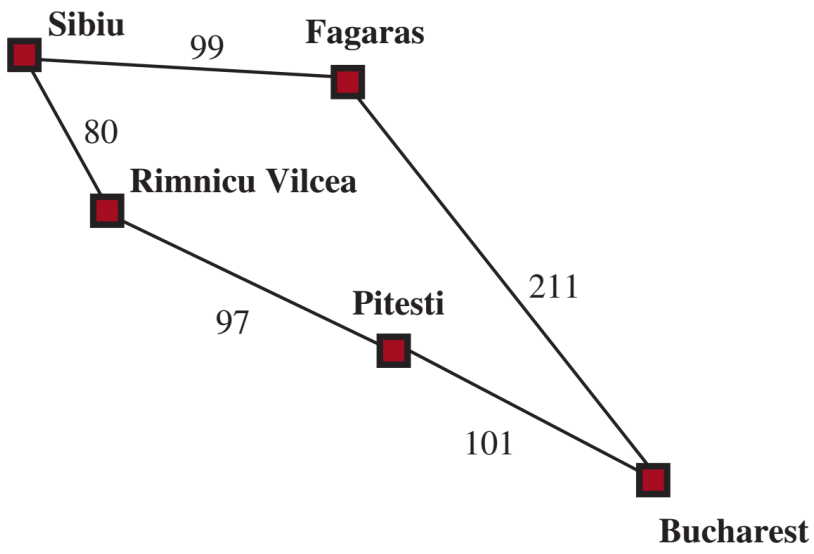
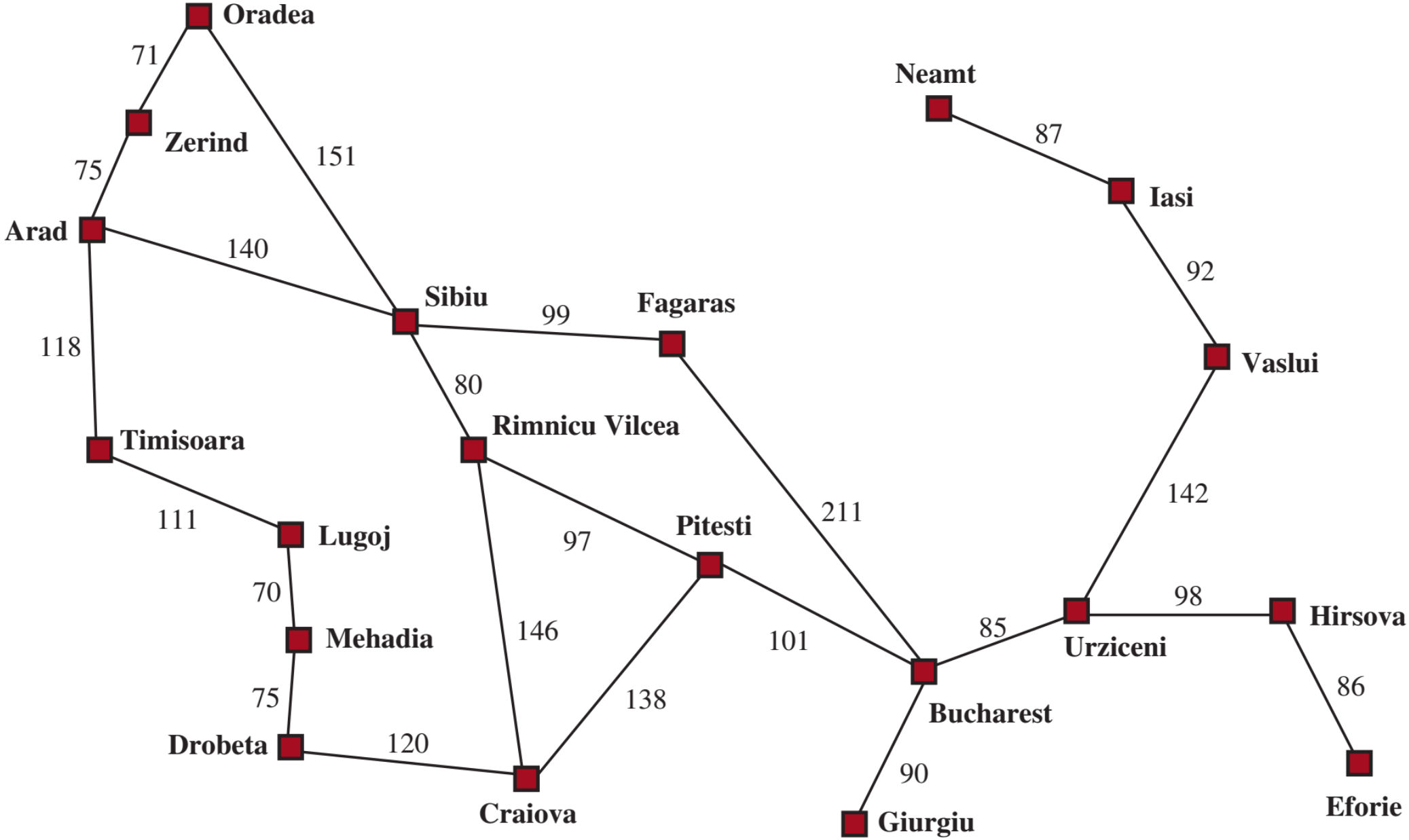
Lecture 3
Solving Problems by Searching

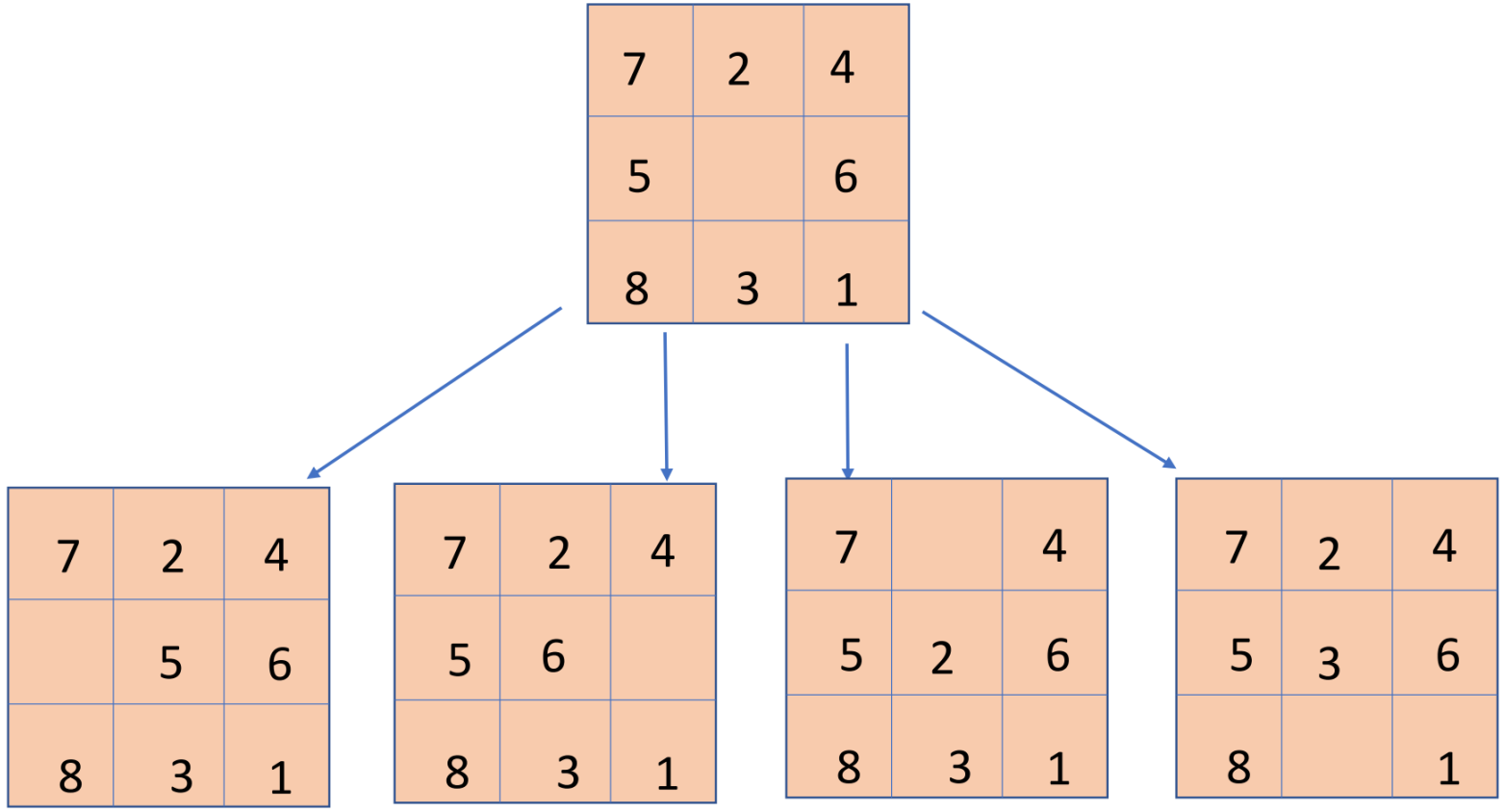
Gleb Sizov
Norwegian University of Science and Technology





A simplified road map of part of Romania, with road distances in miles.

8-puzzle



YouTube - Infinite Mario AI

Problem-solving process

> For fully observable and deterministic environments.

1. **Goal formulation**, e.g. reach Bucharest
2. **Problem formulation** - description of states and actions necessary to reach the goal
3. **Search** - **simulate** sequences of actions in the world model to find a sequence that reaches the goal.
   This sequence of actions is a **solution**.
4. **Execution** - execute the actions in the solution, one at a time

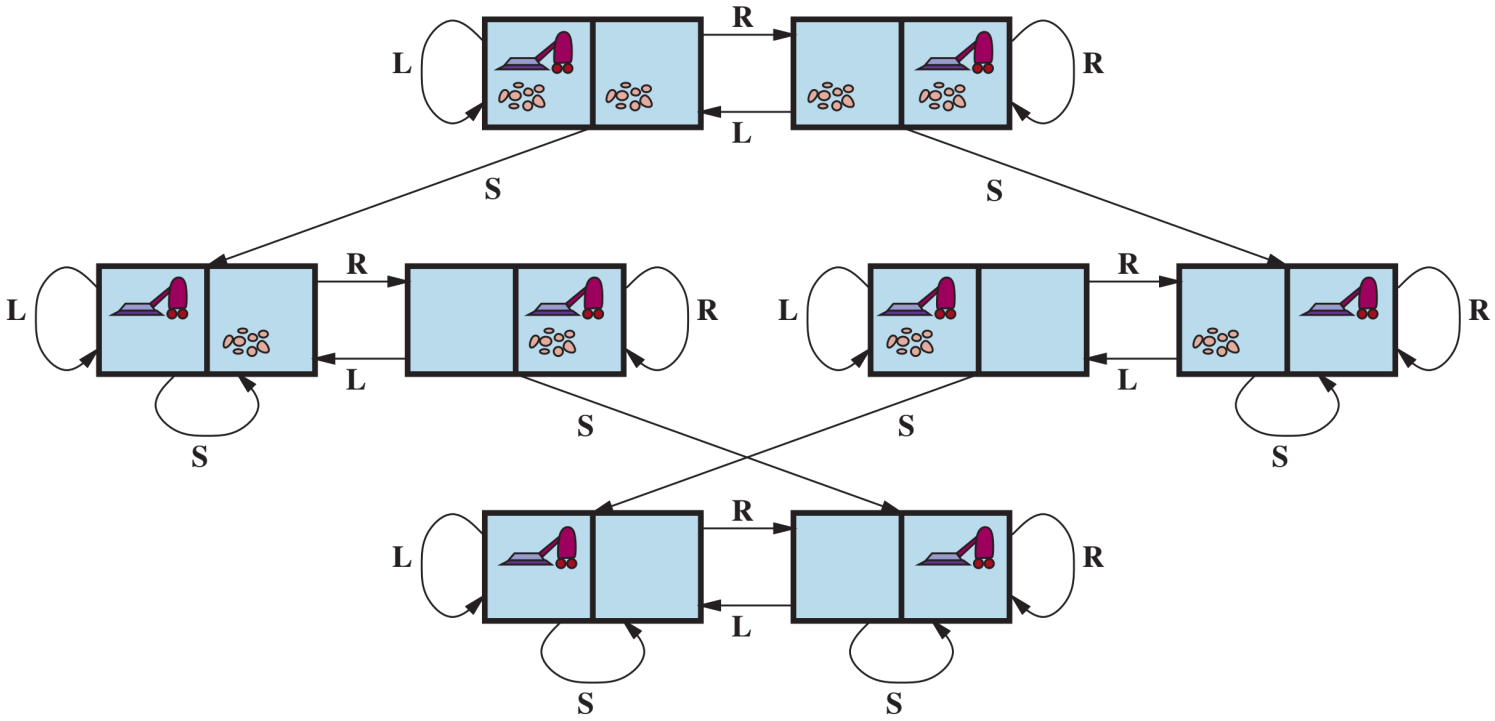
Problem formulation - Romania example

1. State-space, e.g. locations on the map
2. Initial state, e.g. Arad
3. Goal states: $IsGoal(Bucharest) = True$
4. Actions: $Actions(Arad) = ToSibiu, ToTimisoara, ToZerind$
5. Transition model: $Result(Arad, ToZerind) = Zerind$
6. Action cost function: $ActionCost(s, a, s')$

> A good problem formulation has the right level of abstraction.

Problem formulation - Vacuum world

1. State-space?
2. Initial state?
3. Goal states?
4. Actions?
5. Transition model?
6. Action cost function?



Real-world problems - Route finding

1. Airline travel planning
2. Military operations planning
3. Routing of video stream
4. Robotic navigation

Real-world problems - Touring problems

> **Traveling salesman problem** - find shortest route that visits each location once and returns to initial location.

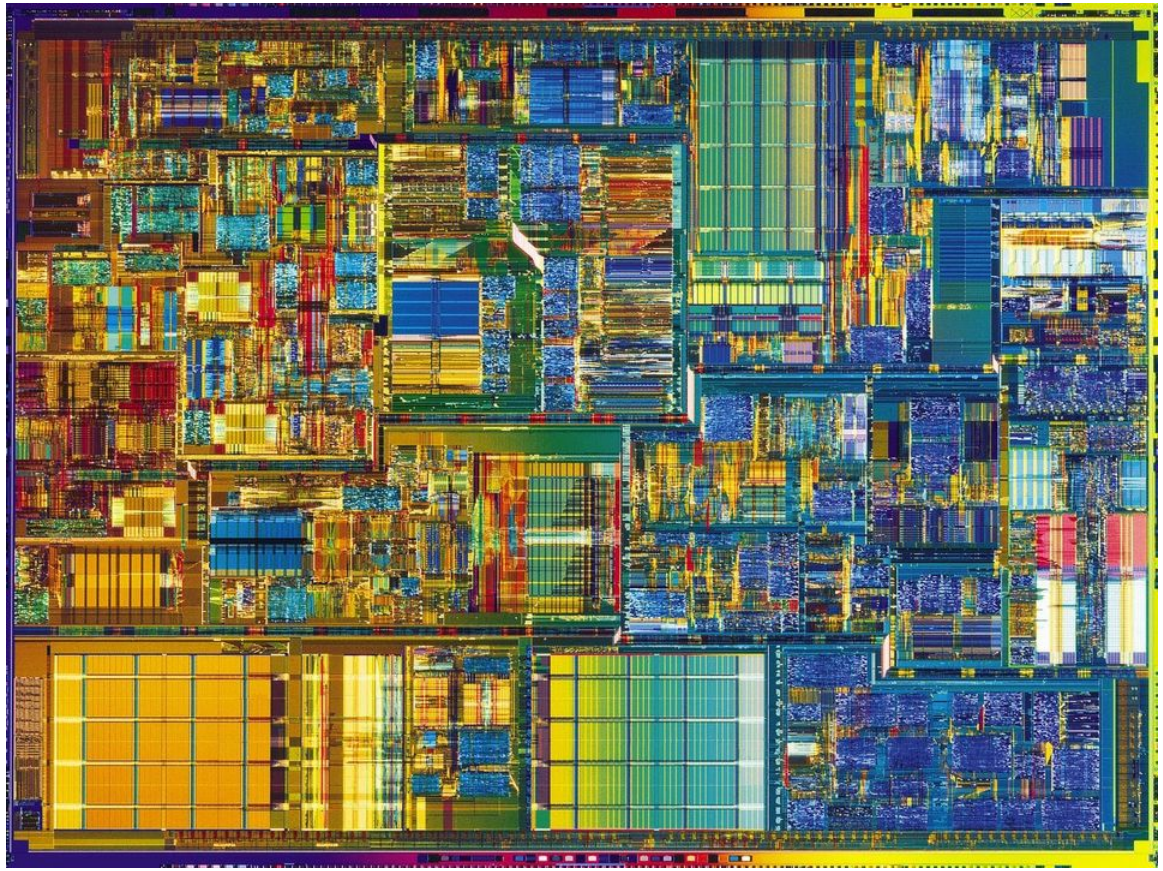
1. Delivery services
2. Automatic circuitboard drills
3. Stocking machines

Real-world problems - Assembly problems

> Find an order in which to assemble the parts of some object

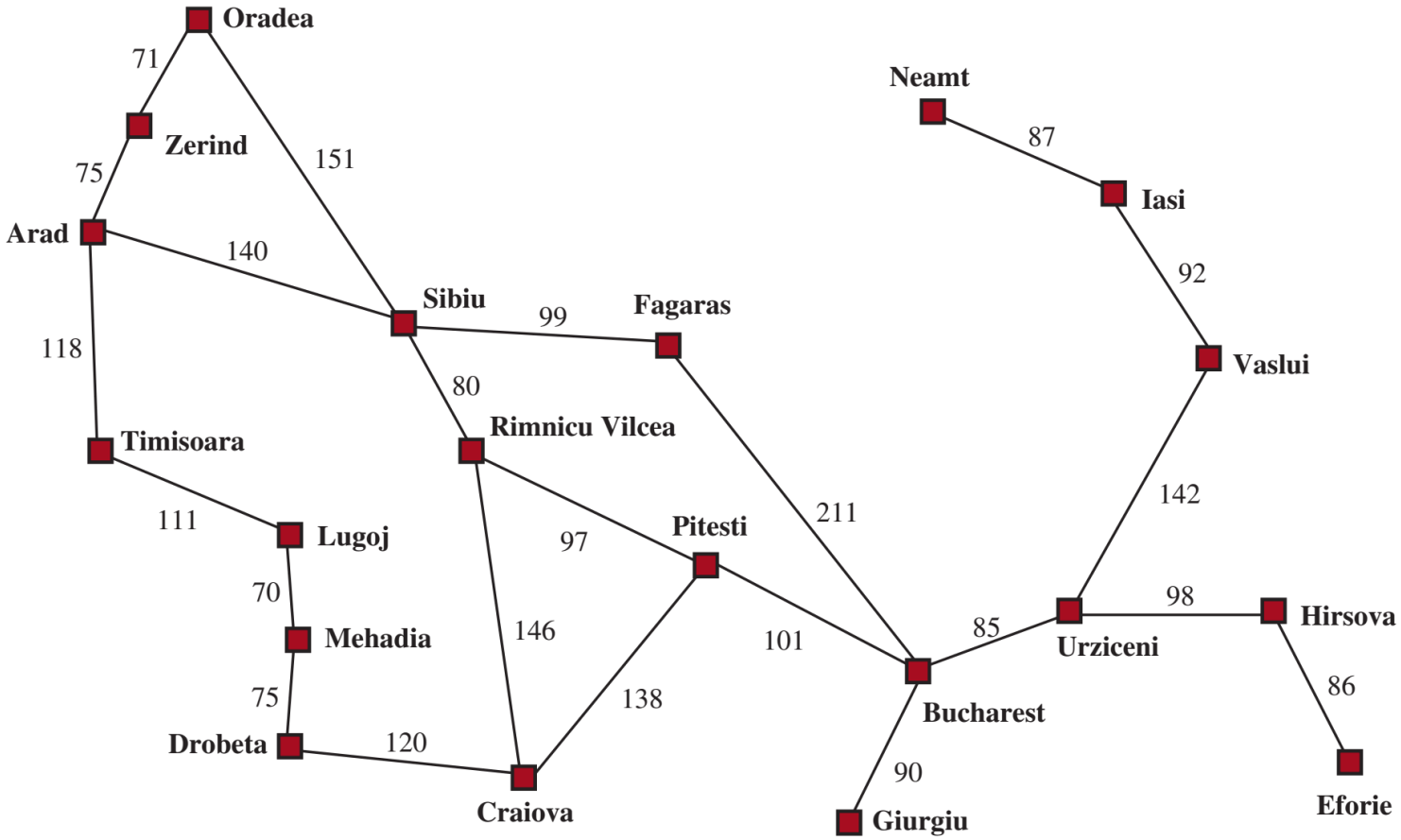
1. Manufacturing
2. Protein design

Very large-scale integration (VLSI) layout

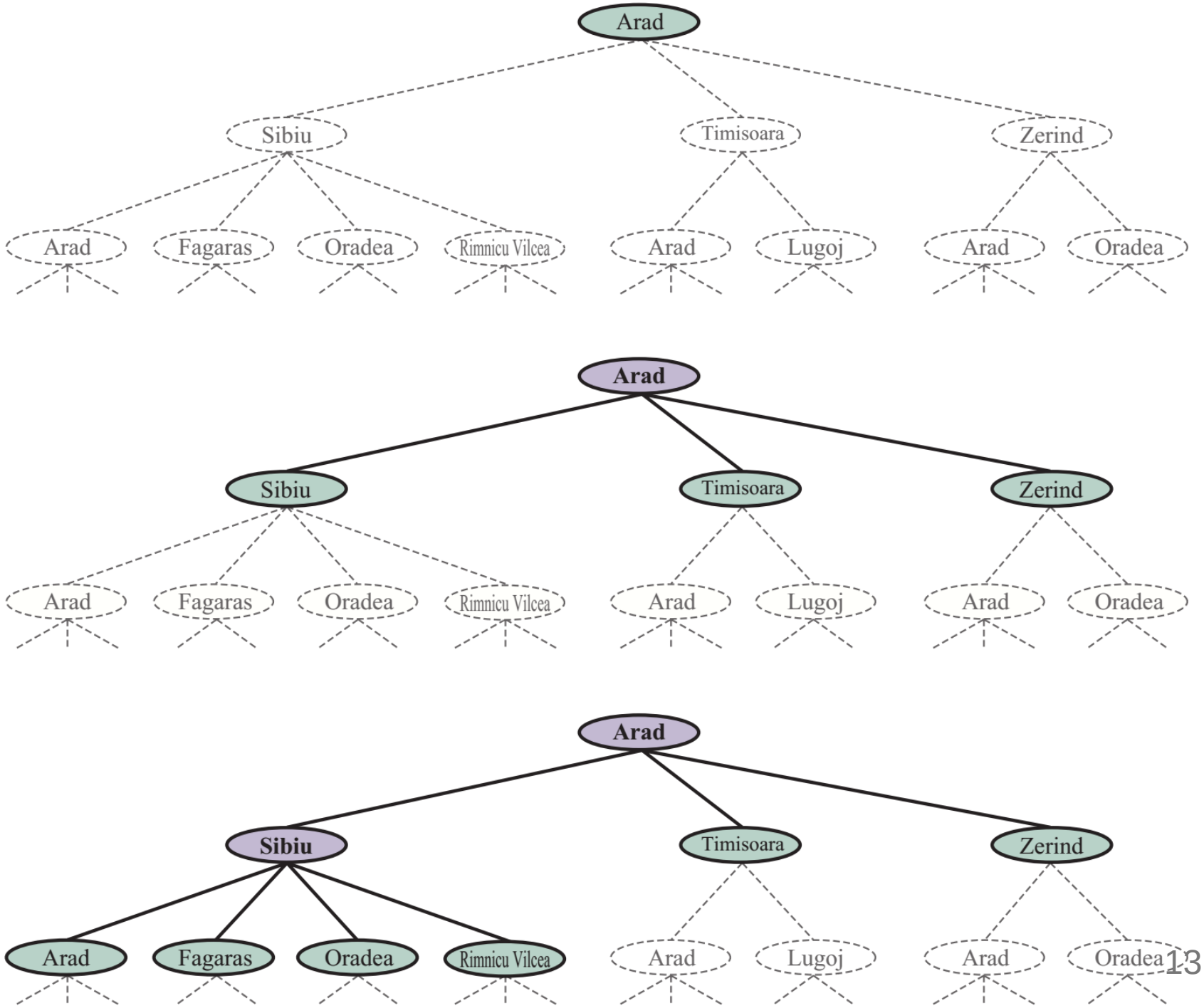


Search algorithm

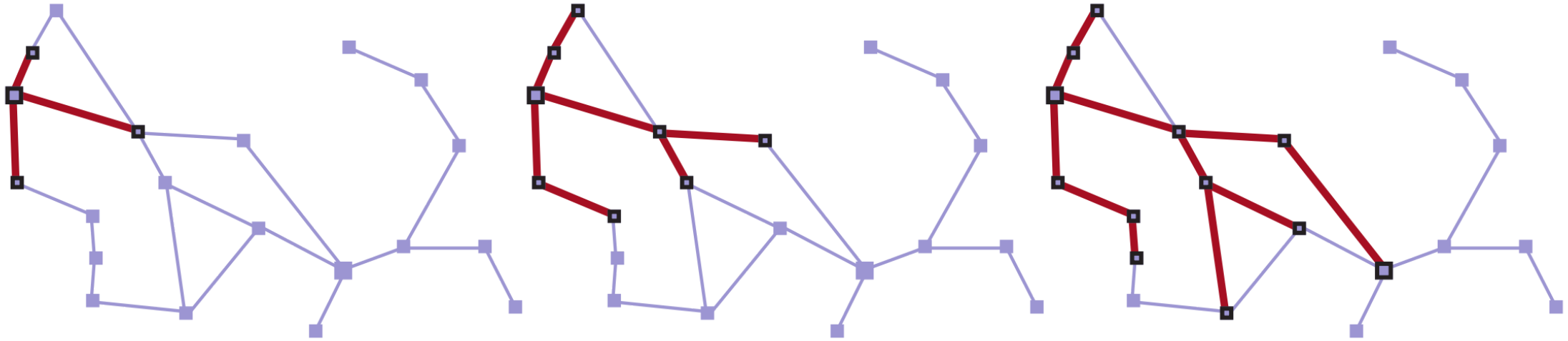
function $Search(problem)$ **returns** a solution or *failure*



Search trees



Search trees



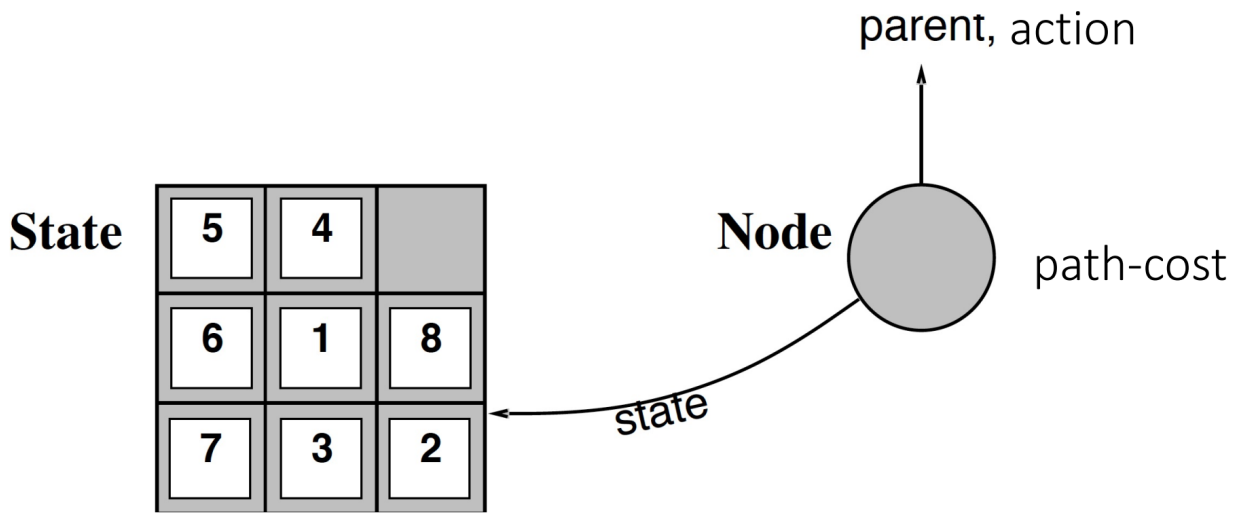
Best-first search

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
    node ← NODE(STATE=problem.INITIAL)
    frontier ← a priority queue ordered by f, with node as an element
    reached ← a lookup table, with one entry with key problem.INITIAL and value node
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
                reached[s] ← child
                add child to frontier
    return failure

function EXPAND(problem, node) yields nodes
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s' ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
        yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Node data structure

- node.STATE
- node.PARENT
- node.ACTION
- node.PATH-COST



Frontier operations

- $IsEmpty(frontier)$- returns true only if there are no nodes in the frontier
- $Pop(frontier)$ - removes the top node from the frontier and returns it.
- $Top(frontier)$ - returns (but does not remove) the top node of the frontier.
- $Add(node, frontier)$ - inserts node into its proper place in the queue.

Frontier data structures

- Priority queue - best-first search
- FIFO queue - breadth-first search
- LIFO queue (stack) - depth first search

Dealing with redundant paths

Cycle (loop) - special case of redundant path

1. Remember previously reached states, e.g. best-first search
2. Don't worry about redundant path, e.g. assembly problem
3. Check for cycles but not for other redundant paths

Graph search vs tree-like search.

Measuring problem-solving performance

Completeness - is algorithm guaranteed to find a solution (or failure)?

Cost optimality - does it find a solution with the lowest path cost?

Time complexity - how long it takes to find a solution (in seconds or states and actions considered)?

Space complexity - how much memmory needed, e.g. $frontier$, $reached$?

How to measure complexity

For explicit graph:

$|V| + |E|$ - size of state-space graph

For implicit graph defined by initial state, actions and transition model:

- d - depth - number of actions in optimal path
- b - branching factor - number of successors of node to consider
- m - maximum depth of the state space (may be infinite)

Uninformed vs informed search

Uninformed search - no clue about how close a state is to the goal(s).

Informed search - have some estimate of how close a state is to the goal(s).

S1

7	5	2
	4	3
8	1	6

S2

1	2	3
4		6
7	5	8

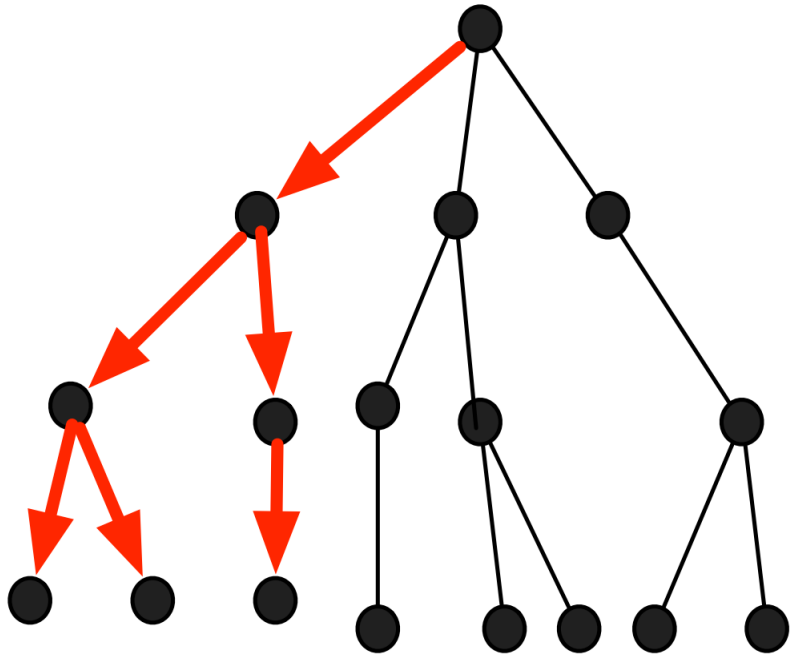
1	2	3
4	5	6
7	8	

Goal state

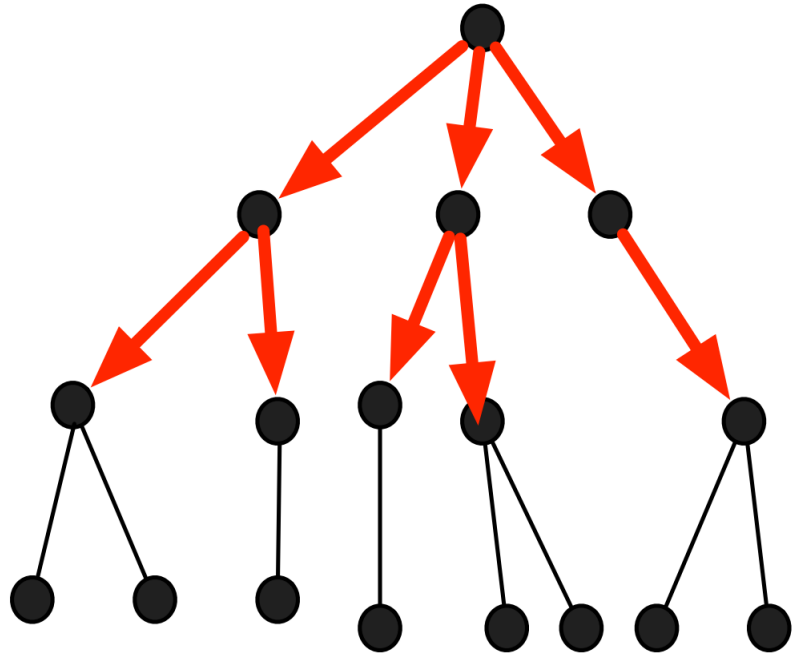
Uninformed search strategies

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Depth-first vs Breadth-fist



Depth-First



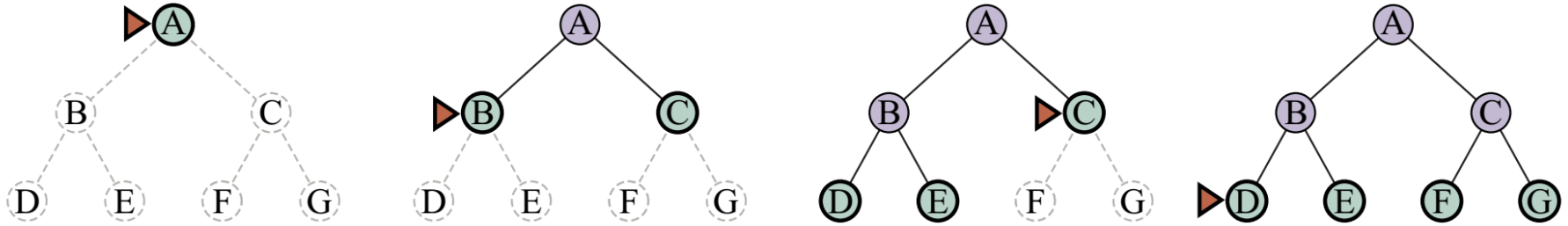
Breadth-First

Course info

1. First assignment lecture, Friday 2022-09-16 16:15.
   See Blackboard.
2. Reference group, 3-4 students.
   Let me know during break, after class or send email to odderik@ntnu.no.

Breadth-first search

> Uses FIFO, expands nodes with least depth first.



Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier ← a FIFO queue, with node as an element
    reached ← {problem.INITIAL}
     while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if problem.IS-GOAL(s) then return child
            if s is not in reached then
                add s to reached
                add child to frontier
    return failure
```

Properties of bread-first search

Complete: Yes

Cost optimal: Yes - if actions have the same cost

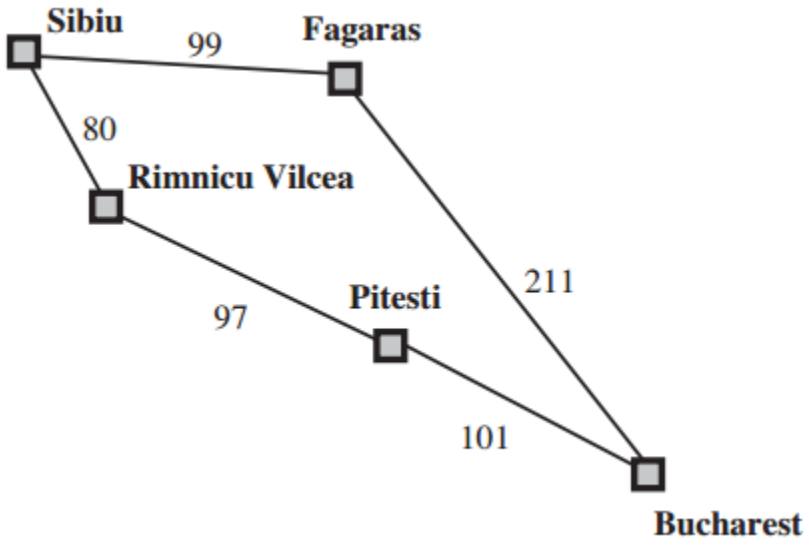
Time and space: $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Uniform-cost search

- Handles varying positive step cost
- Uses priority queue sorted by path cost
- Expands nodes with least path cost first.

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)



Properties of uniform-cost search

Complete: Yes - if actions have positive cost

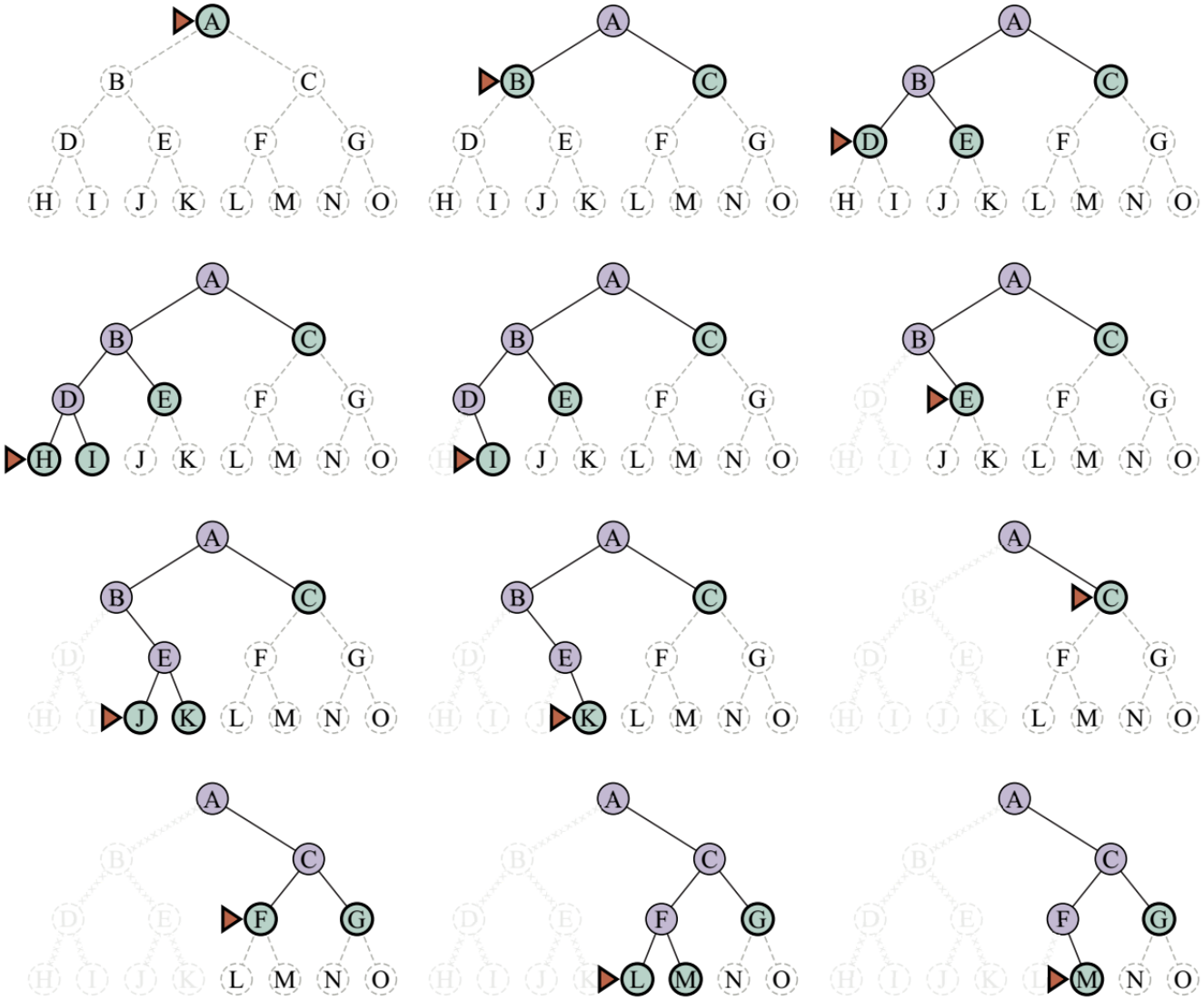
Cost optimal: Yes

Time and space: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

- C^* - optimal solution cost
- ϵ - action cost lower bound

Depth-first search

> LIFO, expands the deepest nodes first.



Properties of depth-first search

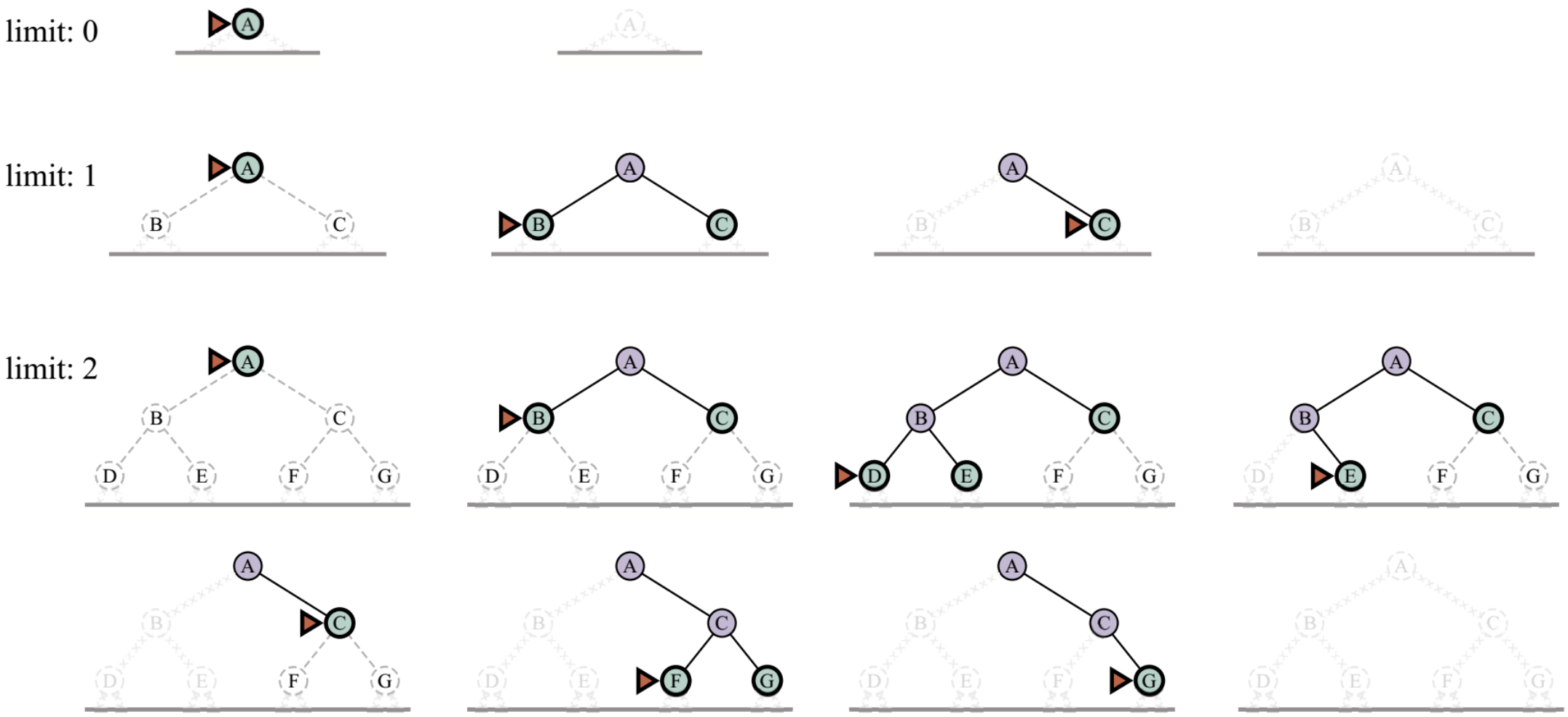
Complete: No - fails in infinite-depth spaces and spaces with loops

Cost optimal: No

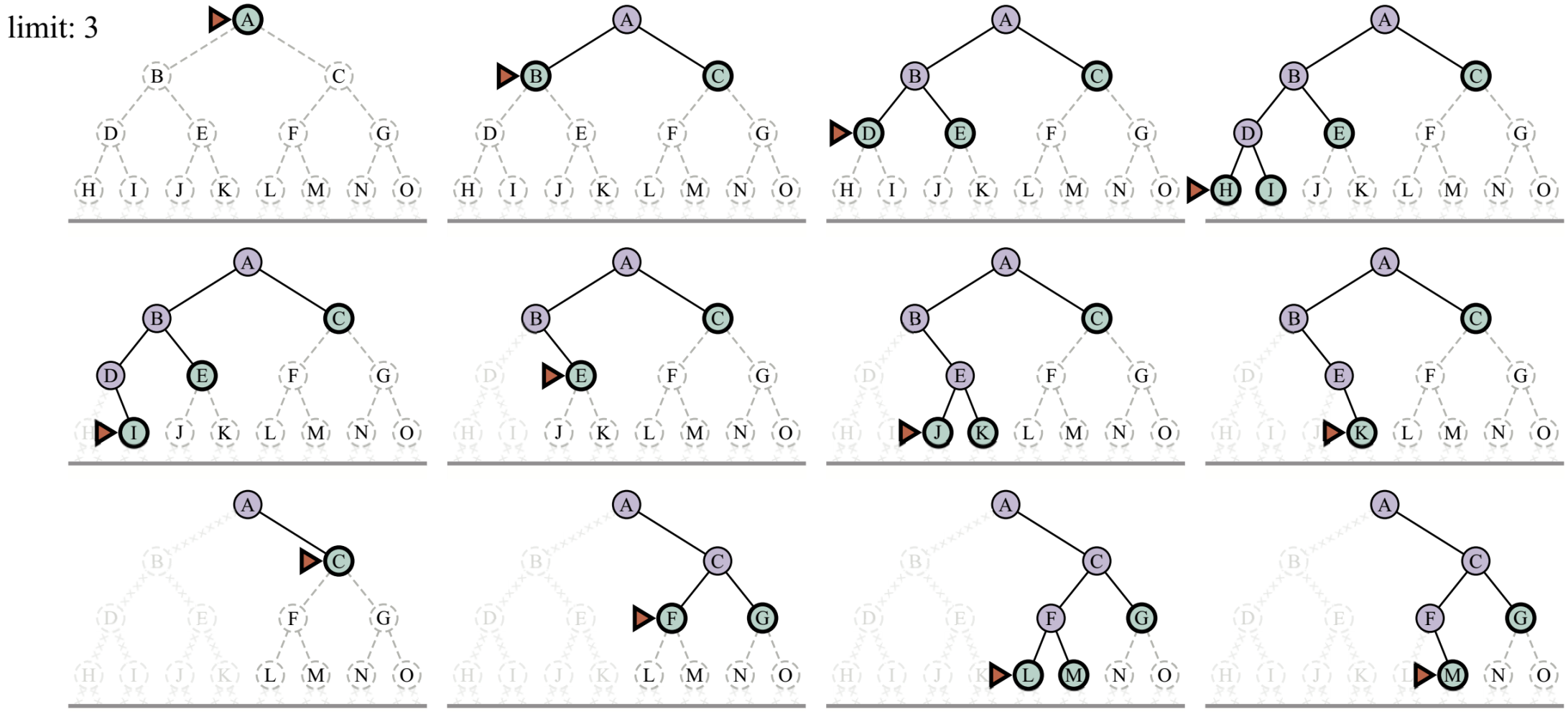
Time: O(b^m)

Space: O(bm) - linear, nice!

Depth-limited and iterative deepening search



Depth-limited and iterative deepening search



Iterative deepening search

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

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

Properties of iterative deepening

Complete: Yes

Cost optimal: Yes, if cost are equal

Time: $(d)b^1 + (d-1)b^2 + (d-2)b^3 \ldots + b^d = O(b^d)$

Space: $O(bd)$

> In general, iterative deepening is the preferred uninformed search method when the search state space is larger than can fit in memory and the depth of the solution is not known.

Uninformed strategies - Summary of the properties

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

b - branching factor

d - depth of the shallowest solution

m - maximum depth

l - depth limit

Informed (Heuristic) search

> Use domain-specific hints about the location of goals.

Heuristic function:
$h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state

Greedy best-first search

> Expand first the node with the lowest h(n)

Example heuristic:

h_{SLD}—straight-line distances to Bucharest.

Properties of greedy best-first search

Complete:

Yes in finite state space

No in infinite state space

Cost optimal: No

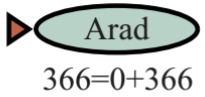
Time and space: $O(b^m)$, with good heuristic can be $O(bm)$

A* search

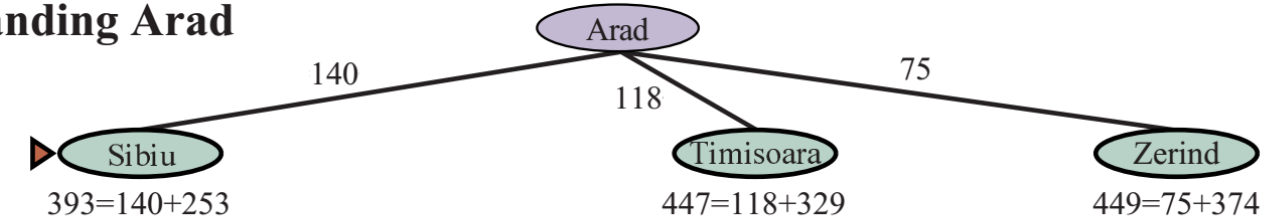
Best-first search with $f(n) = g(n) + h(n)$

- $g(n)$ - path cost from initial node to node n
- $h(n)$ - estimated cost of the shortest path from n to goal state

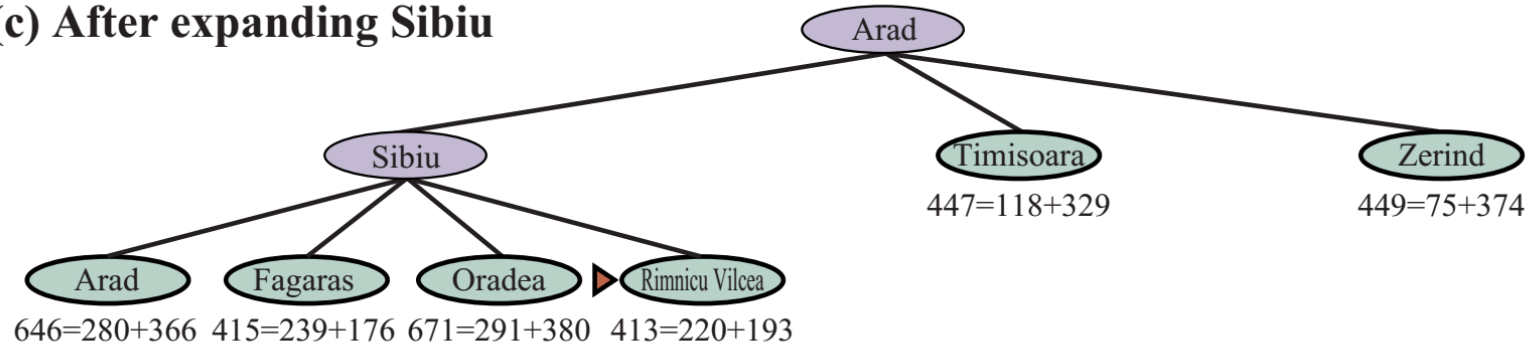
(a) The initial state



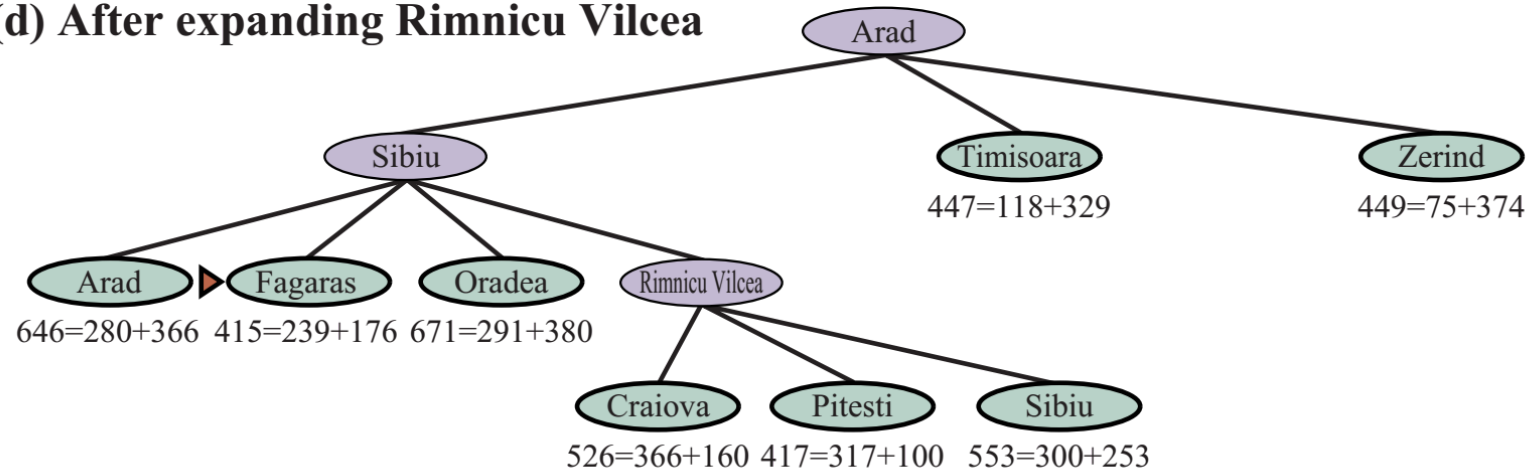
(b) After expanding Arad



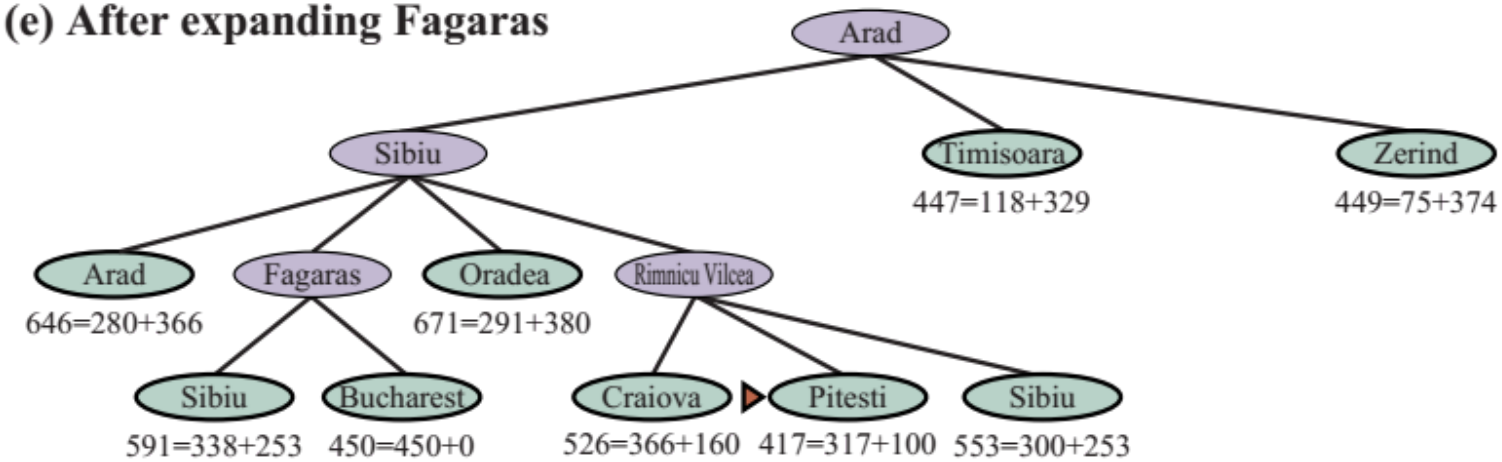
(c) After expanding Sibiu



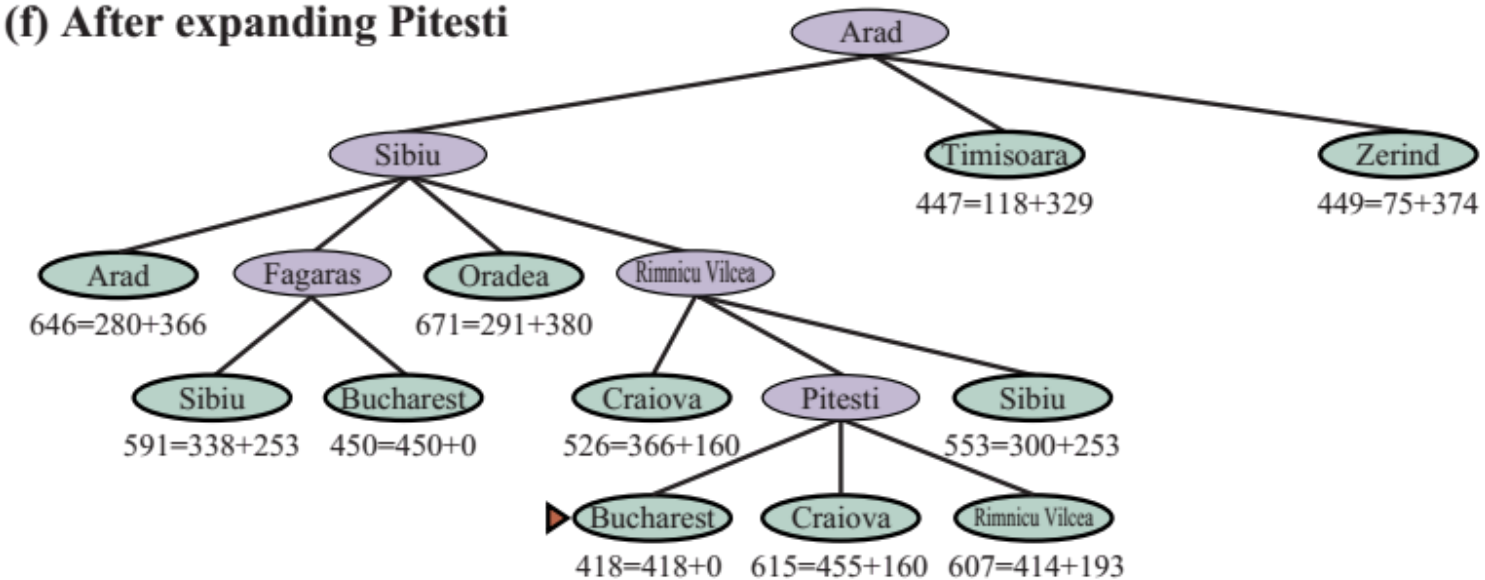
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti



Heuristic admissibility

Admissibility - an admissible heuristic never overestimates the cost to reach a goal.

It is optimistic!

S1

7	5	2
	4	3
8	1	6

S2

1	2	3
4		6
7	5	8

1	2	3
4	5	6
7	8	

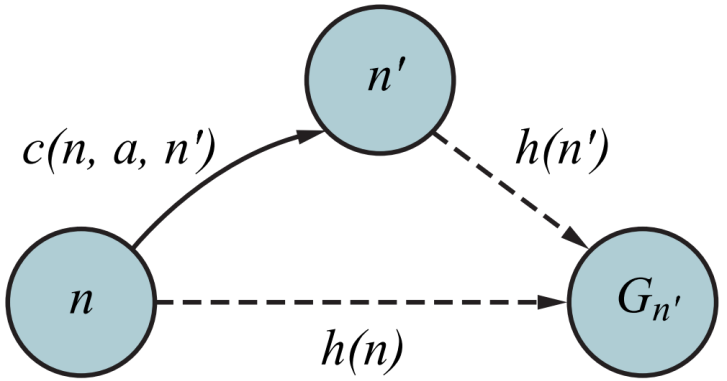
Goal state

Heuristic consistency

Consistency - $h(n) \leq c(n, a, n') + h(n')$
for every node n and every successor n' of n generated by an action a

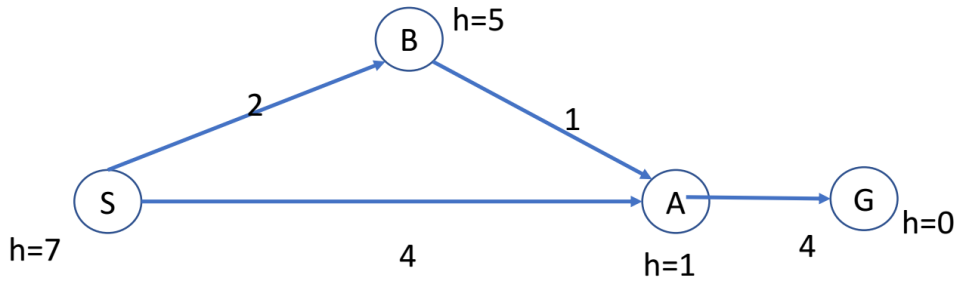
- $c(n, a, n')$ - action cost

> Every consistent heuristic is admissible, but not vice versa.

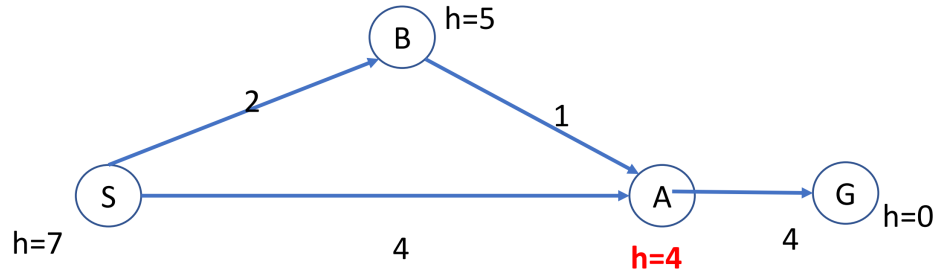


Re-expansion of nodes

A:



B:



function BEST-FIRST-SEARCH(*problem*, f) **returns** a solution node or *failure*
  node ← NODE(STATE=*problem*.INITIAL)
  frontier ← a priority queue ordered by f, with *node* as an element
  reached ← a lookup table, with one entry with key *problem*.INITIAL and value *node*
  while not IS-EMPTY(*frontier*) **do**
    node ← POP(*frontier*)
    if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
    for each *child* **in** EXPAND(*problem*, *node*) **do**
      s ← *child*.STATE
      if s is not in *reached* **or** *child*.PATH-COST $<$ *reached*$[s]$.PATH-COST **then**
        reached$[s]$ ← *child*
        add *child* to *frontier*
  return *failure*

function EXPAND(*problem*, *node*) **yields** nodes
  s ← *node*.STATE
  for each *action* **in** *problem*.ACTIONS(s) **do**
    s' ← *problem*.RESULT(s, *action*)
    cost ← *node*.PATH-COST + *problem*.ACTION-COST(s, *action*, s')
    yield NODE(STATE=s', PARENT=*node*, ACTION=*action*, PATH-COST=*cost*)

Properties of A*

Complete:

Yes, for positive action costs, state space has a solution or is finite

Cost optimal:

Yes, for tree search with admissible heuristic

Yes, for graph search with consistent heuristic

Time and space: $O(b^{*d})$, for good heuristic $b^* < b$

To be continued ...

- "Good enough" solution
- Effect of heuristics on performance
- How to construct heuristics
- Bidirectional search
- Memory-bounded search

What search algorithm to use here and why?