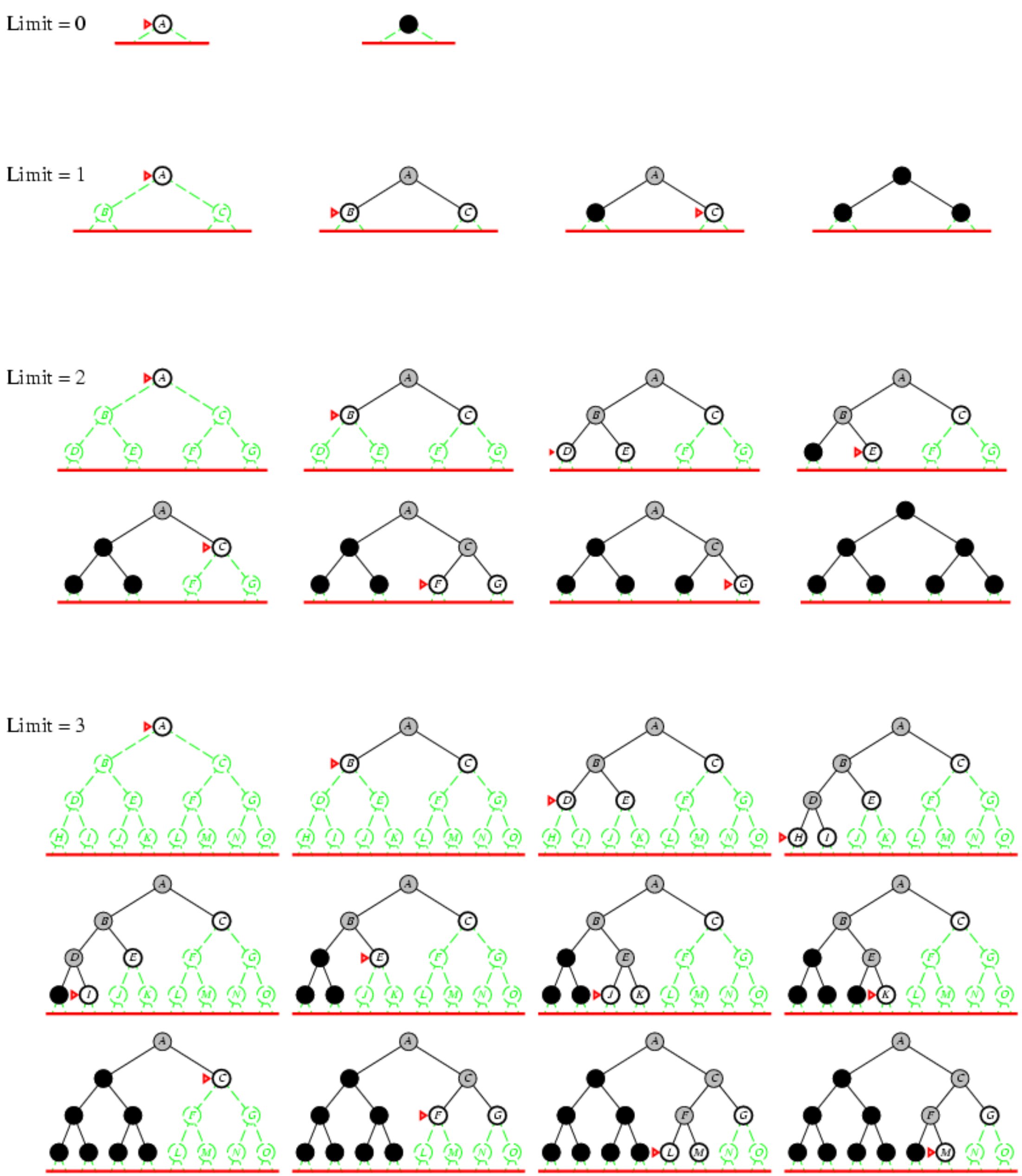
- by gradually increasing the depth limit
- combines the benefits of BFS and DFS
  - **completeness** (when the branching factor is finite)
  - **optimal** (when the path cost is a non-decreasing function of the depth of the node)
  - low **memory** consumption **O(bd)**
  - What about **time complexity**?
    - $d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d = \mathbf{O(b^d)}$
    - in fact, this is even better than BFS, which explores one more level
      - Ex. (b= 10, d = 5): BFS = 1.111.100, DLS = 111.110, IDS = 123.450

Iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

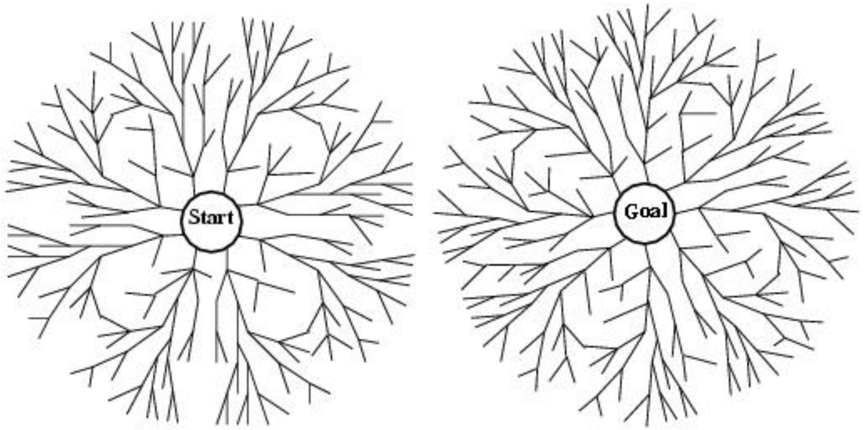
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or fail-
ure
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Iterative deepening (example)



We can run two simultaneous searches – **one forward from the initial state and the other backward from the goal** (hoping that the two searches meet in the middle).



Why?

- $\mathbf{b^{d/2} + b^{d/2} << b^d}$
- Ex. (b= 10, d = 6): BFS = 11.111.100, BiS = 22.200

How?

- Before expanding a node, verify that it is not present in the frontier of the other search tree.
- One of the frontiers must be kept in memory $\mathbf{O(b^{d/2})}$ so the intersection check can be done. With a hash table it will take constant time.
- If using breadth-first search, we obtain a **complete algorithm** (still the first solution found may not be optimal and some additional search is necessary to make sure there isn't any shortcut).

Forward search explores the set of states, but **what is explored during backward search and how?**

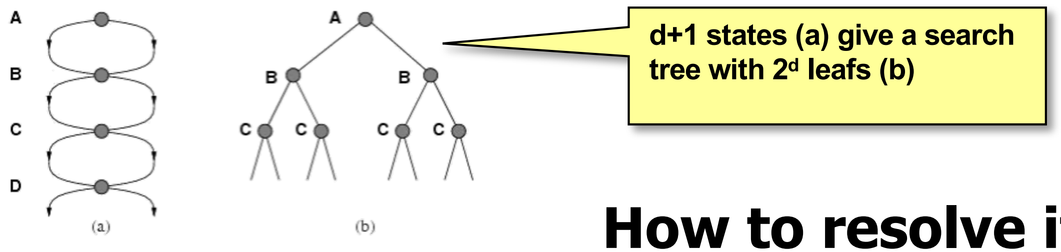
- If the goal is a single state, we can go through the states (path Arad → Bucharest).
- If the goal is a set of goal states given explicitly (vacuum world), we can use a **dummy state** such that all the goal states are its predecessors or we can use a **meta-state** to describe a set of states.
- The worst case is when there is an **abstract description of the goal state** (8-queens).

Another problem is **definition of predecessors** of a state.

- The predecessors of n are all those states that have n as a successor.
- Again, the most complex case is when the state transition is defined using a general function.

So far we ignored one of typical problems of tree search – **expansion of already visited states**.

- sometimes, it is not a problem (a good problem formulation, ex. 8-queens)
- sometimes, the search tree is much bigger than the search space



How to resolve it?

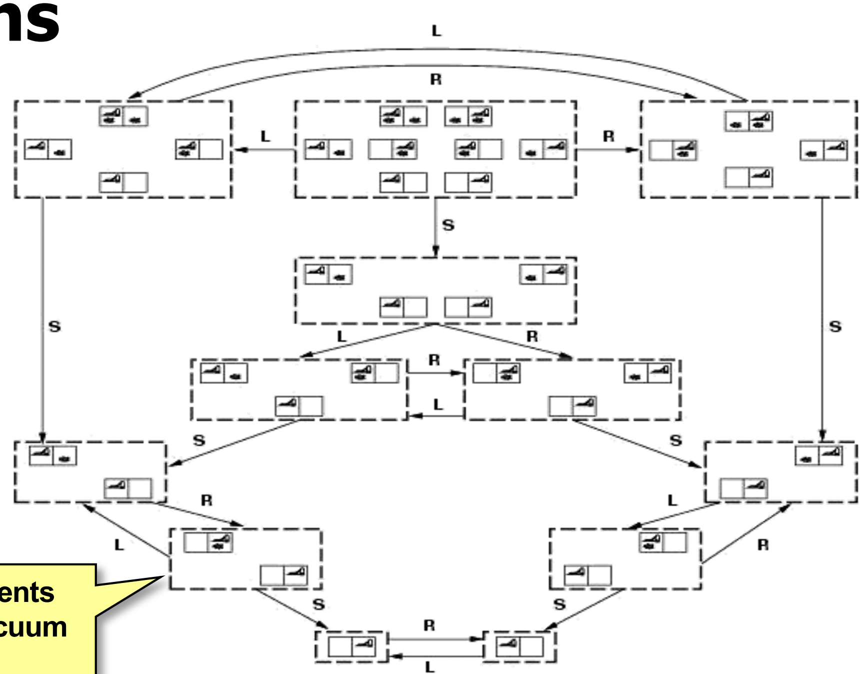
- remember already expanded states – **closed states**
- if the closed state is visited again, it is treated as it has no successors

function GRAPH-SEARCH(*problem*, *fringe*) **returns** a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
    loop do
        if *fringe* is empty **then return** failure
        node ← REMOVE-FRONT(*fringe*)
        if GOAL-TEST[*problem*](STATE[*node*]) **then return** SOLUTION(*node*)
        if STATE[*node*] is not in *closed* **then**
            add STATE[*node*] to *closed*
            fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)

So far we assumed static, fully observable, discrete and deterministic environment.

What if some **information is missing**?

- **no sensors** (the agent does not know the current state)
  - work with **belief states** – sets of real states – looking for a belief state containing only goal states (*conformant problems*)
- **non-deterministic actions**
  - plans with alternatives (*contingency problems*)
- **unknown actions**
  - solved by exploration (*exploration problems*)



The belief state represents a set of real states (vacuum world).

Summary

Criterion	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Complete?	YES*	YES*	NO	YES, if $l \geq d$	YES*	YES*
Time	b^{d+1}	$b^{C^*/\varepsilon}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^{d+1}	$b^{C^*/\varepsilon}$	bm	bl	bd	$b^{d/2}$
Optimal?	YES*	YES*	NO	NO	YES*	YES*

b – the branching factor
d – the depth of the shallowest solution
C* – the cost of optimal solution
ε – the minimal step cost (the minimal action cost)
m – the maximum depth of the search tree
l – the depth limit
* complete if b is finite (BFS)
optimal if step costs are all identical

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz