TDDE56: Problem Solving as Search

Fredrik Heintz
Dept. of Computer Science, Linköping University
fredrik.heintz@liu.se
@FredrikHeintz

LINKÖPINGS UNIVERSITET

Problem-Solving through Search

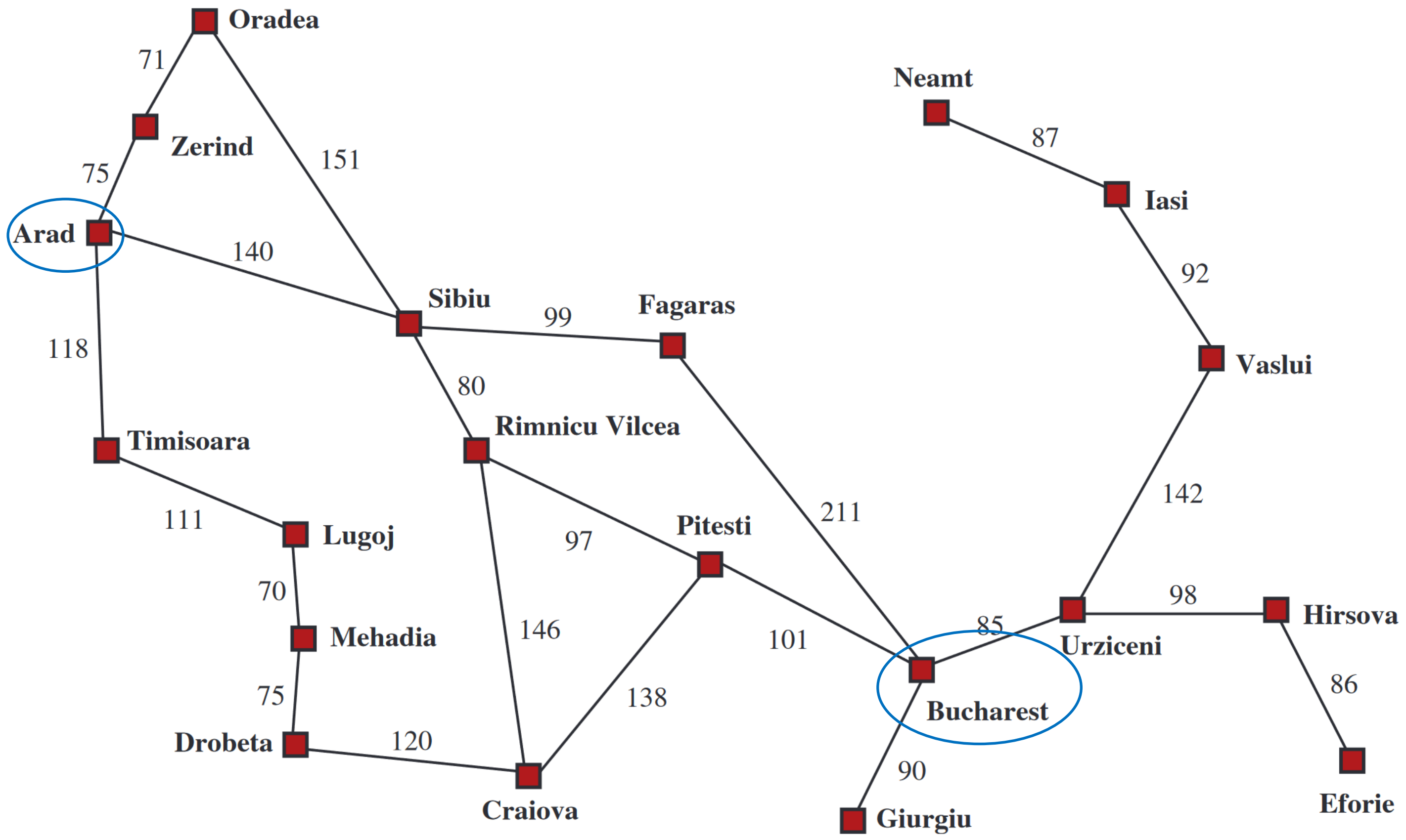


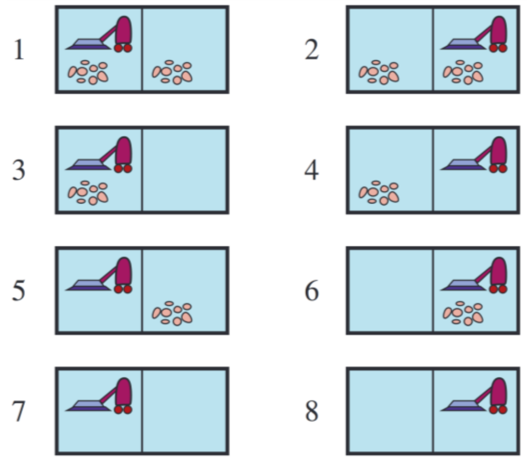
Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

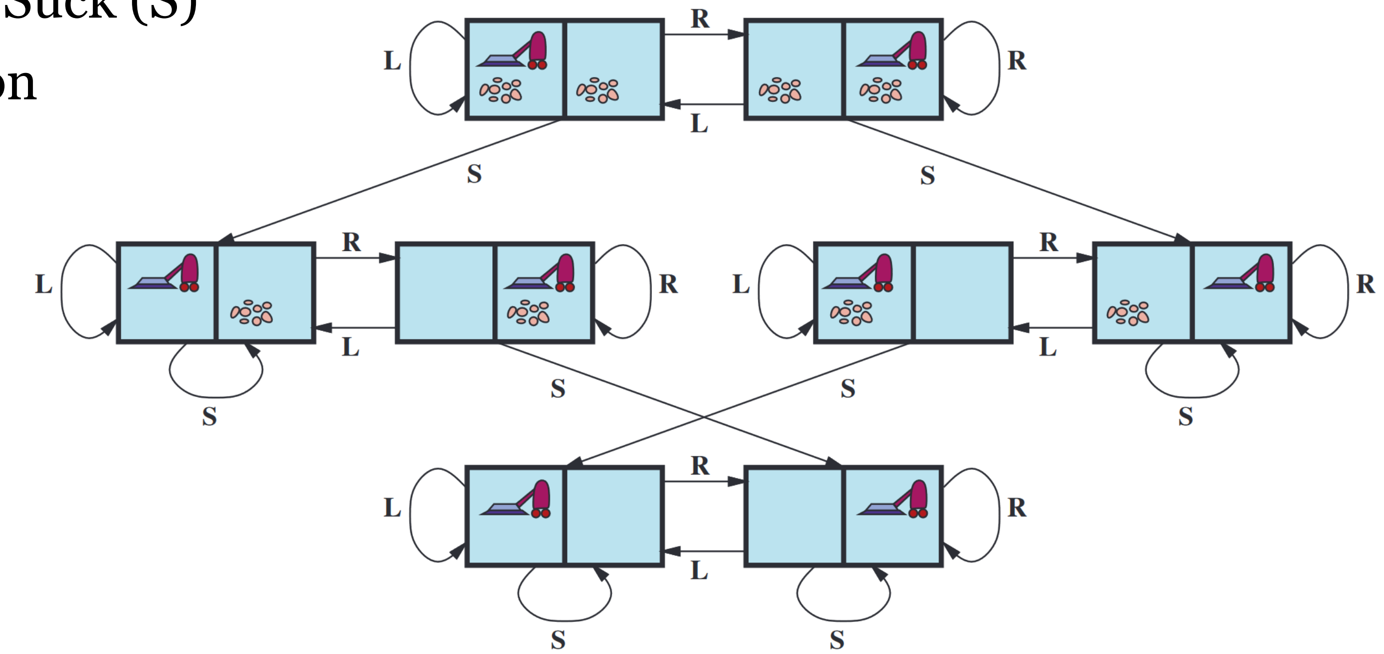
Search – Problem Definition

- **Initial State**: The state in which the agent starts or initial condition of the agent.
- **States**: All states that are reachable from initial state by any sequence of actions or all possible states that the agent can take. This is also referred to as State space.
- **Actions**: All possible actions that the agent can execute. Specifically, it provides the list of actions, that an agent can perform in a particular state. This is also referred to as Action space.
- **Transition Model**: This property describes the results of each action taken in a particular state.
- **Goal Test**: A way to check, whether a state is the goal.
- **Path Cost**: A function that assigns a numeric cost to a path w.r.t. performance measure

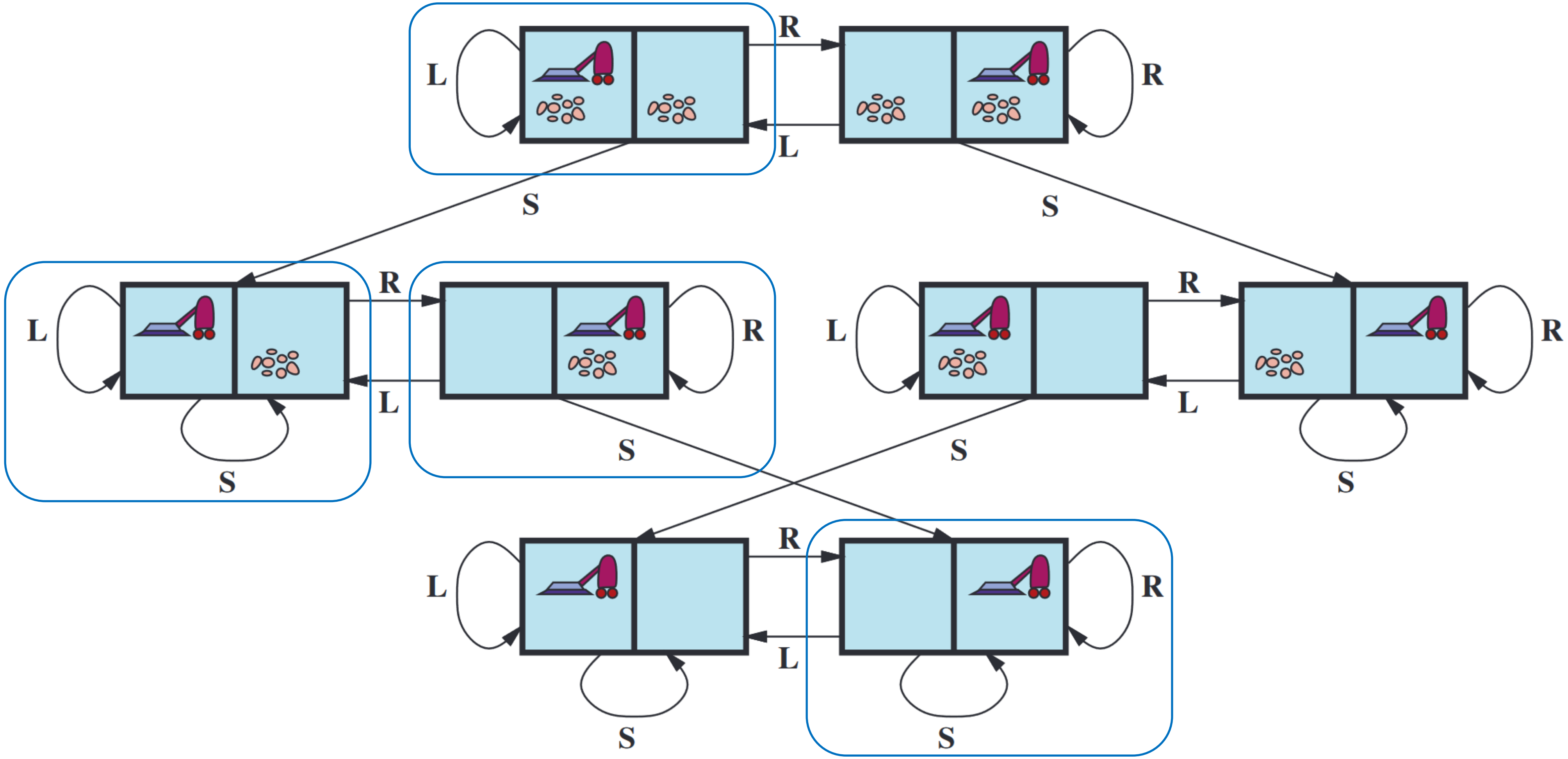
Vacuum World

- State Space: 2 positions, dirt or no dirt
- Initial State: Choose
- Goal States: States with no dirt in the rooms
- Actions: Left (L), Right (R), or Suck (S)
- Action costs: one unit per action
- Transition model:

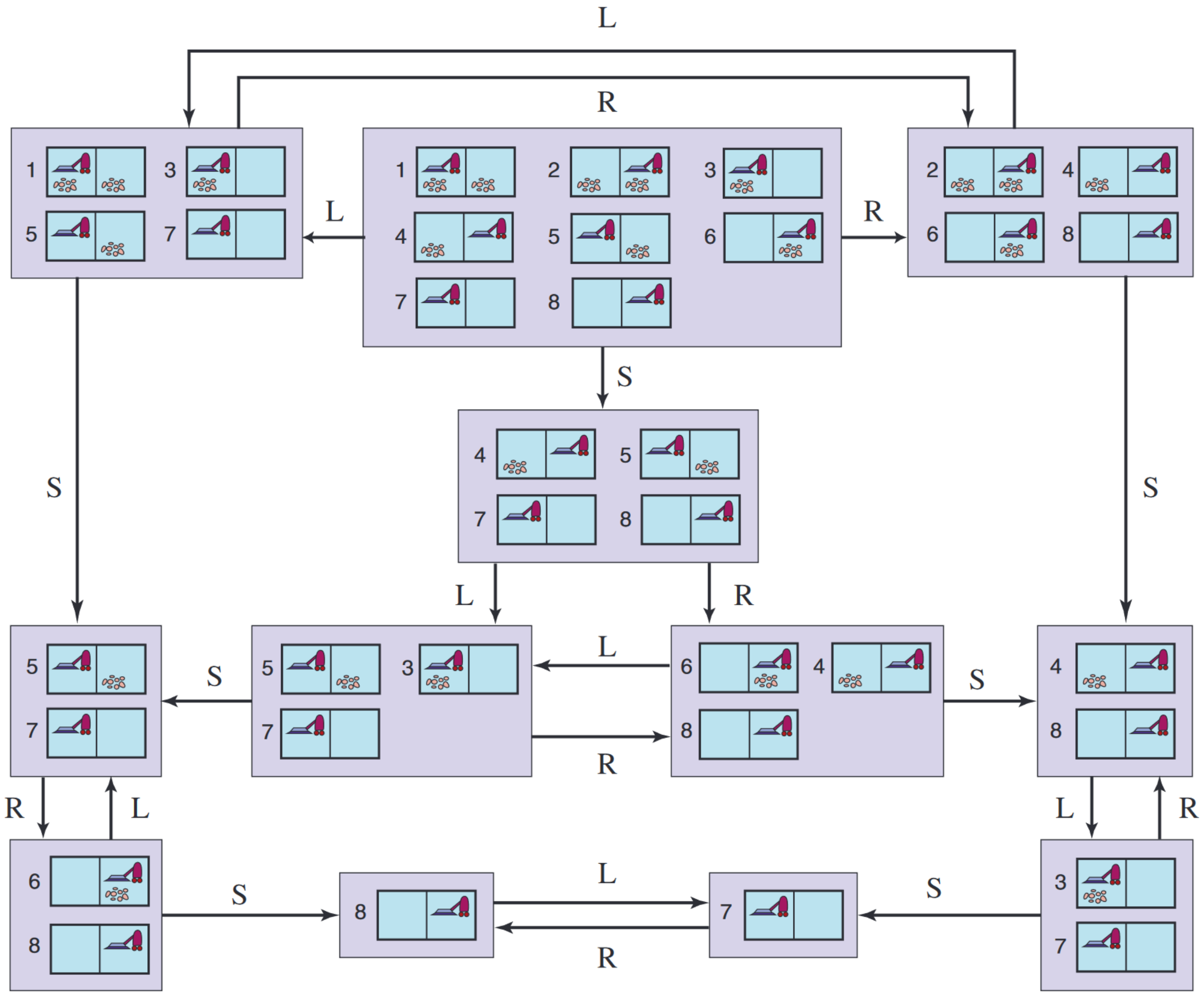




Solving the Vacuum World

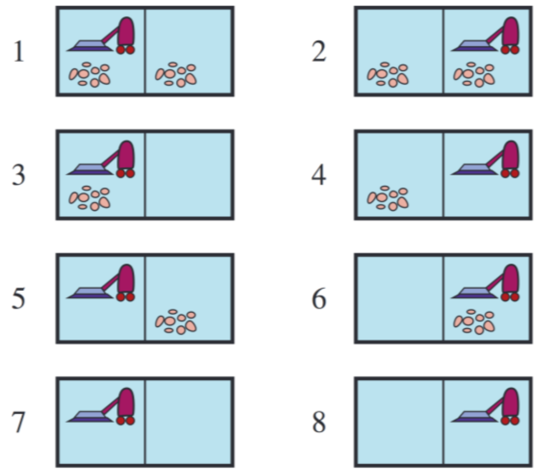


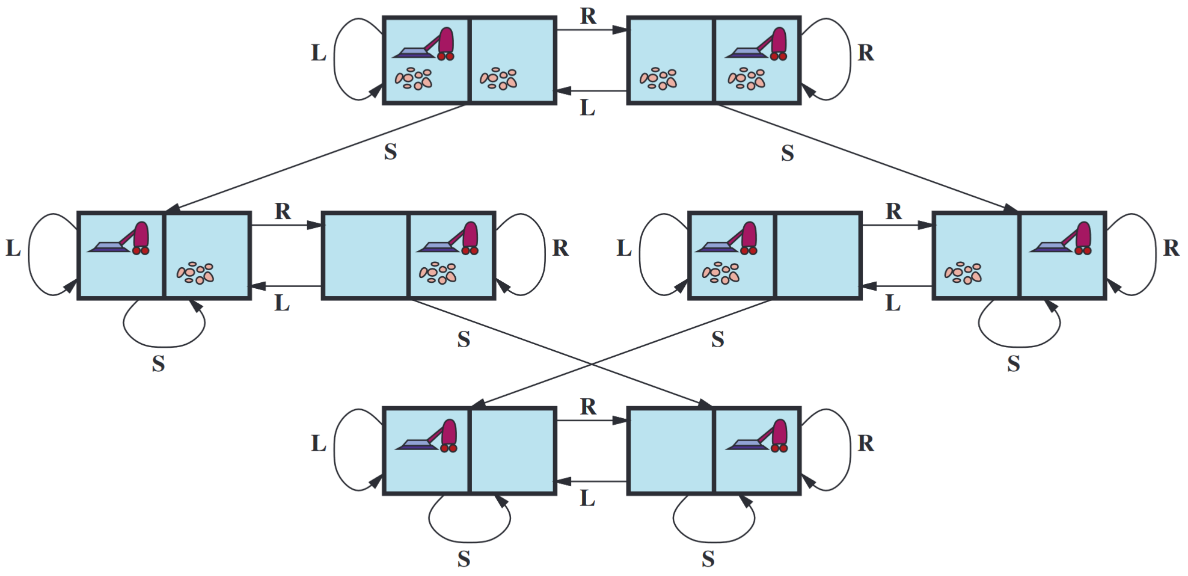
Solving the Vacuum World without Sensors



Search – Search Space

- **State space**: physical configuration
- **Search space**: abstract configuration often represented by a search tree or graph where a path is a possible solution.
- **Search tree**: representation of configurations and how they are connected by actions. A path represents a sequence of actions. The *root* is the initial state. The actions taken make the *branches* and the *nodes* are results of those actions. A node has depth, path cost and associated state in the state space.





Search Tree Construction

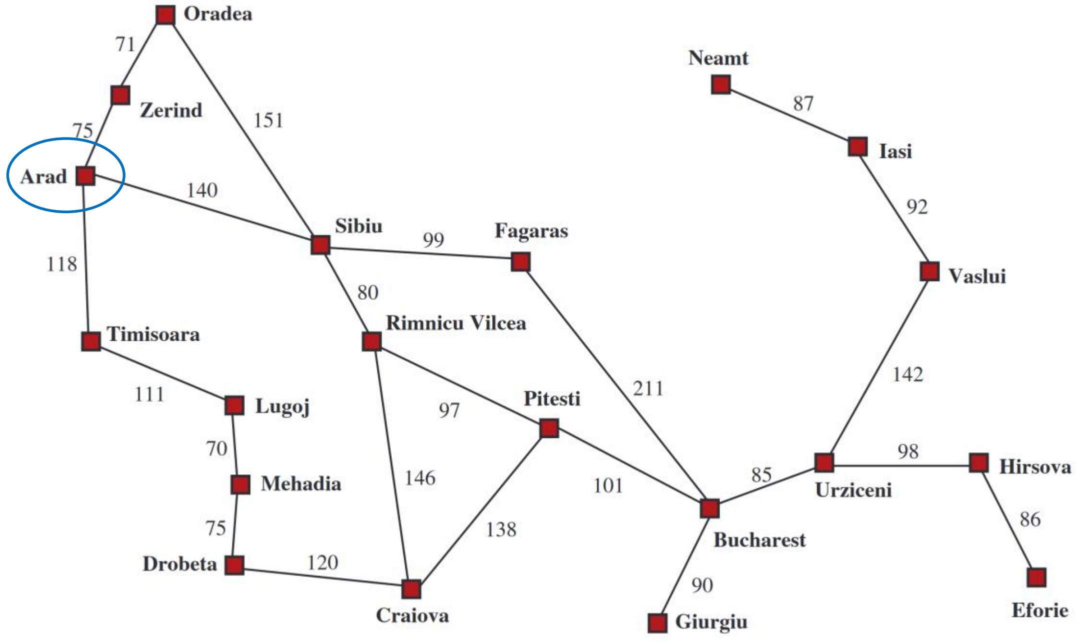
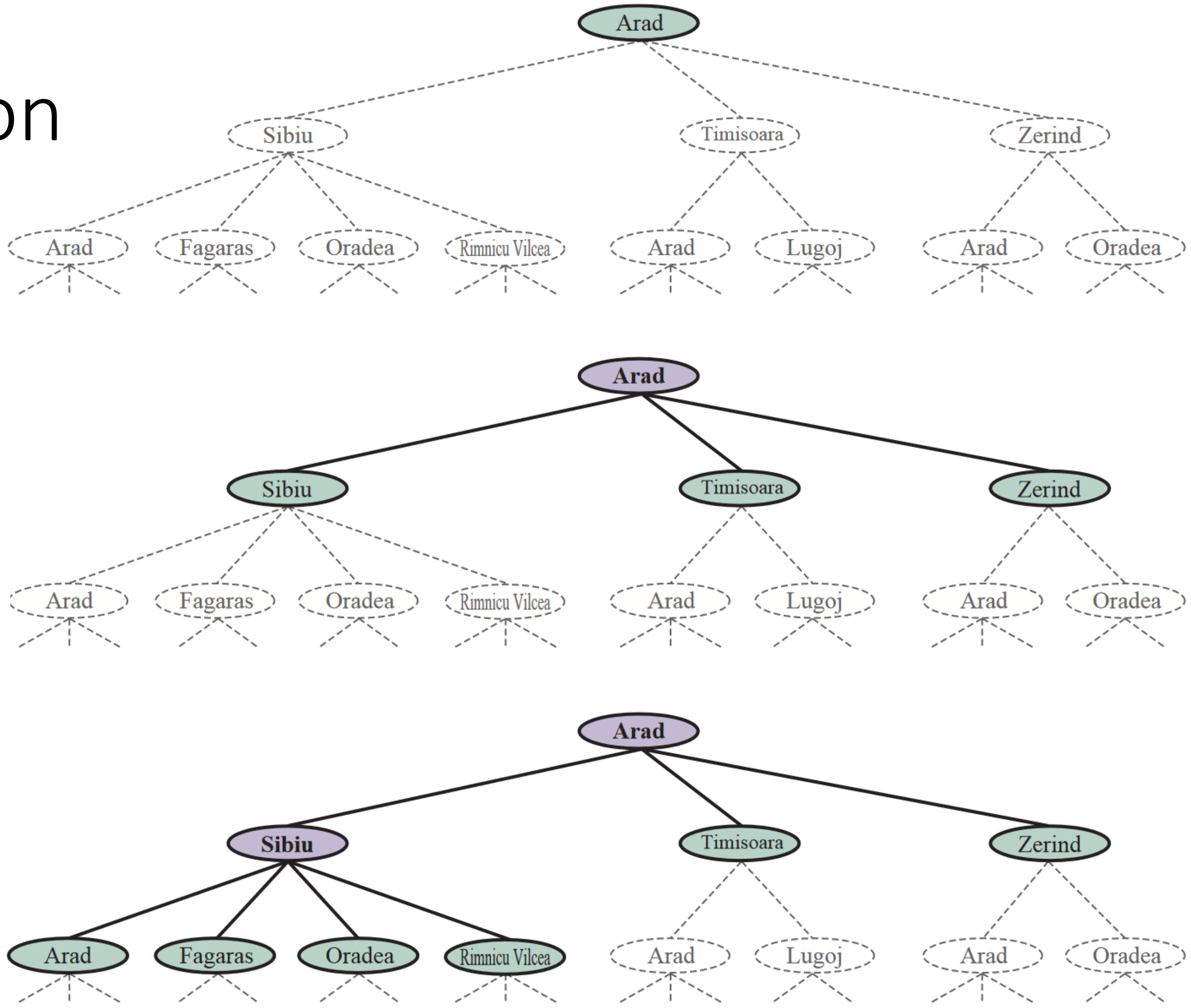


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.



Search Strategies

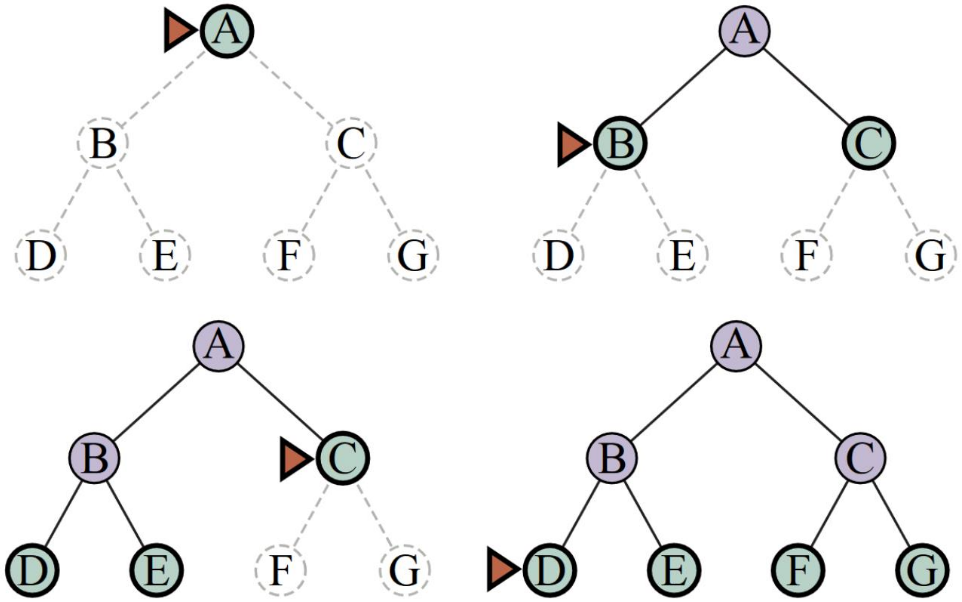
- A *strategy* is defined by picking the order of node expansion.
- Strategies can be *evaluated* along the following dimensions:
  - *Completeness* – does it find a solution if it exists?
  - *Time Complexity* – number of nodes generated/expanded
  - *Space Complexity* – maximum number of nodes in memory
  - *Optimality* – does it always find a least cost solution
- Time and space complexity are *measured* in terms of:
  - *b* – *maximum branching* factor of search tree
  - *d* – *depth of the least cost solution* in the search tree
  - *m* – *maximum length* of any path in the state space (possibly infinite)

Some Search Classes

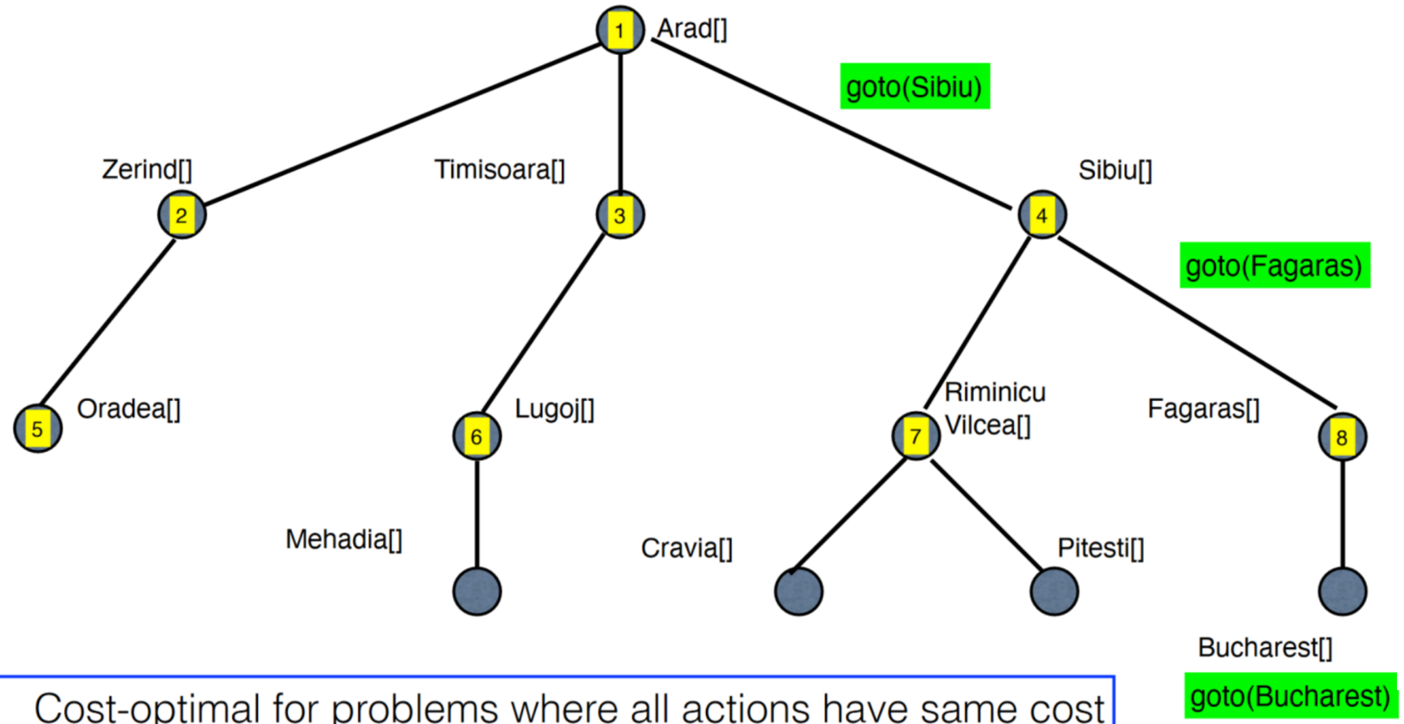
- *Uninformed Search* (Blind Search)
  - No additional information about states besides that in the problem definition
  - Can only generate successors and compare against state.
  - Some examples:
    - Breadth-first search, Depth-first search, Iterative deepening DFS
- *Informed Search* (Heuristic Search)
  - Strategies have additional information as to whether non-goal states are more promising than others.
  - Some examples:
    - Greedy Best-First Search, A* Search

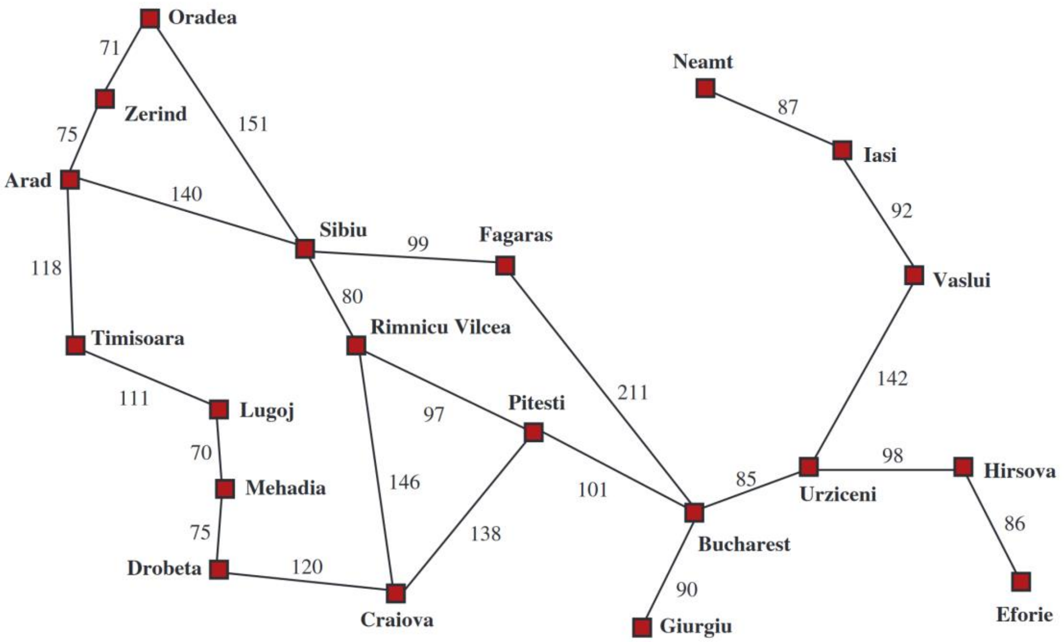
Breadth-First Search

```
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
```



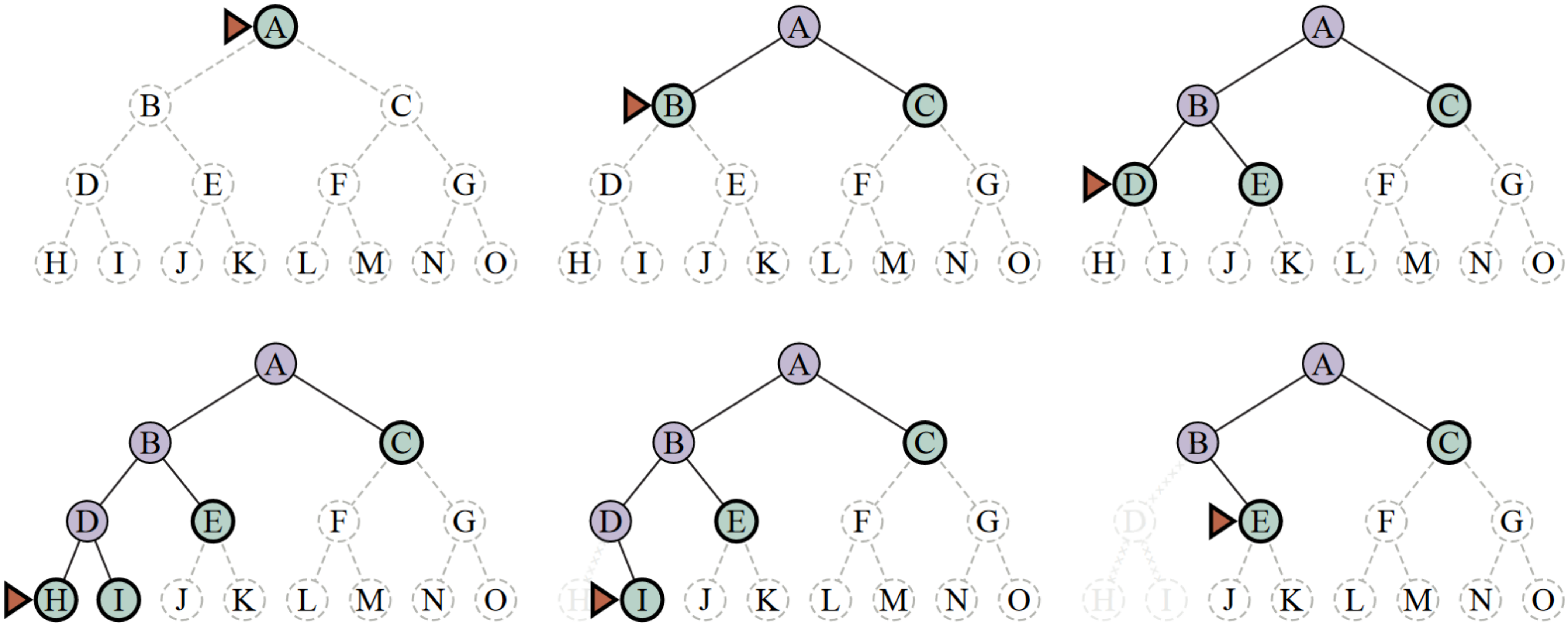
Breadth-First Search



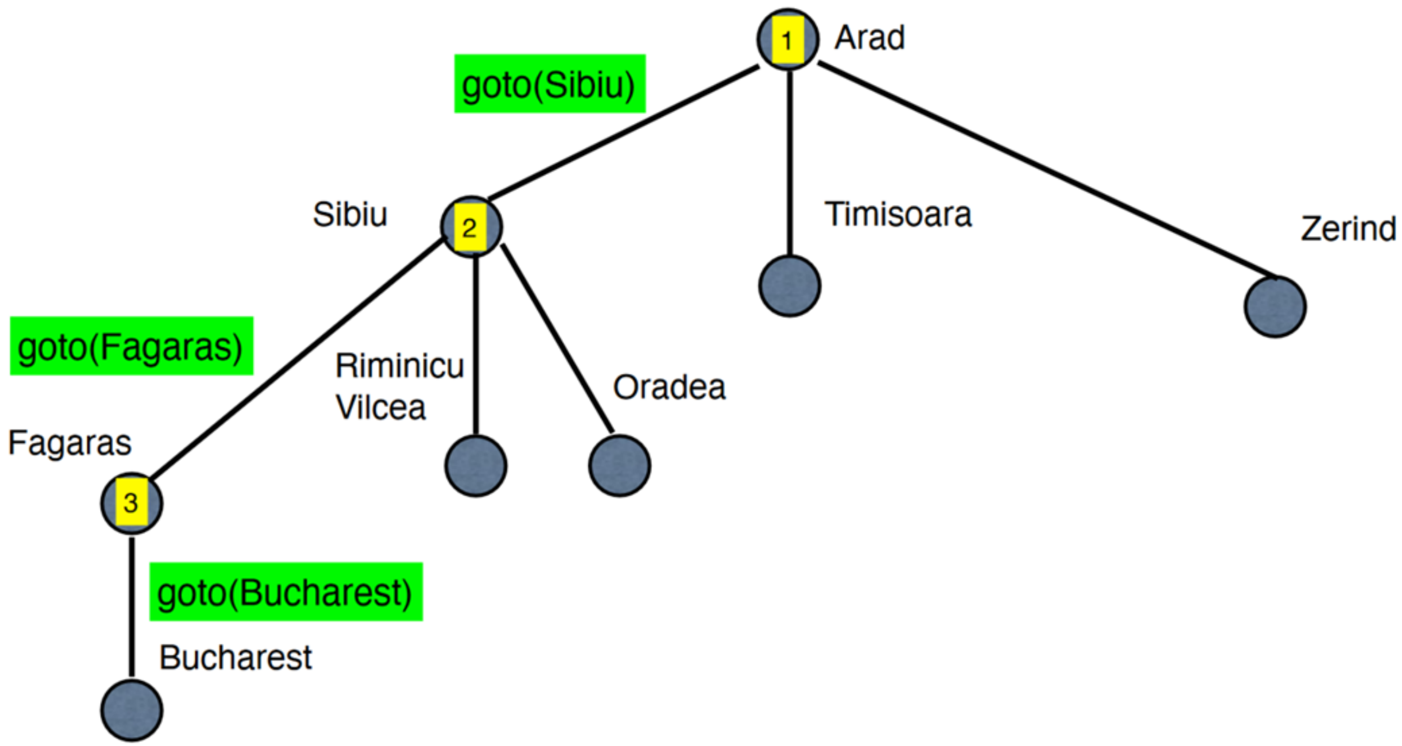


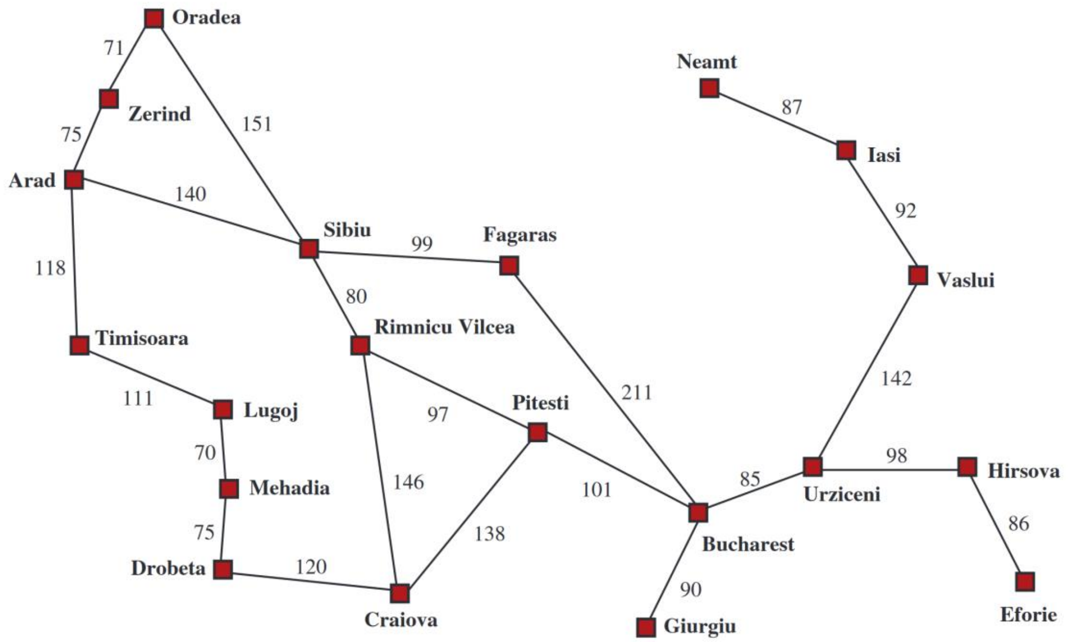
- Cost-optimal for problems where all actions have same cost
- Finds solution with minimal number of actions
- It is complete

Depth-First Search



Depth-First Search

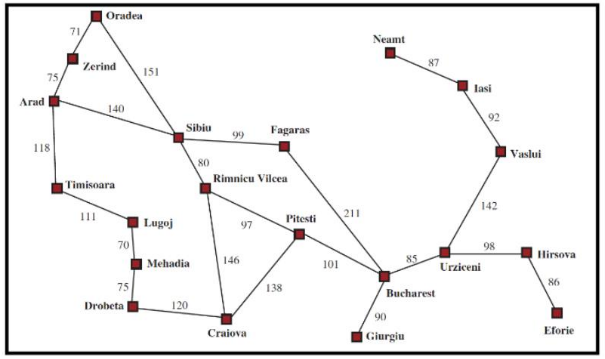




Heuristic Search

Straight line distance from city ***n*** to goal city ***n'***

Assume the cost to get somewhere
is a function of the distance traveled



Straight line distance to
Bucharest from any city

$h_{SLD()}$

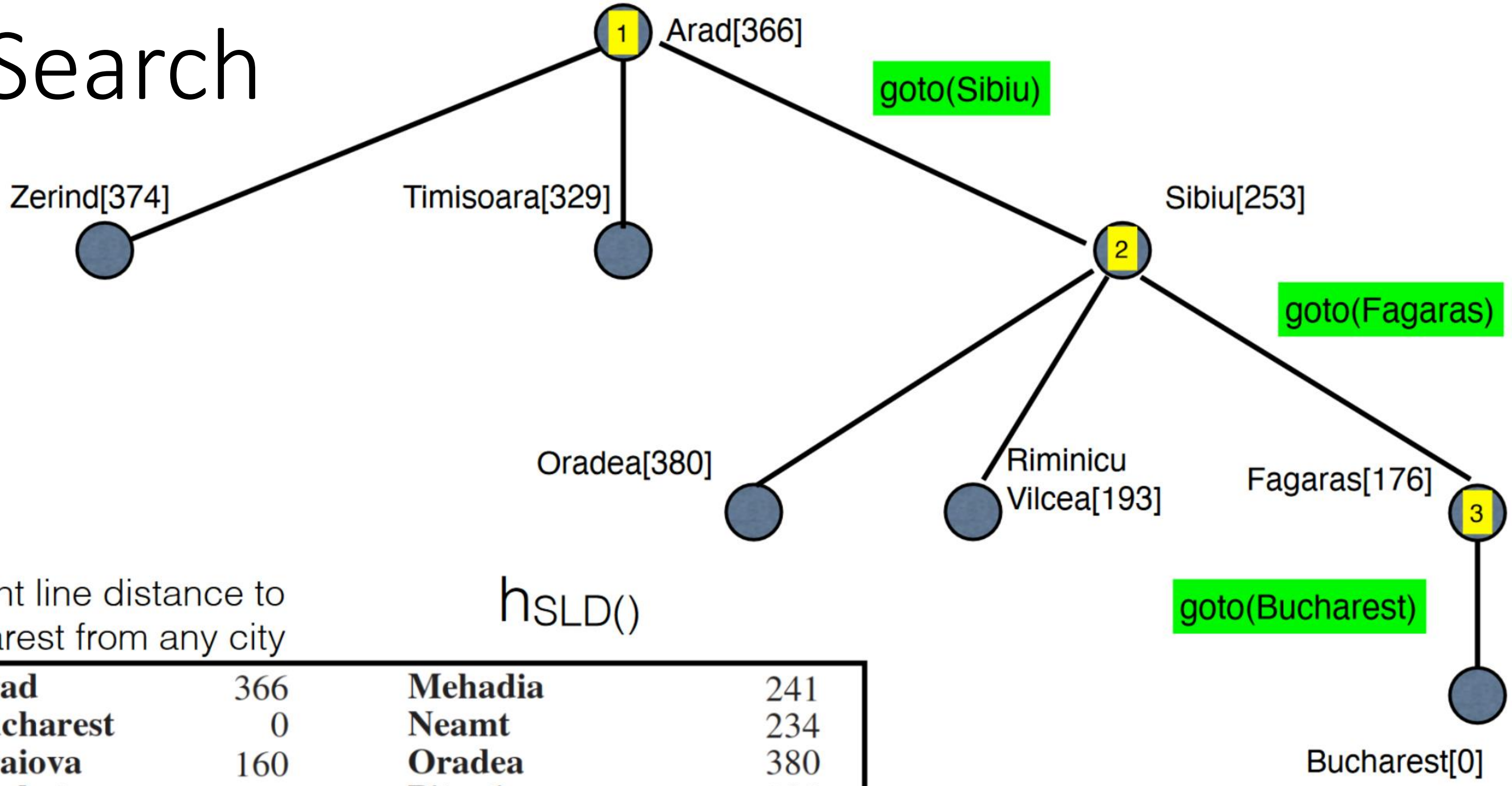
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Heuristic:

$$f(n) = h_{SLD}(n)$$

Notice the SLD under estimates the actual cost!

Heuristic Search



Straight line distance to Bucharest from any city

$h_{SLD}()$

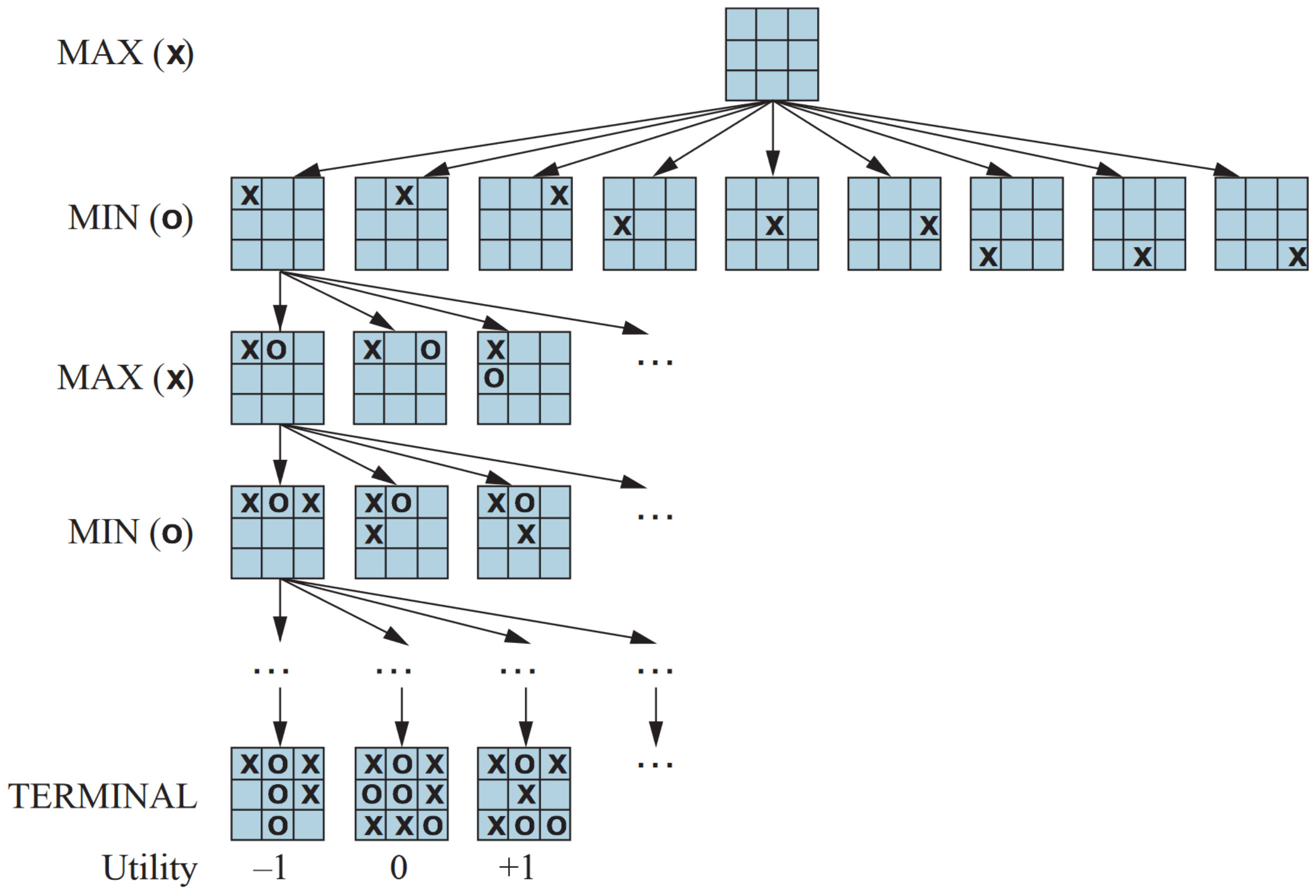
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Adversarial Search – Minimax

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```



Applications of Search

- Game playing (chess, Go, …)
- Constraint satisfaction
- Optimization
- Machine learning
- Planning
- …