SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Search

PROF. LIM KWAN HUI

50.021 Artificial Intelligence

Outline & Objectives

- Be able to formulate a problem in terms of state space, initial state, goal test, actions, transition model, path cost

- Understand the general characteristics behind search strategies

Recap: Environment Types

- Fully Observable vs Partially Observable? Agent is aware of complete state of environment
- Deterministic vs Stochastic? Next state of environment is based on agent’s action on current states
- Episodic vs Sequential? Choice of agent’s action in current “episode” is not based on previous “episodes”
- Static vs Dynamic? Environment does not change while agent is considering actions
- Discrete vs Continuous? A distinct number of percepts and actions
- Single Agent vs Multi Agent? Only a single agent acting in the same environment

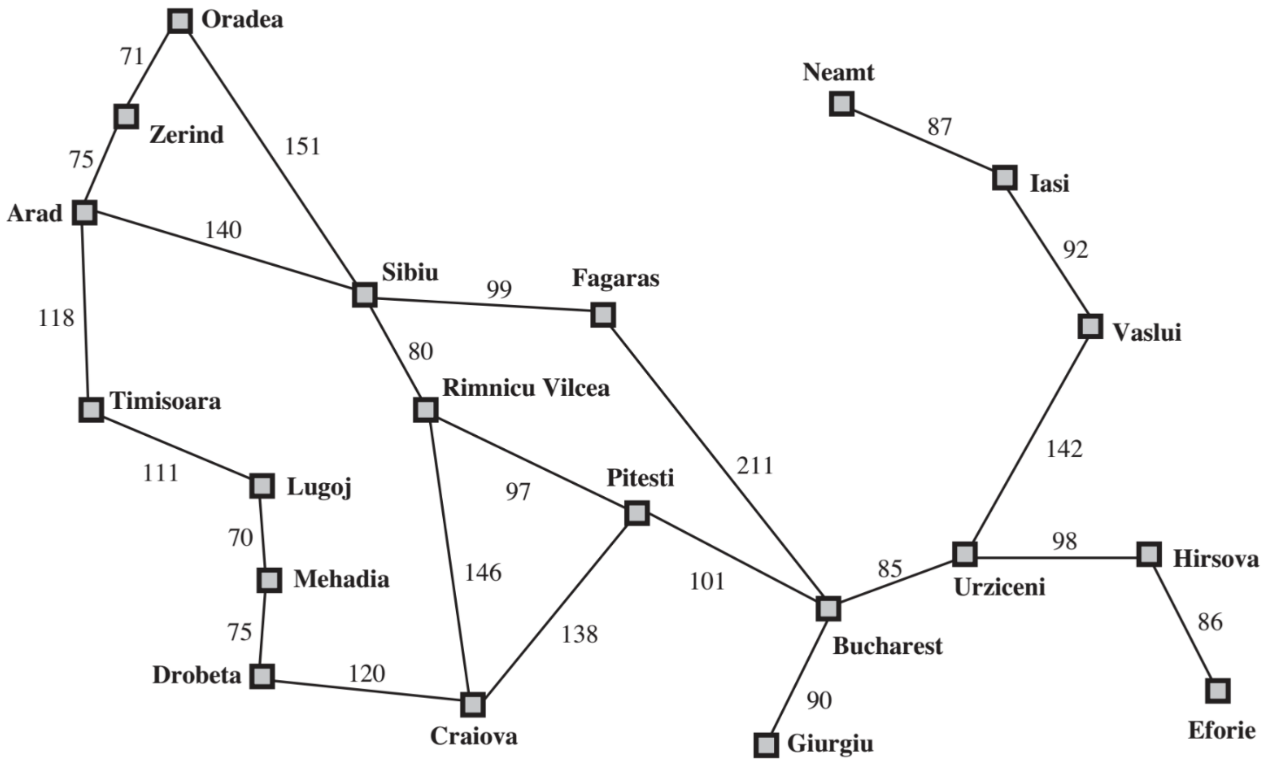
Recap: Environment Types

- For the next part on search, we will assume a simple environment, that is
  - Fully observable
  - Deterministic
  - Sequential
  - Static
  - Discrete
  - Single-agent

Problem Solving Agents

Example: Romania Holiday

- Task: Get to Bucharest from Arad



Problem-solving agent

- Formulate goal
  - What is/are the desired end state?
    - Goal:
- Formulate search problem
  - What actions and states to consider?
    - States:
    - Actions:
- Search for solutions
  - Determine the sequence of actions that lead to the goal
    - E.g.,

Problem-solving agent

- Formulate goal
  - What is/are the desired end state?
    - Goal: Reach Bucharest
- Formulate search problem
  - What actions and states to consider?
    - States: Individual cities
    - Actions: Move from city to city
- Search for solutions
  - Determine the sequence of actions that lead to the goal
    - E.g., Arad → Sibiu → Fagarus → Bucharest

Search Problem Formulation

- **State space**, e.g. *At(Arad), At(Bucharest)*
- **Initial state**, e.g. *At(Arad)*
- **Actions**, set of actions given a specific state
  - **Transition model** e.g., *Result(At(Arad),Go(Zerind))* → *At(Zerind)*
  - **Path cost** (additive), e.g., sum of distances, number of actions, etc
- **Goal test**, can be
  - Explicit, e.g. *At(Bucharest)*
  - Implicit, e.g. *checkmate(x)*

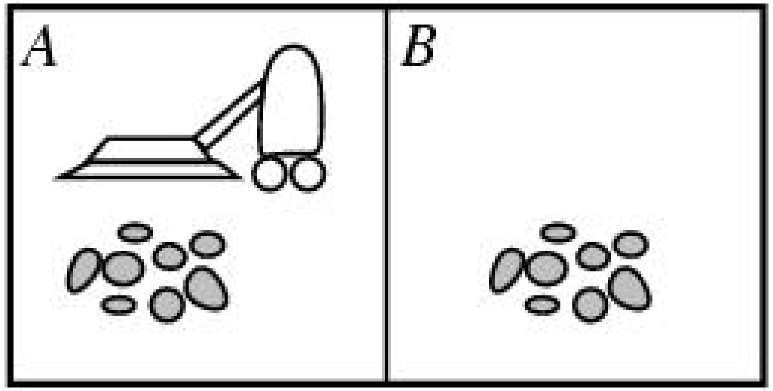
Search Problem Solution

- A solution is a sequence of actions from the initial state to a goal state
  - E.g., Arad → Sibiu → Fagarus → Bucharest
- An optimal solution is a solution with the lowest path cost

Problem Formulation

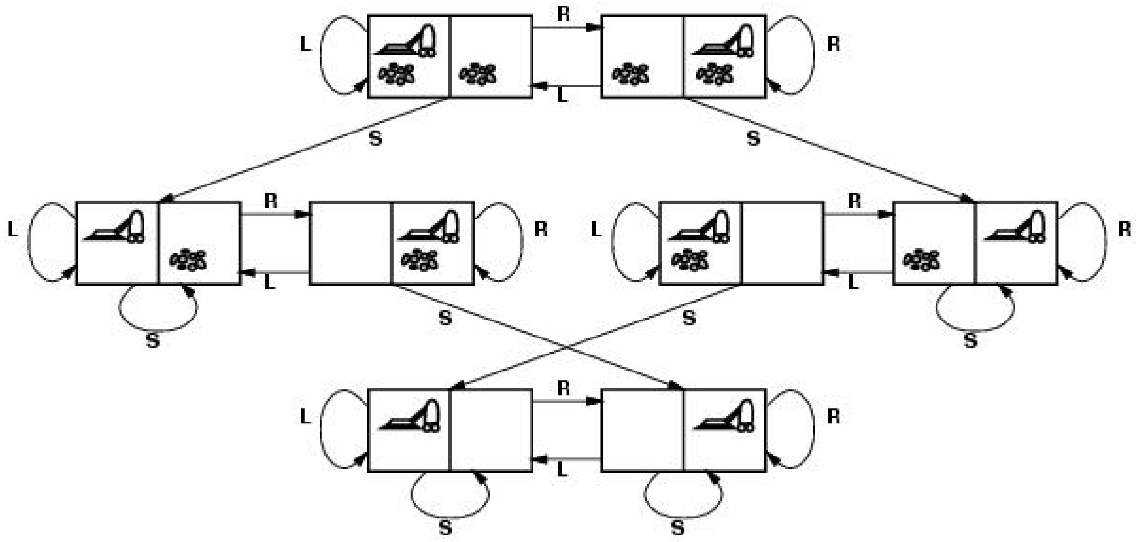
- Various things to consider:
  - Many different possible representations for states, actions, transition model, path cost
  - How do we choose this?
- Is this choice important?
  - Affects the combinatorial search space and how fast we can find a solution
    - States
    - Actions
    - Transition model
    - Path cost

Vacuum-cleaner world



- **State space:** ?
- **Initial state:** ?
- **Actions:** ?
- **Transition Model:** ?
- **Path cost:** ?
- **Goal test:** ?

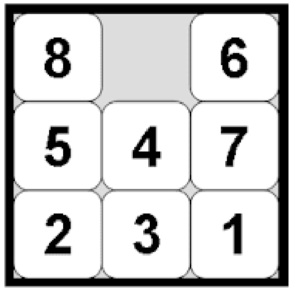
Vacuum-cleaner world

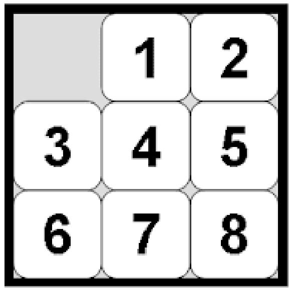


State space:	All possible combinations of cells/vacuum/dirt
Initial state:	Can be any of the above
Actions:	{left, right, suck}
Transition Model:	As above diagram
Path cost:	Number of actions to reach goal
Goal test:	All cells are clean

Exercise: 8-puzzle

Initial State

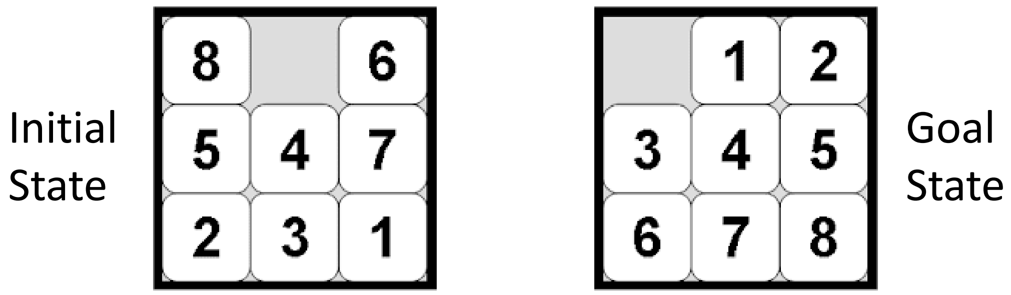




Goal State

- **State space:** ? `all possible combinations of tile in slot`
- **Initial state:** ? `as above`
- **Actions:** ? `move tile {left, right, up, down}`
- **Transition Model:** ? `tiles can move into empty spaces from adjacents slots`
- **Path cost:** ? `number of actions to reach goal`
- **Goal test:** ? `numbers are in order as shown`

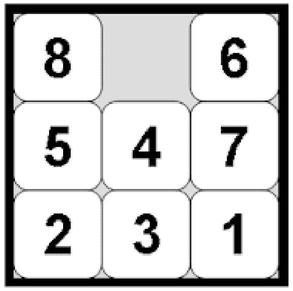
Exercise: 8-puzzle

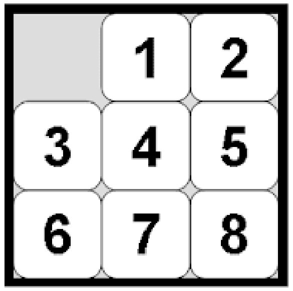


State space:	Number tiles in each cell position
Initial state:	[8,-,6,5,4,7,2,3,1]
Actions:	Move tile *{Left, Right, Up, Down}*
Transition Model:	Update tiles in current and target cell positions
Path cost:	Number of moves
Goal test:	Compare to positions in goal state

Exercise: 8-puzzle

Initial State





Goal State

Is this the only possible problem formulation?

○ **State space**:	Number tiles in each cell position
○ **Initial state:**	[8,-,6,5,4,7,2,3,1]
○ **Actions**:	Move tile *{Left, Right, Up, Down}*
○ **Transition Model**	Update tiles in current and target cell positions
○ **Path cost**:	Number of moves
○ **Goal test**:	Compare to positions in goal state

Selecting a state space

- Real world is absurdly complex.

  State space must be *abstracted* for problem solving.

- (Abstract) state = set of real states.

- (Abstract) action = complex combination of real actions.
  - e.g. Arad $\rightarrow$Zerind represents a complex set of possible routes, detours, rest stops, etc.
  - The abstraction is valid if the path between two states is reflected in the real world.

- (Abstract) solution = set of real paths that are solutions in the real world.

- Each abstract action should be "easier" than the real problem.

General Search

Basic search algorithms

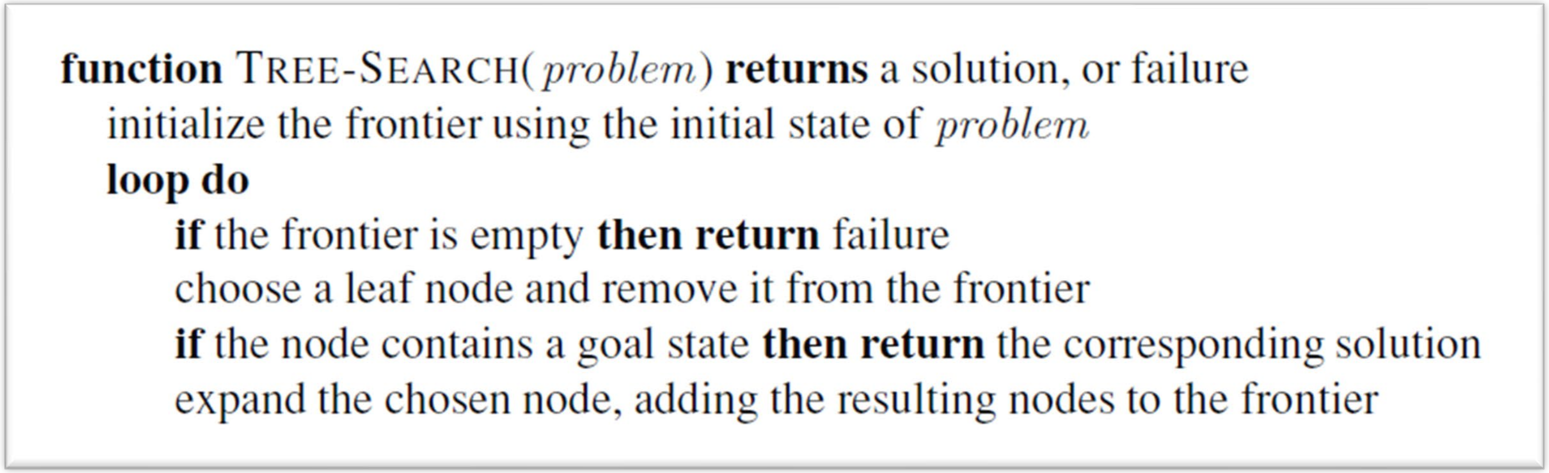
- How do we find solutions to search problems?
  - Search the state space
    - Complexity of space depends on state representation

  - How exactly? Search via an *explicit tree generation*
    - Root = initial state
    - Nodes and leaves
      - Generated through transition model (successor function)

  - Tree search treats different paths to the same state as distinct

General Tree Search

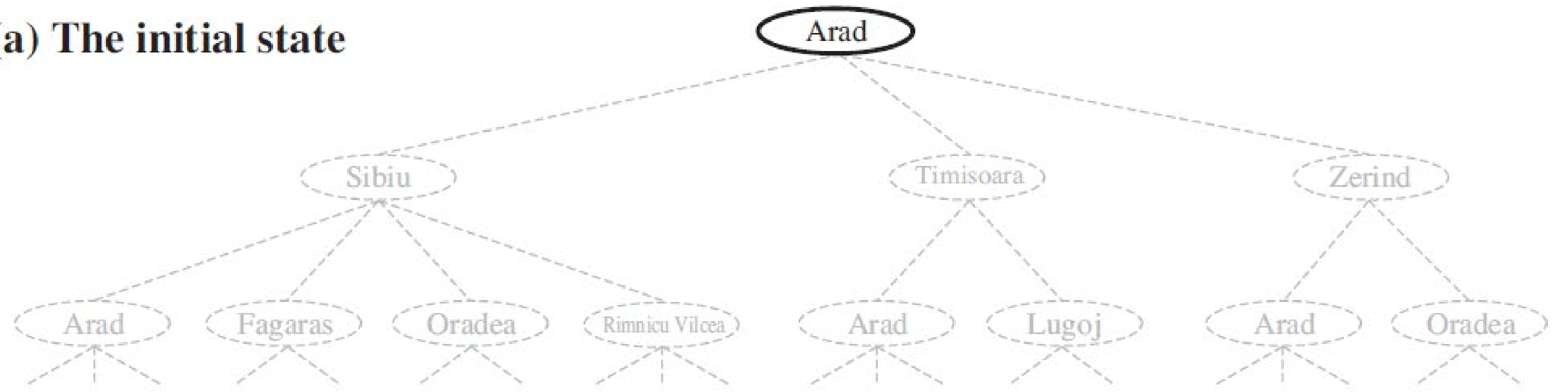
- Basic Idea
  - Offline, simulated exploration of state space
  - Expanding states by generating successors of already-explored states



```
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
```

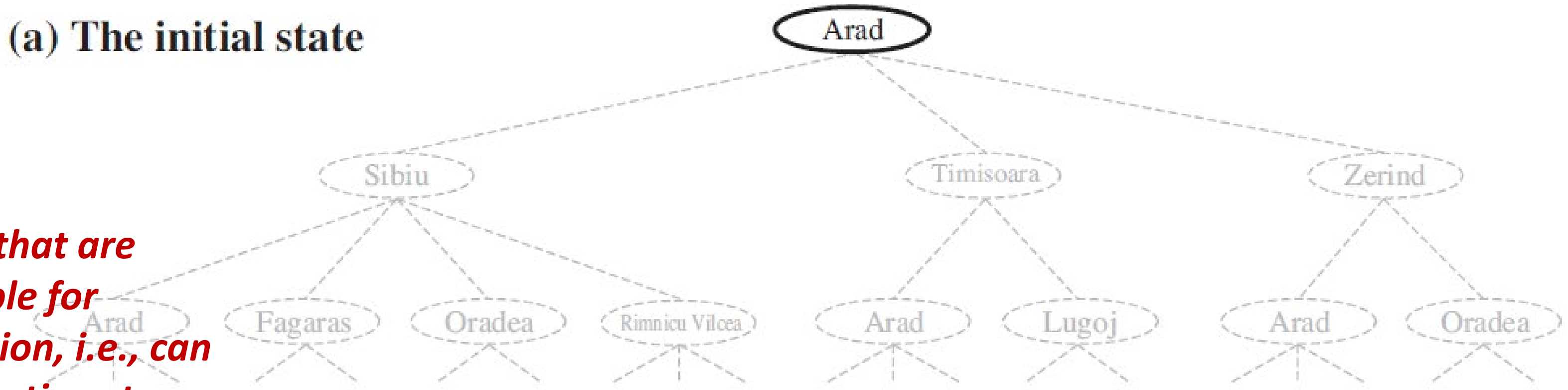
General Tree Search

(a) The initial state



```
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
```

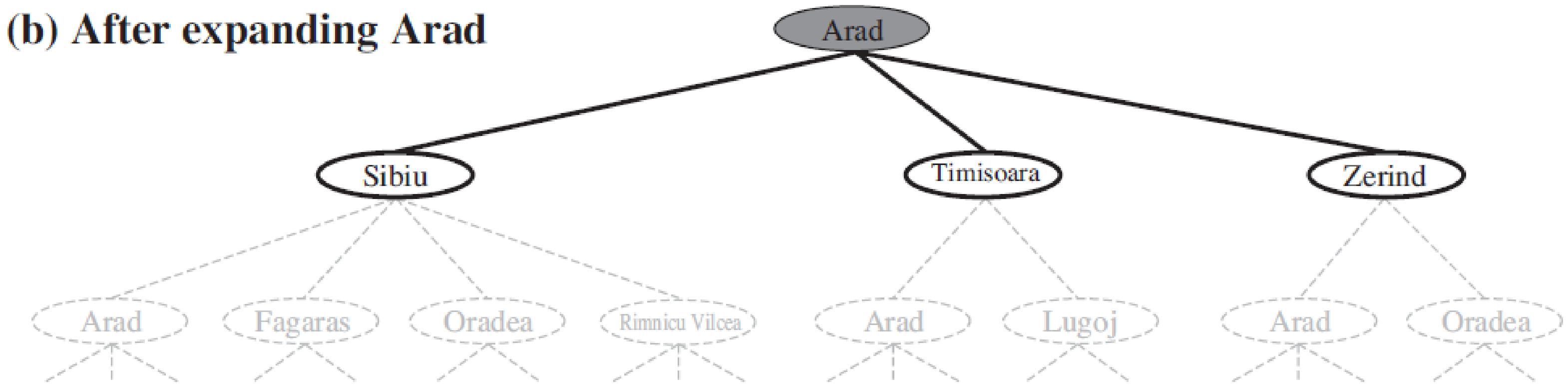
General Tree Search



States that are available for expansion, i.e., can apply actions to

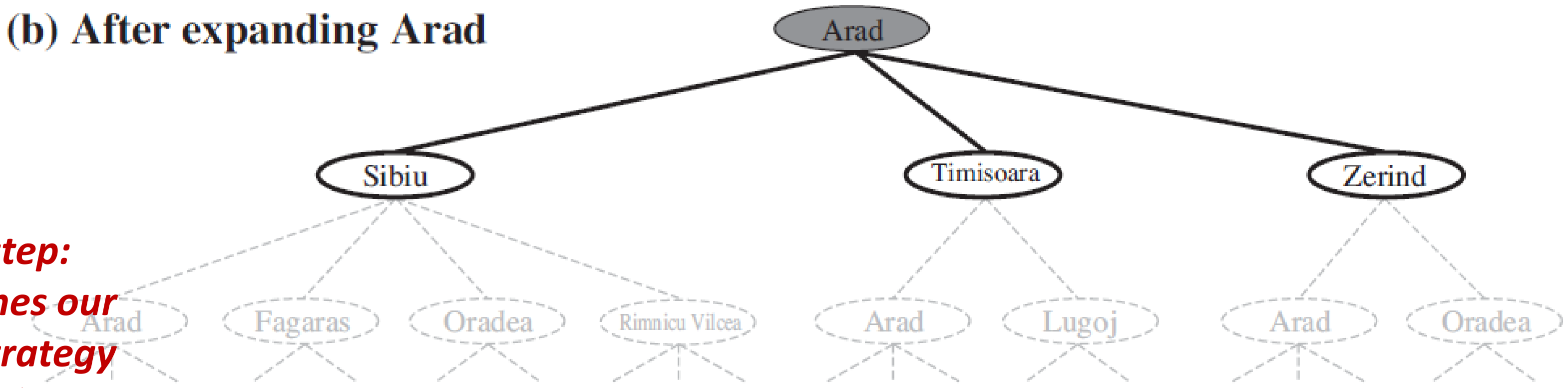
```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     expand the chosen node, adding the resulting nodes to the frontier
```

General Tree Search



```
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 Tree Search

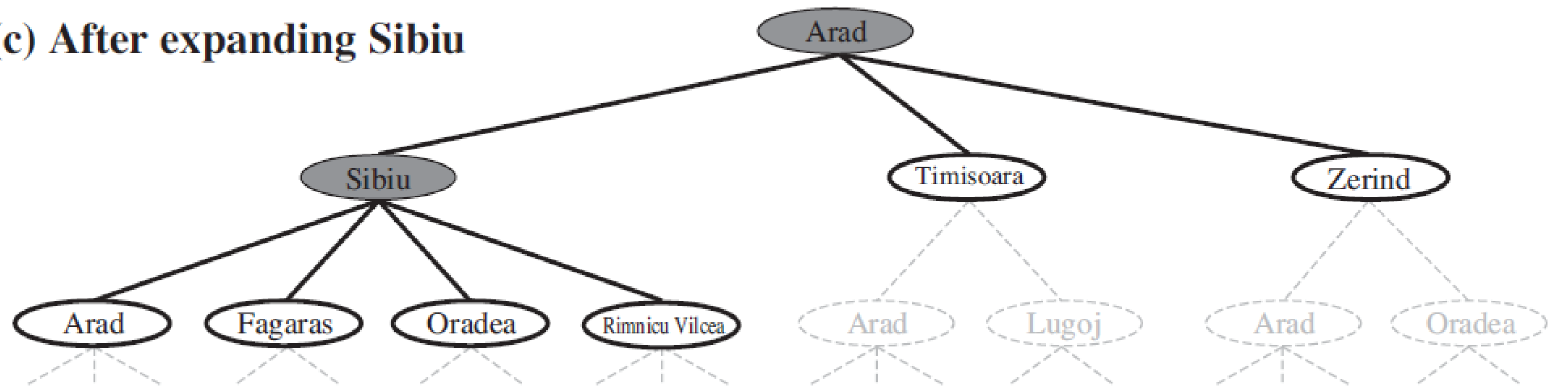


Expand step: Determines our search strategy

```
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 Tree Search

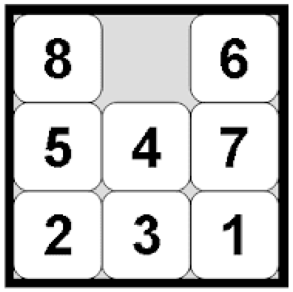
(c) After expanding Sibiu

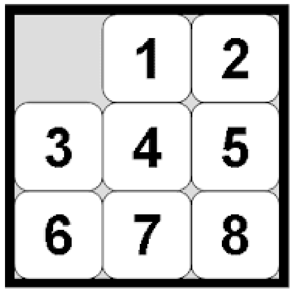


```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      expand the chosen node, adding the resulting nodes to the frontier
```

States and Nodes

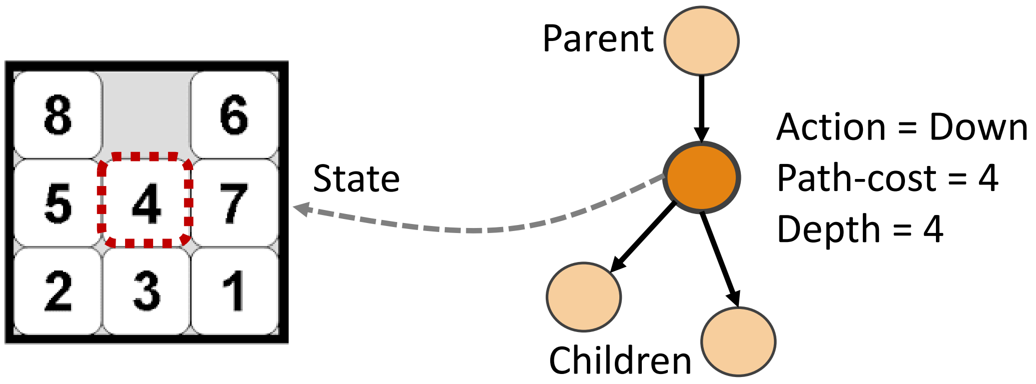
- A *state* is a (representation of a) *physical configuration*





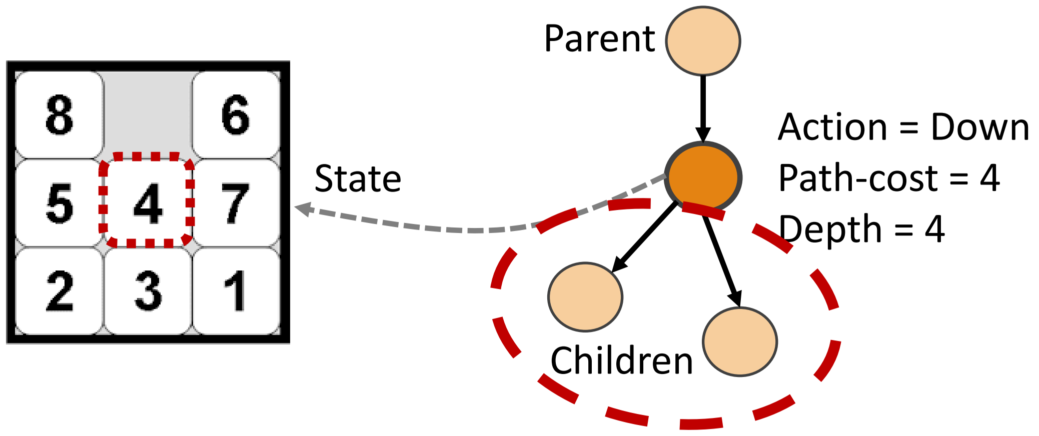
States and Nodes

- A *state* is a (representation of a) *physical configuration*
- A *node* is a data structure constituting *part of a search tree*
  - Comprising *state, parent-node, child-node(s), action, path-cost, depth*
  - In contrast*, states* do not have parents, children, depth or path cost



States and Nodes

- A *state* is a (representation of a) *physical configuration*
- A *node* is a data structure constituting *part of a search tree*
  - Comprising *state, parent-node, child-node(s), action, path-cost, depth*
  - In contrast, states do not have parents, children, depth or path cost

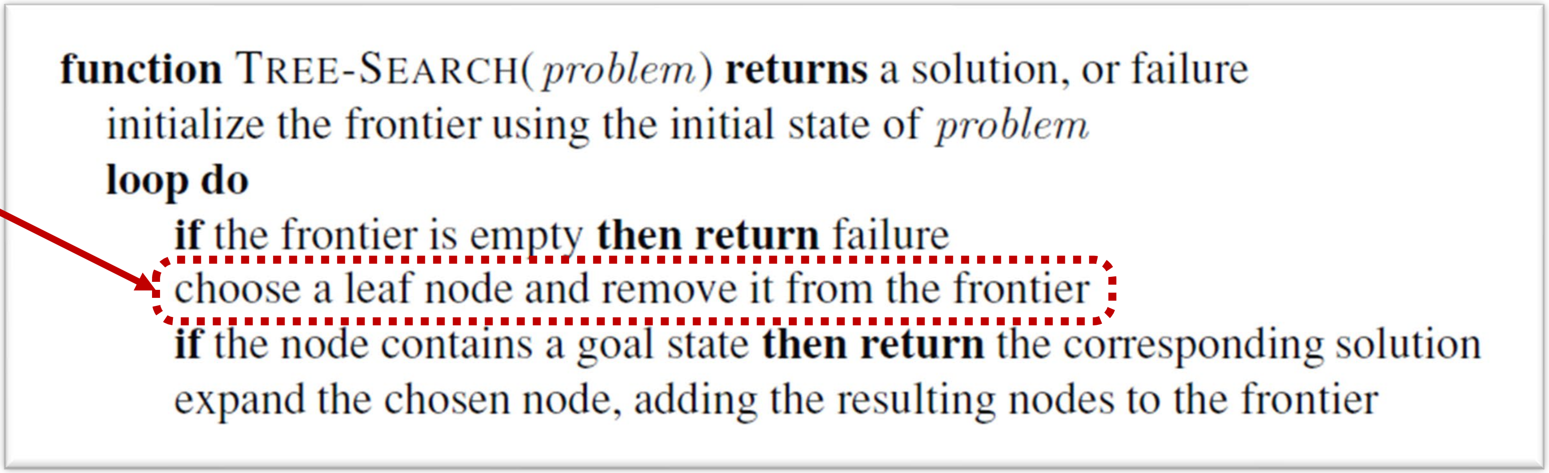


- The Expand function creates new nodes (and their various fields)
  - using the Actions and Transition Model to create the corresponding states

Search Strategies

- The Expand function creates new nodes (and their various fields)
  - using the Actions and Transition Model to create the corresponding states
- A **search strategy** is defined by picking the *order of node expansion*

Expand step: Determines our search strategy



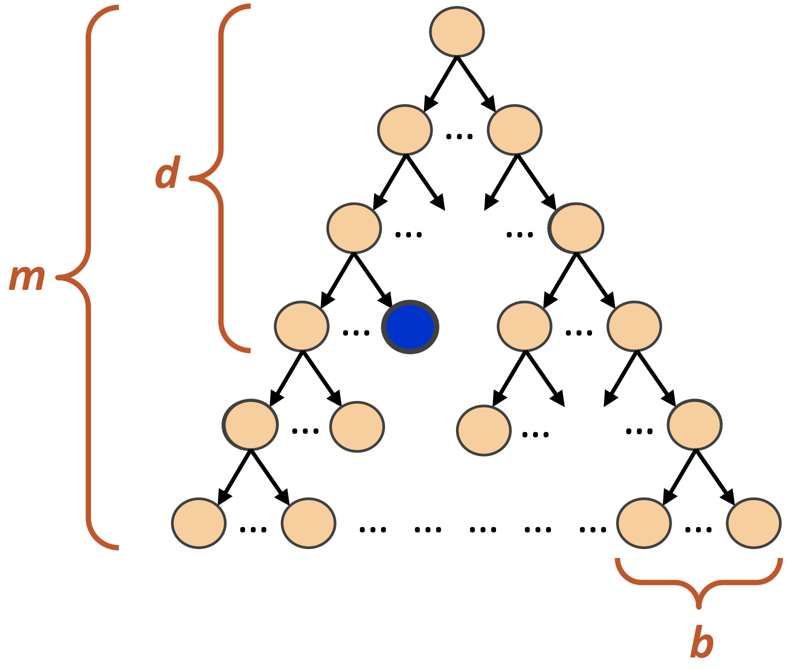
```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      expand the chosen node, adding the resulting nodes to the frontier
```

Search Strategies

- Strategies are evaluated 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*** - number of nodes generated/expanded
  - ***Space complexity*** - maximum number of nodes in memory

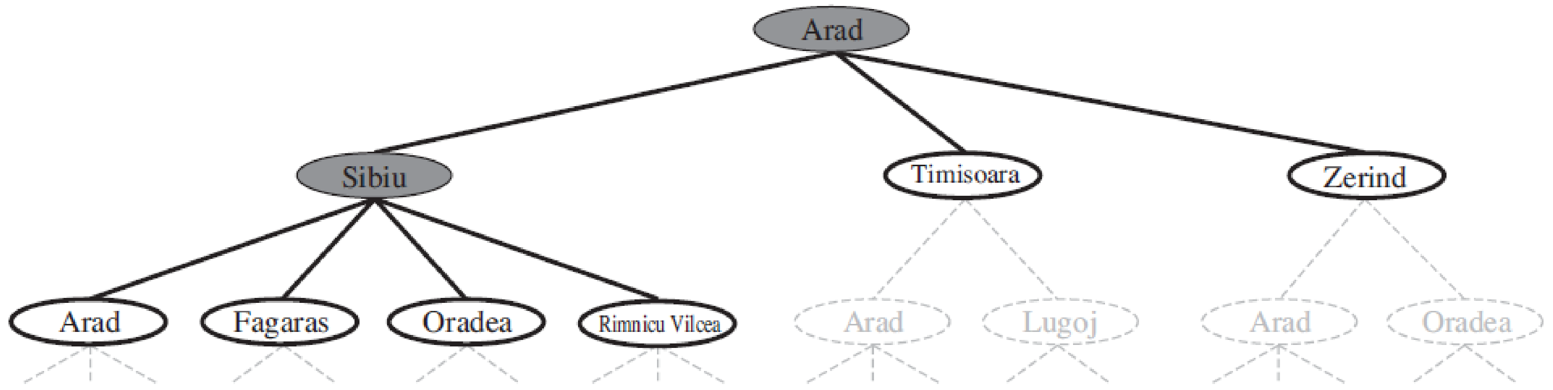
Search Strategies

- Time and space complexity are measured in terms of
  - ***b*** - maximum branching factor of the search tree
  - ***d*** - depth of the least-cost solution
  - ***m*** - maximum depth of the state space (may be infinite)



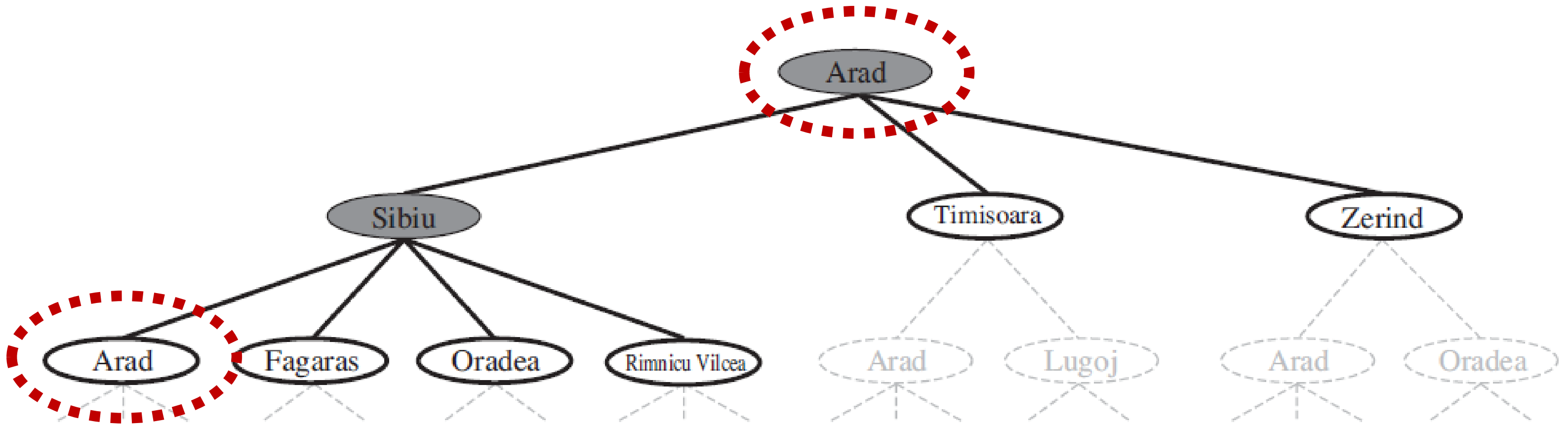
● Least-cost solution

Tree Search Problem



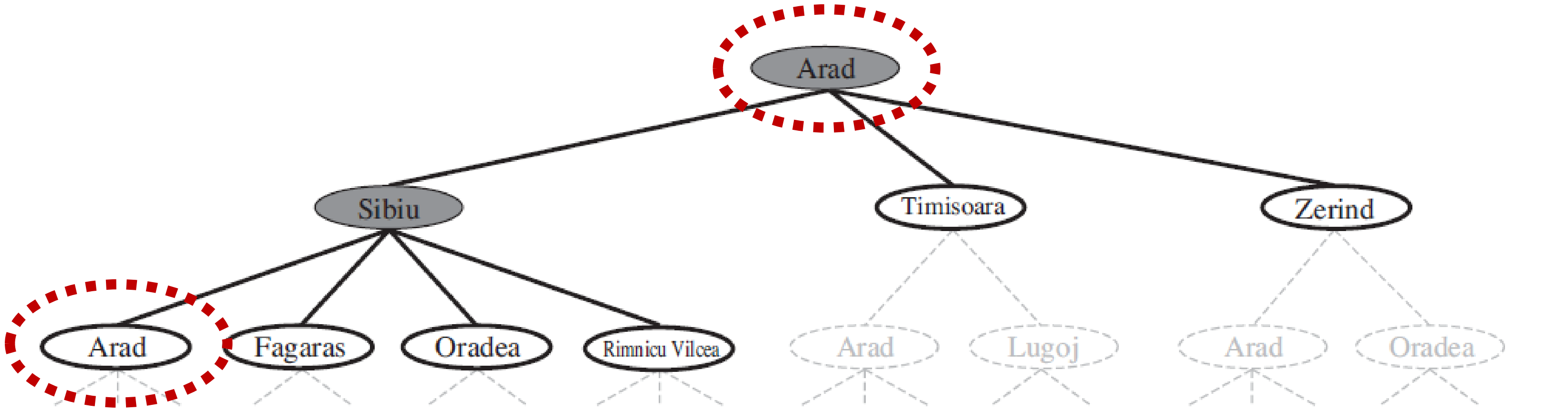
- What is the problem?

Tree Search Problem



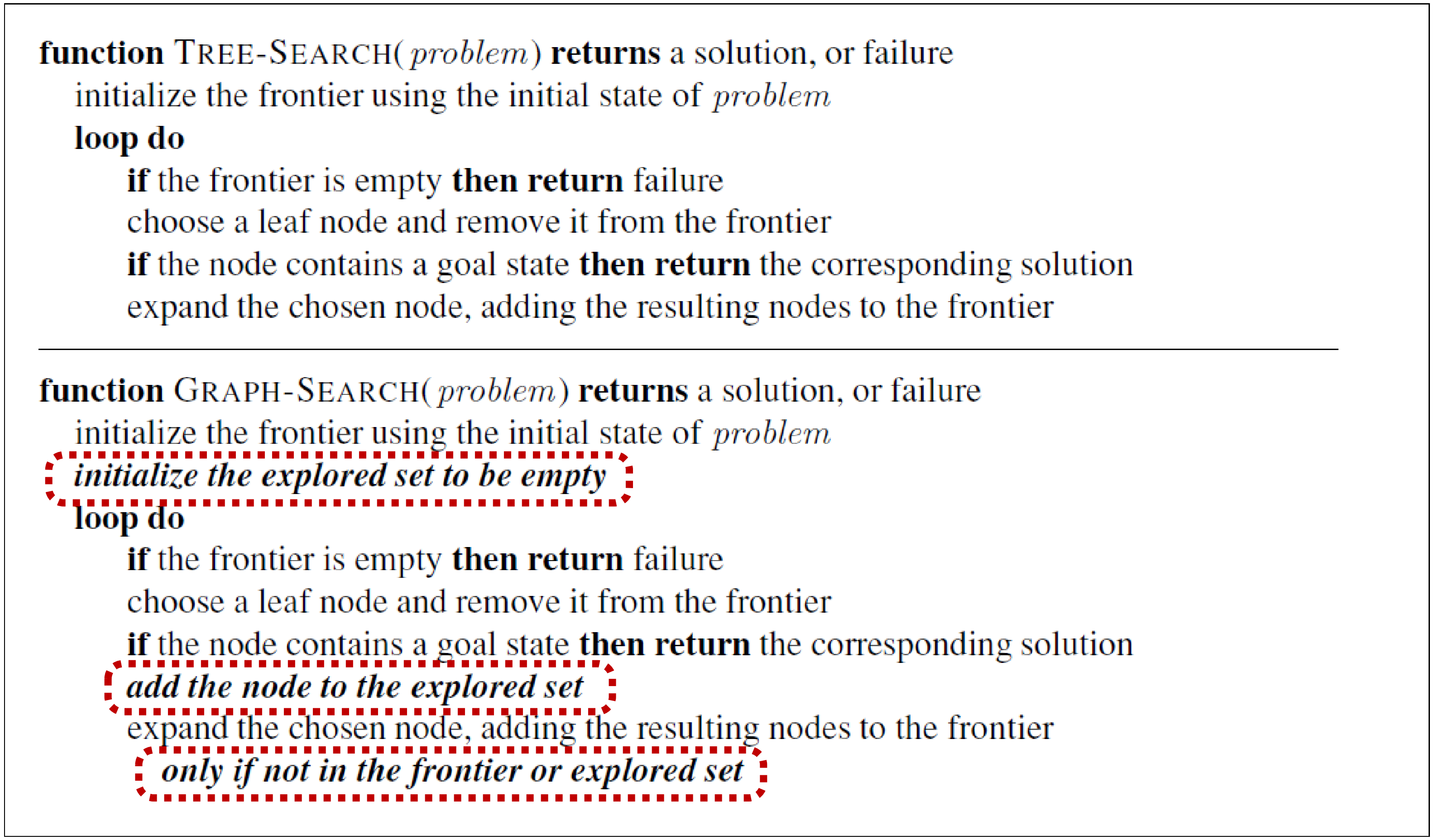
- What is the problem? Repeated states
  - Redundant paths can cause a tractable problem to become intractable
- Solution?

Tree Search Problem



- What is the problem? Repeated states
  - Redundant paths can cause a tractable problem to become intractable
- Solution? Graph search
  - Modify tree search to keep track of previously visited states (to not re-visit)
  - But potentially large number of states to track

Graph Search vs Tree Search



```
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
```

```
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
```

Types of Search

- Uninformed Search
  - No additional information about states beyond that in the problem definition (AKA blind search)
- Informed Search
  - Uses problem-specific knowledge beyond the definition of the problem itself
- Adversarial Search
  - Used in multi-agent environment where the agent needs to consider the actions of other agents and how they affect its own performance.