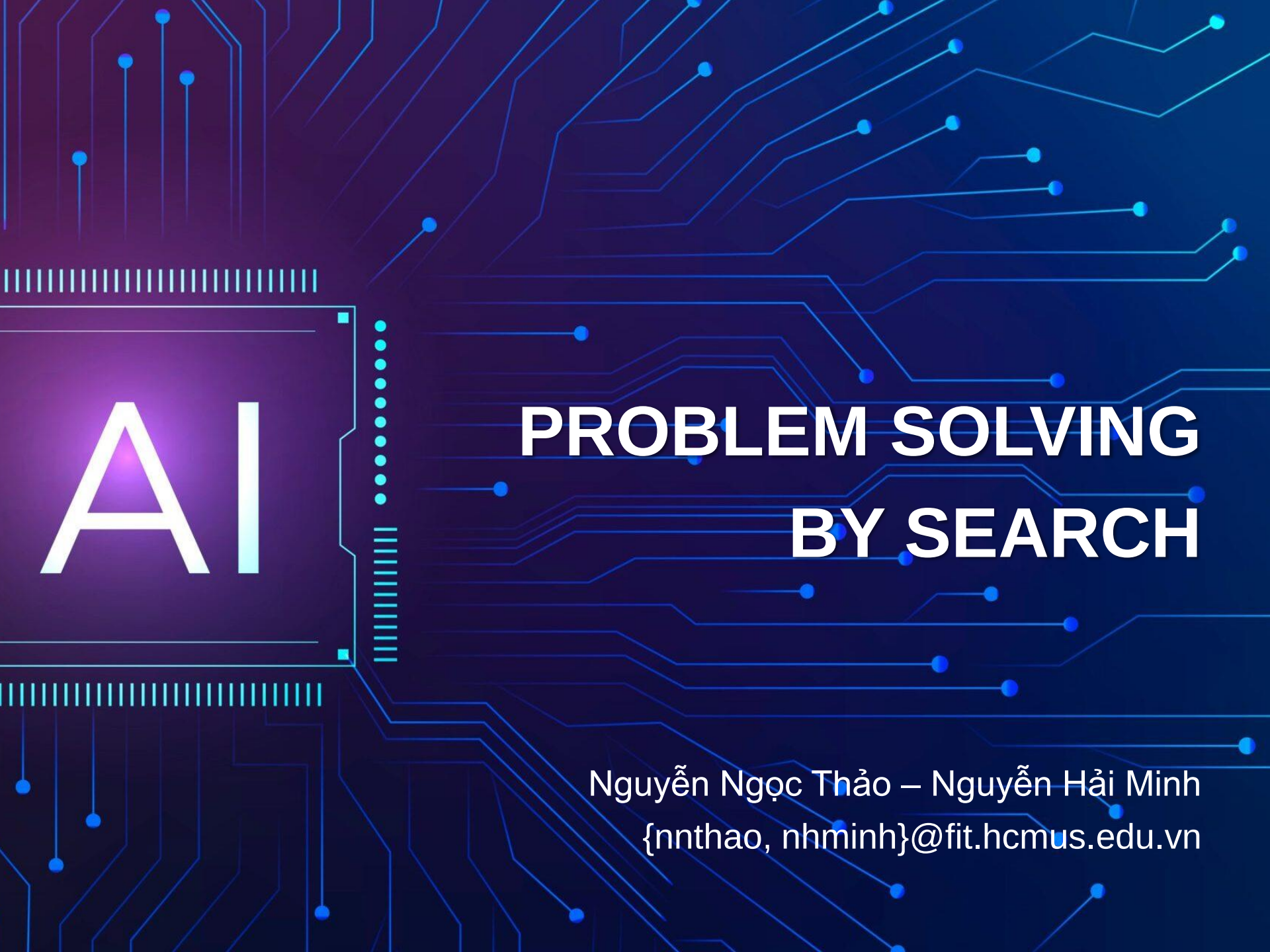


PROBLEM SOLVING BY SEARCH

Nguyễn Ngọc Thảo – Nguyễn Hải Minh
{nnthao, nhminh}@fit.hcmus.edu.vn

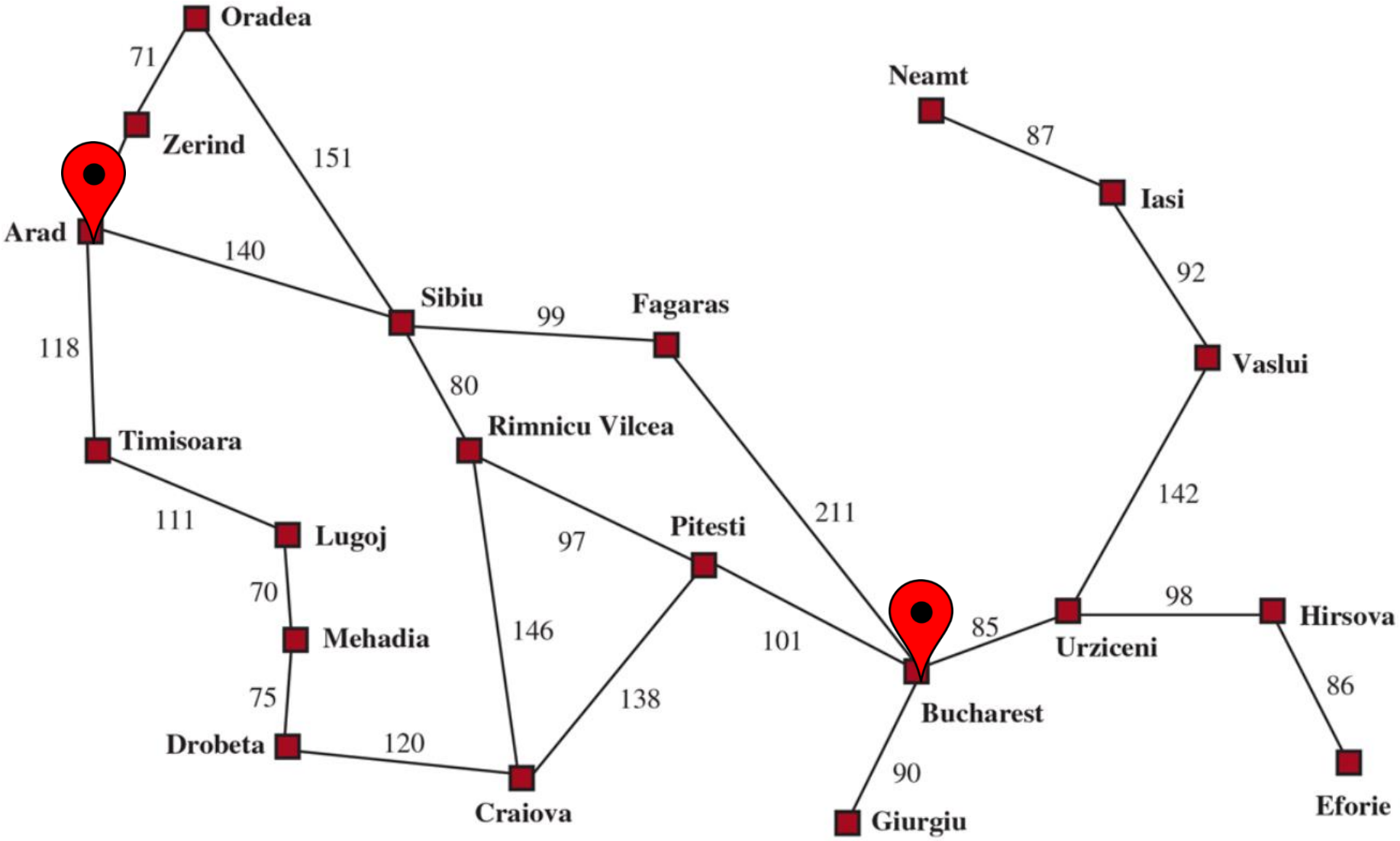
Outline

- Problem-solving agents
- Example problems
- Searching for solutions

Problem-solving agents

- *Well-defined problems and solutions*
- *Formulating problems*

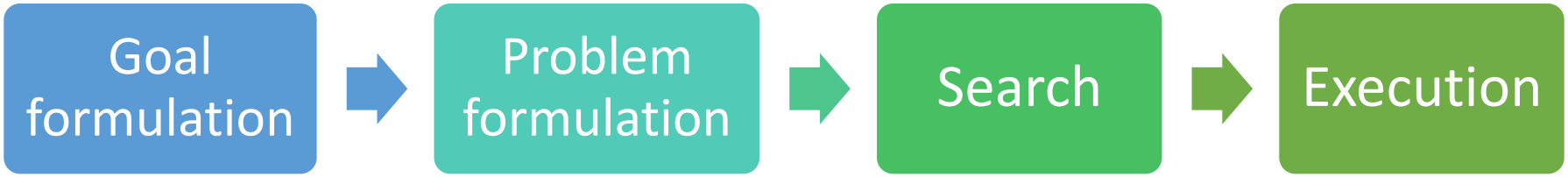
A touring holiday in Romania



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

Problem-solving process

- Assume the agent always has access to information about the world by using a map.
- Then, it follows the four-phase problem-solving process



Problem-solving process

- Goal formulation: Goals organize behavior by limiting the objectives and hence the actions to be considered.
  - E.g., in the touring holiday example, the goal is reaching Bucharest.

- Problem formulation: The agent devises a description of the states and actions necessary to reach the goal.
  - E.g., in the above example, one good model is to consider the actions of traveling from one city to an adjacent city.

Problem-solving process

- Search: The agent simulates sequences of actions in its model, searching until it finds a **solution**.
  - This is done before taking any action in the real world.
  - A solution is a sequence of actions that reaches the goal, e.g., going from Arad to Sibiu to Faragas to Bucharest.
- Execution: The agent can now execute the actions in the solution, one at a time.

Search problems and solutions

- A search problem can be defined formally as follows.
- A set of possible states that the environment can be in, called the state space
  - E.g., the cities, {Arad, Zerind,..}, in the touring holiday example
- The initial state that the agent starts in
  - E.g., in the above example, the agent begins at Arad.
- A set of one or more goal states
  - E.g., in the above example, the agent wants to arrive at Bucharest.
  - There can be one or several goal states, or the goal is defined by a property that applies to many states (potentially an infinite number).

Search problems and solutions

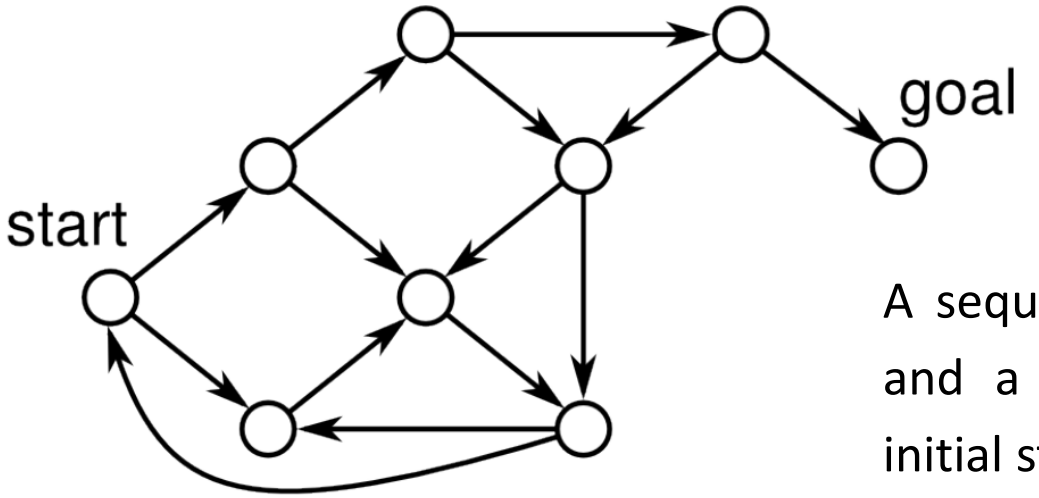
- A finite set of actions that are executable in state s, $Action(s)$
  - E.g., $Action(Arad) = \{ToSibiu, ToTimisoara, ToZerind\}$
- A transition model, $Result(s, a)$, returns the state that results from doing action a in state s.
  - E.g., $Result(Arad, ToZerind) = Zerind$

Search problems and solutions

- An action cost function, $Action\text{-}Cost(s, a, s')$, gives the numeric cost of applying action a in state s to reach state s'.
  - The cost function should reflect the agent's performance measure.
  - E.g., for route-finding agents, the cost might be the length in miles or the time to complete the action.
- Action costs are assumed to be positive and additive.
- The total cost of a path is the sum of individual action costs.
- An optimal solution has the lowest cost among all solutions.

The state space

- The state space can be represented as a graph.
- The vertices are states and the directed edges between them are actions.



A sequence of actions forms a **path**, and a **solution** is a path from the initial state to a goal state.

Formulating problems

- The formulation of the problem is a model—an abstract mathematical description—and not the real thing.

How do you solve a navigation problem?

VS.

Compare the simple atomic state description to an actual cross-country trip, where the state of the world includes so many things: the traveling companions, the condition of the road, the weather, the traffic, and so on.

The model deals with the change in location only while leaving out all the other considerations.

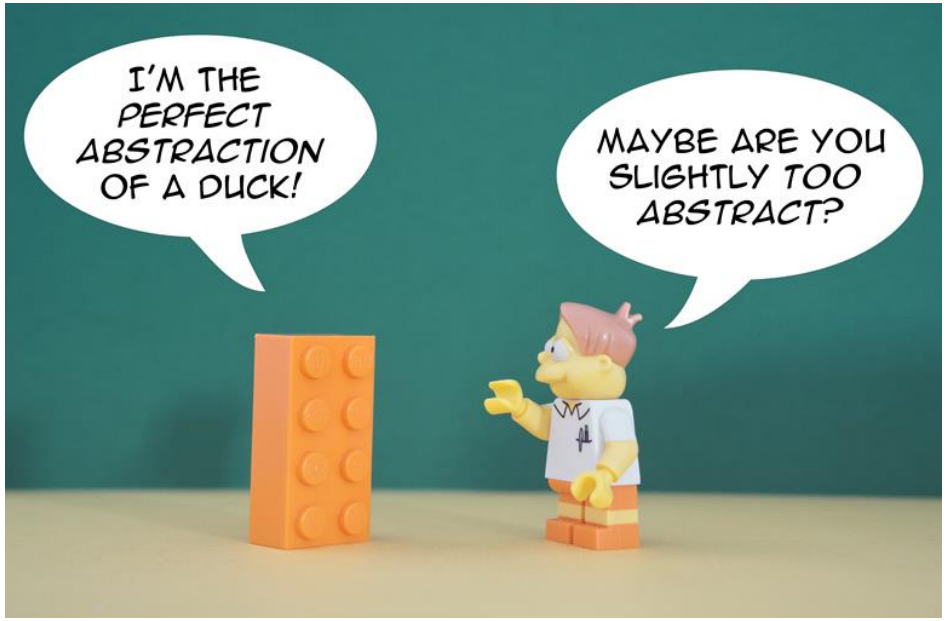
Formulating problems

- Abstraction involves removing detail from a representation.
- It creates an approximate and simplified model of the world, which is critical for automated problem solving.

- A valid abstraction lets any abstract solution be turned into a solution in the more detailed world.
- An abstraction is useful if carrying out each of the actions in the solution is easier than the original problem.

Formulating problems

- A good abstraction eliminates as much detail as possible.
- Meanwhile, it retains validity and ensures that the abstract actions are easy to carry out.



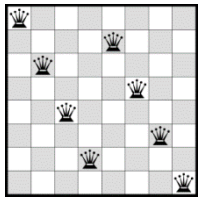
Example problems

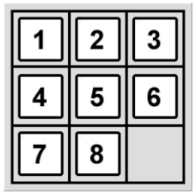
- *Standardized problems*
- *Real-world problems*

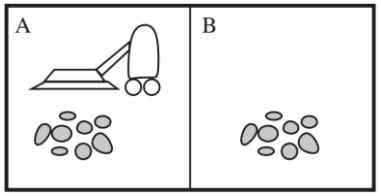
Example problems

Standardized problems

- Illustrate or exercise various problem-solving methods.
- Descriptions are concise and exact.
- Suitable as a benchmark for comparing the performance of algorithms.



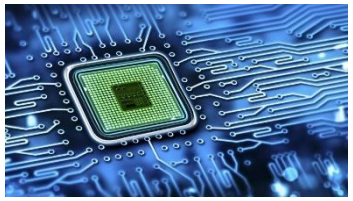





```
  T W O
+ T W O
-------
F O U R
```

Real-world problems

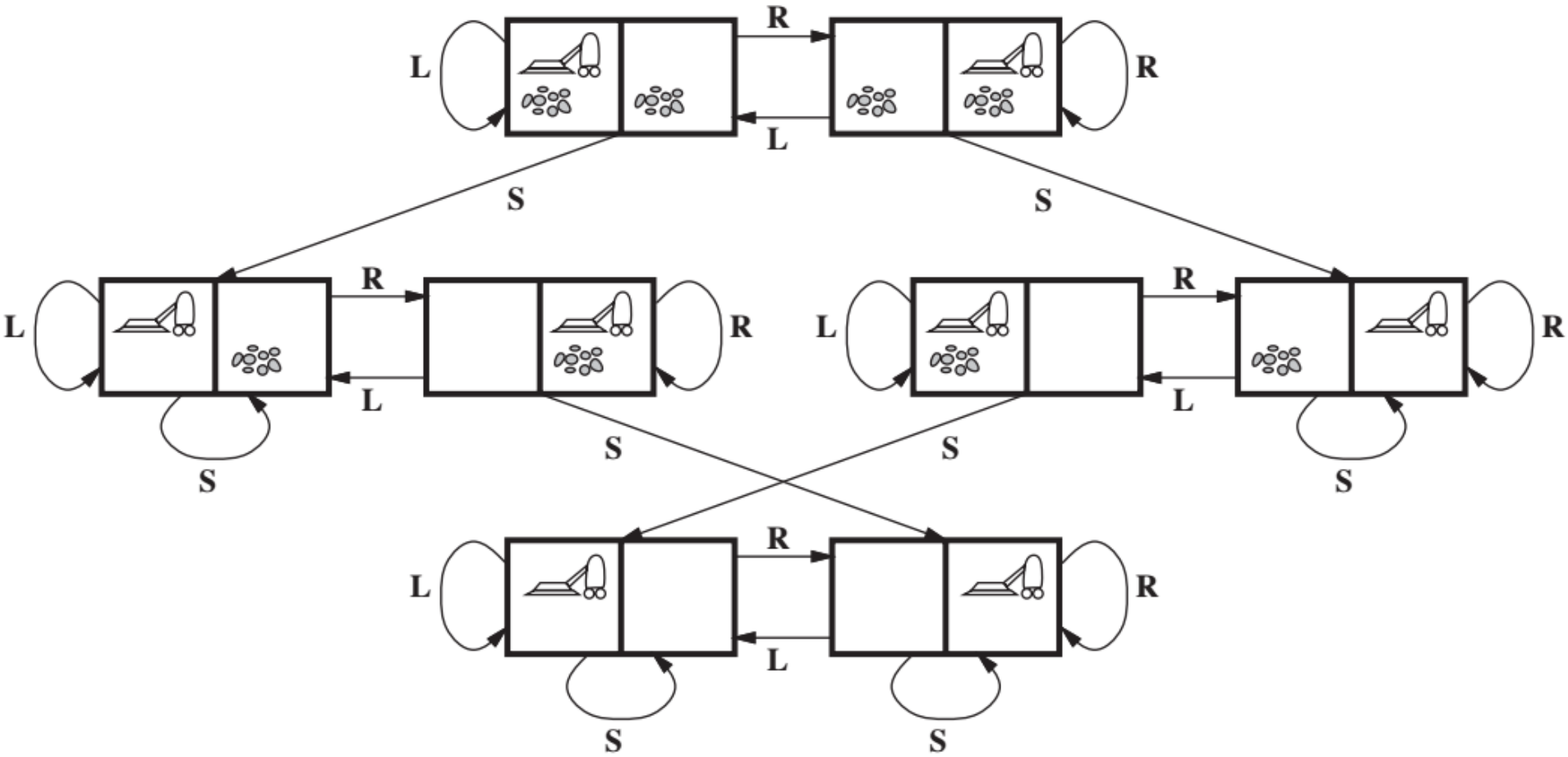
- Bring solutions to practical issues
- Formulations are idiosyncratic and not standardized.



The Vacuum-cleaner world

- States: determined by the agent location and the availability of dirt
  - $2 \times 2^2 = 8$ possible world states ($n \times 2^n$ for environment with n cells)
- Initial state: Any state can be designated as the initial state.
- Actions: Left, Right, and Suck
  - Larger models may include Upward and Downward.
- Transition model: The actions have their expected effects.
  - Note that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.
- Goal states: The states in which every cell is clean.
- Action cost: Each step costs 1.

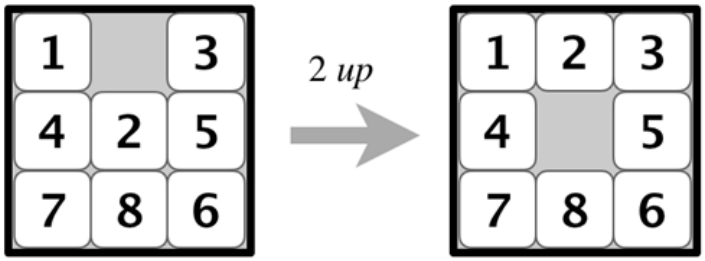
The Vacuum-cleaner world



The state space for the vacuum world
Links denote actions: L = Left, R = Right, S = Suck.

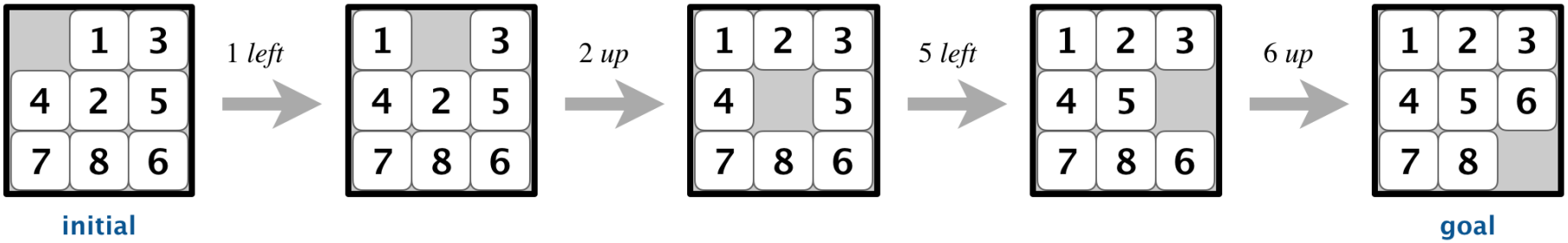
The 8-puzzle problem

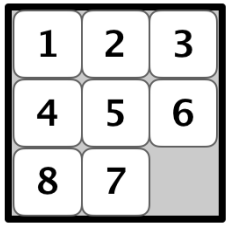
- States: specified by the location of each of the eight tiles.
- Initial state: Any state can be designated as the initial state.
- Actions: The movement of the blank space—Left, Right, Up, Down
- Transition model: Map a state and action to a resulting state.



- Goal states: Any state could be the goal
- Action cost: Each step costs 1.

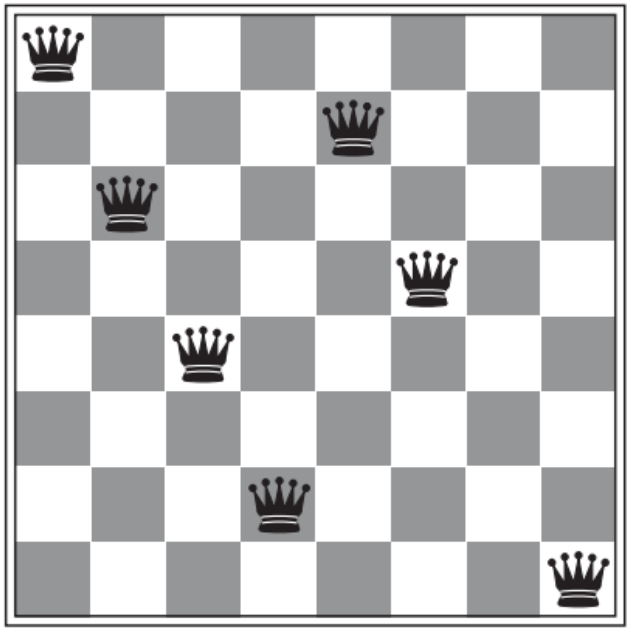
The 8-puzzle problem





- A member of the family of sliding-block puzzles, NP-complete
- 8-puzzle: 9! / 2 = 181,440 reachable states → easily solved.
- 15-puzzle: 16! / 2—over 10 trillion states → optimally solved in millisecs
- 24-puzzle: around 10^{25} states → optimally solved in several hours
- Note the parity property that partitions the state space—any given goal can be reached from exactly half of the possible initial states.

The 8-queens problem



- Incremental formulation: add a queen step-by-step to the empty initial state
- Complete-state formulation: start with all 8 queens on the board and move them around
- The action/path cost is trivial because only the final state counts.

The 8-queens: Incremental formulation

- States: Any arrangement of 0 to 8 queens on the board.
- Initial state: No queens on the board
- Actions: Add a queen to any empty square.
- Transition model: Return the board with a queen newly added to the specified square
- Goal states: 8 queens are on the board, none attacked.
- $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate

The 8-queens: Incremental formulation

- A better formulation would prohibit placing a queen in any square that is already attacked.
- States: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, no queen attacking another.
- Actions: Add a queen to any square in the leftmost empty column such that it is not attacked by other queens

- 8-queens: from 1.8×10^{14} states to just 2,057 states
- 100-queens: from 10^{400} states to about 10^{52} states.
- A big improvement, yet insufficient to make the problem tractable,

Donald Knuth's 4 problem (1964)

- Starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer.

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}} \right\rfloor = 5$$

620,448,401,733,239,439,360,000 steps

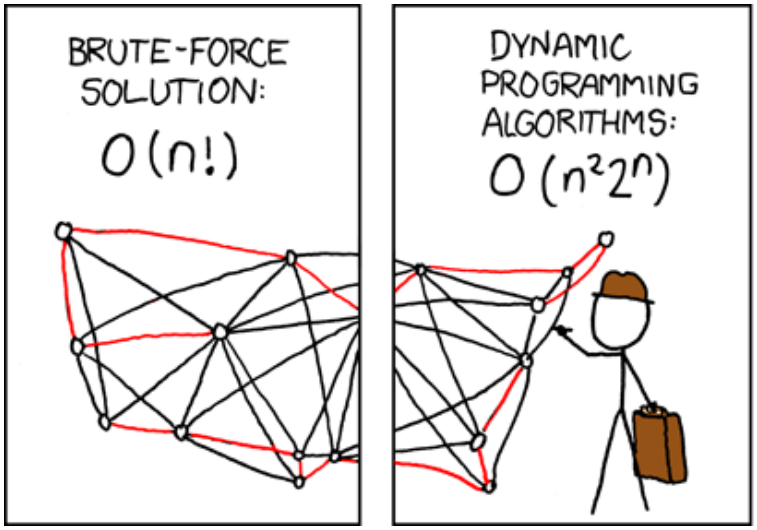
- States: positive numbers $\rightarrow$ **infinite state space**
- Initial state: 4
- Actions: Apply factorial (for integers only), square root, and floor operation
- Transition model: given by the operations' mathematical definitions
- Goal states: The desired positive integer.
- Action cost: Each action costs 1.
- Infinite state spaces arise in tasks like mathematical expressions generation, circuits, proofs, programs, etc.

The route-finding problem

- Consider the airline travel problems solved by a travel-planning Web site.
- States: Each state includes a location (e.g., an airport) and the current time.
  - Extra information about “historical” aspects, e.g., previous segments, fare bases, statuses as domestic or international, are needed.
- Initial state: The user’s home airport.
- Actions: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer.
- Transition model: The state resulting from taking a flight will have the flight’s destination as the new location and the flight’s arrival time as the new time.
- Goal states: A destination city.
- Action cost: Depend on different factors of the performance measure
  - A combination of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, and so on

The touring problems

- Touring problems describe a set of locations that must be visited, rather than a single goal destination.
- Traveling salesperson problem (TSP): Every city must be visited exactly once, and the tour is shortest (if required).
  - Applications: Stocking machines on shop floors, plan movements of automatic circuit-board drills, etc.
  - It is a NP-hard problem instance.



Robot navigation

- Robot navigation generalizes the route-finding problem.
- The search space for a circular robot moving on a flat surface is essentially two-dimensional.
- The robot has arms and legs that must also be controlled $\rightarrow$ the space becomes many-dimensional, one dimension per joint angle.
  - Advanced techniques are required just to make the essentially continuous search space finite.
- Real robots must also handle errors in their sensor readings and motor controls, with partial observability, and with other agents.

Quiz 01: The Towers of Hanoi

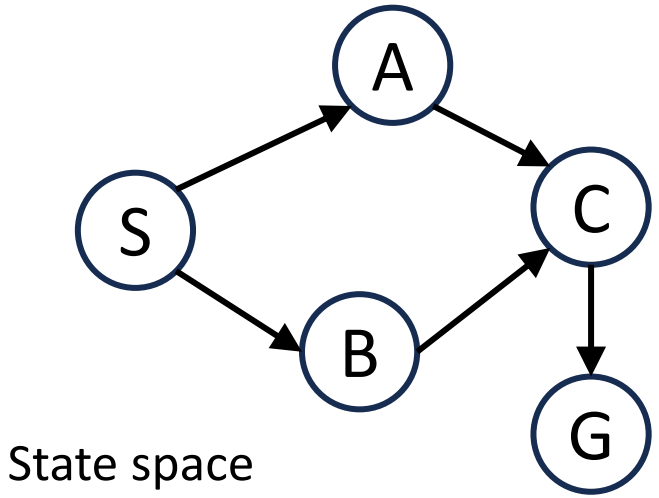
- Formulate the Towers of Hanoi problem with three pegs and three disks.

Search algorithms

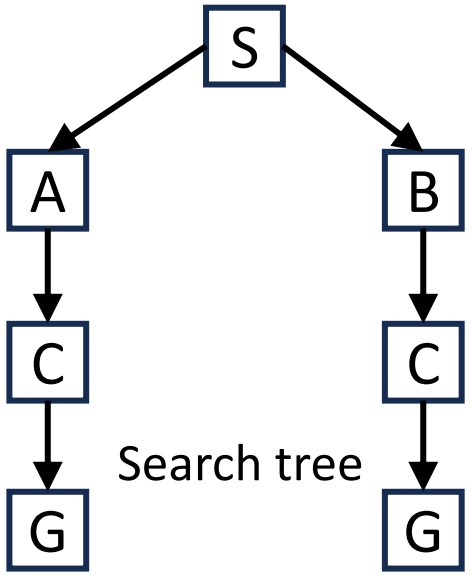
- *Best-first search*
- *Search data structures*

Search algorithms

- A search algorithm takes a search problem as input and returns a solution, or an indication of failure.
- A search tree shows various paths from the initial state, trying to find a path that reaches a goal state.
  - Node = a state in the state space. Edge = action. Root = initial state.



State space

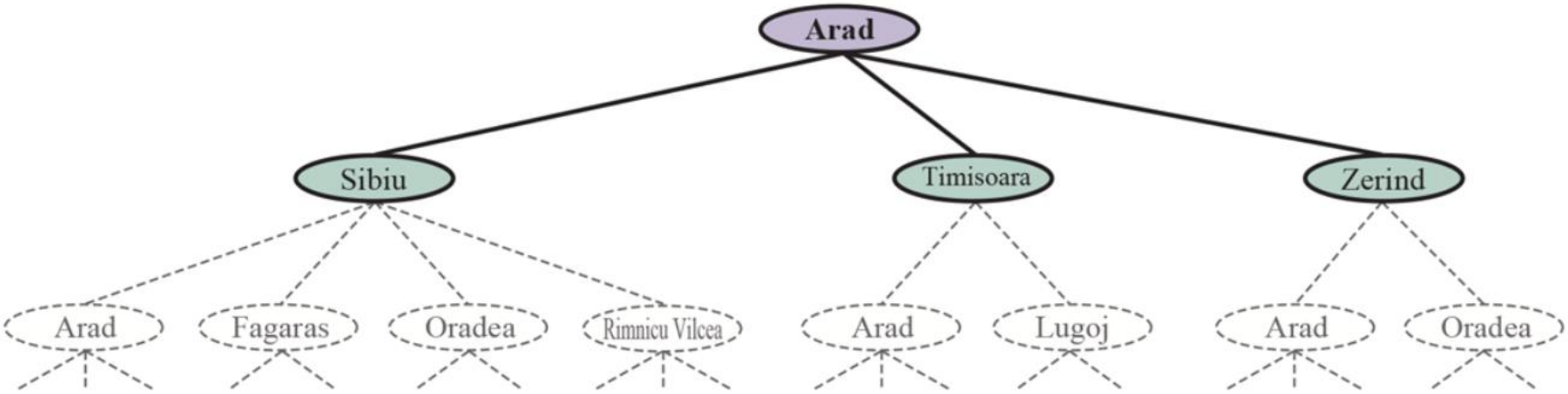


Search tree

State space vs. Search tree

- The state space describes the set of states in the world, and the actions that allow transitions from one state to another.
- The search tree describes paths between these states, reaching towards the goal.
  - There may be multiple paths any given state, but each node in the tree has a unique path back to the root.

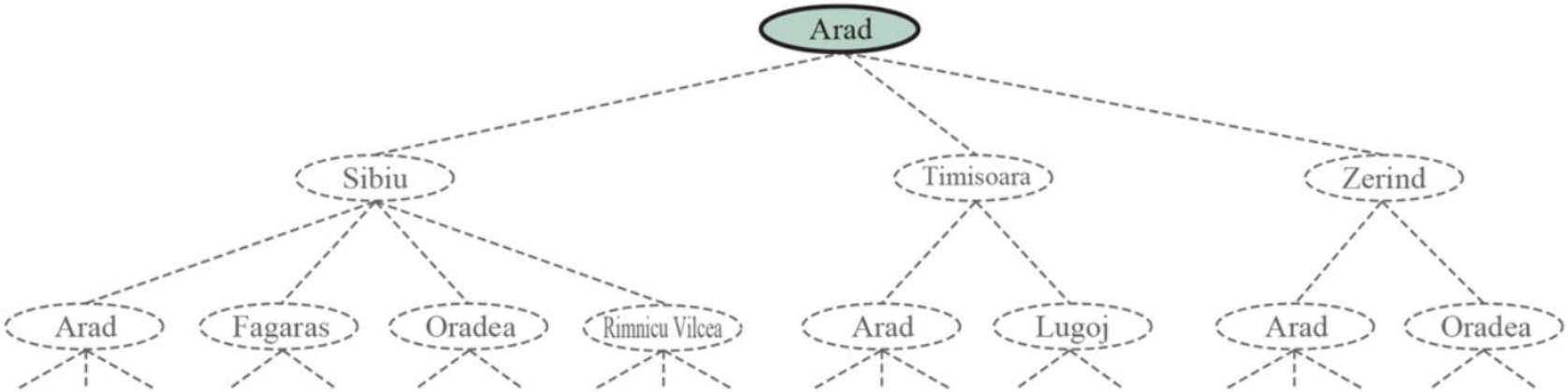
Search tree: An example for Romania map

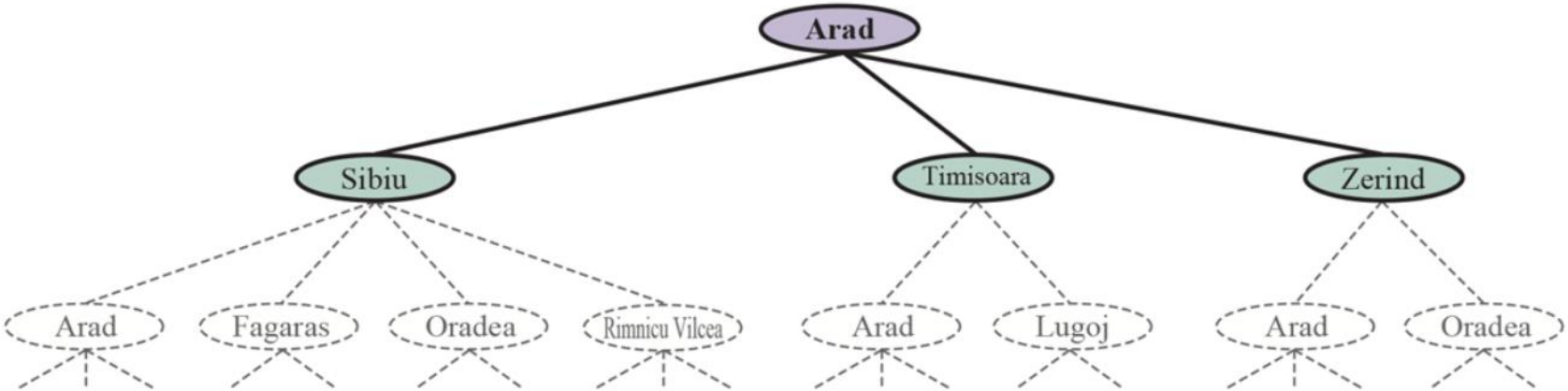


A partial search tree for finding a route from Arad to Bucharest. Nodes that have been expanded are lavender with bold letters; nodes on the frontier that have been generated but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been reached. Nodes that could be generated next are shown in faint dashed lines.

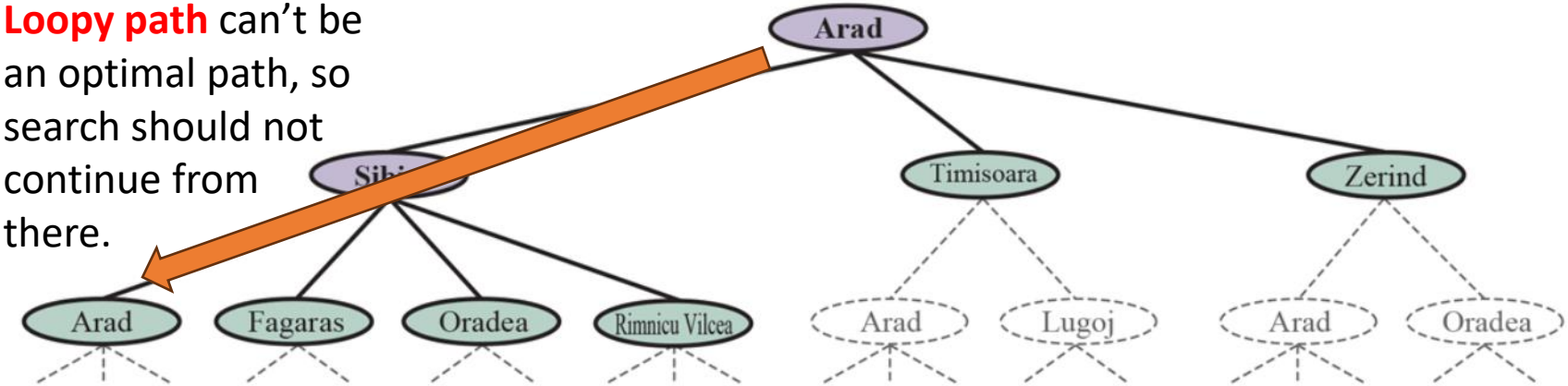
Search tree: Terminologies

- A node is called expanded if we apply available actions onto that state, leading to the generation of successor nodes.
- Those successors are added to a frontier, available for expansion at any given point.
- Any state that has had a node generated for it has been reached.
  - Reached nodes = Expanded nodes + Nodes in the frontier.

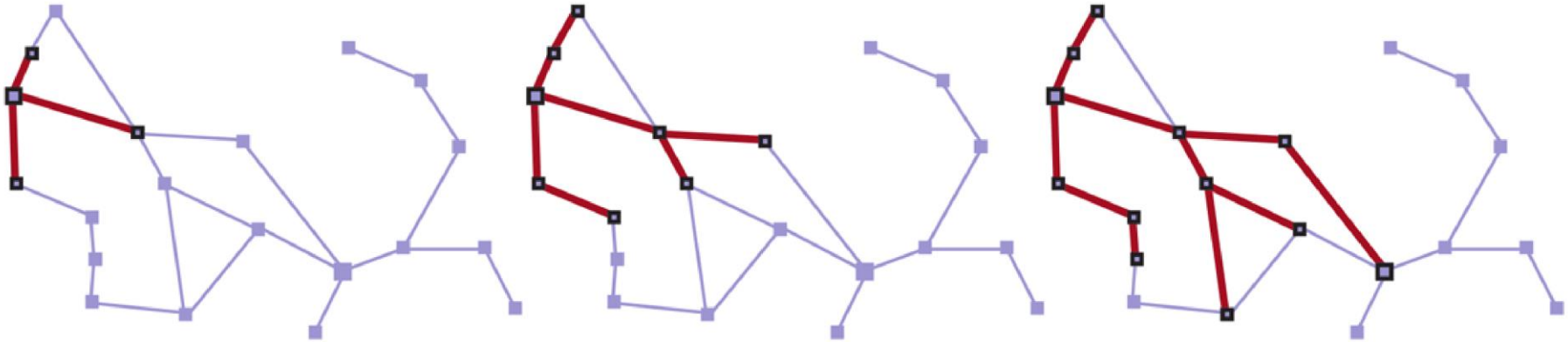




Loopy path can't be an optimal path, so search should not continue from there.



Search tree: An example for Romania map

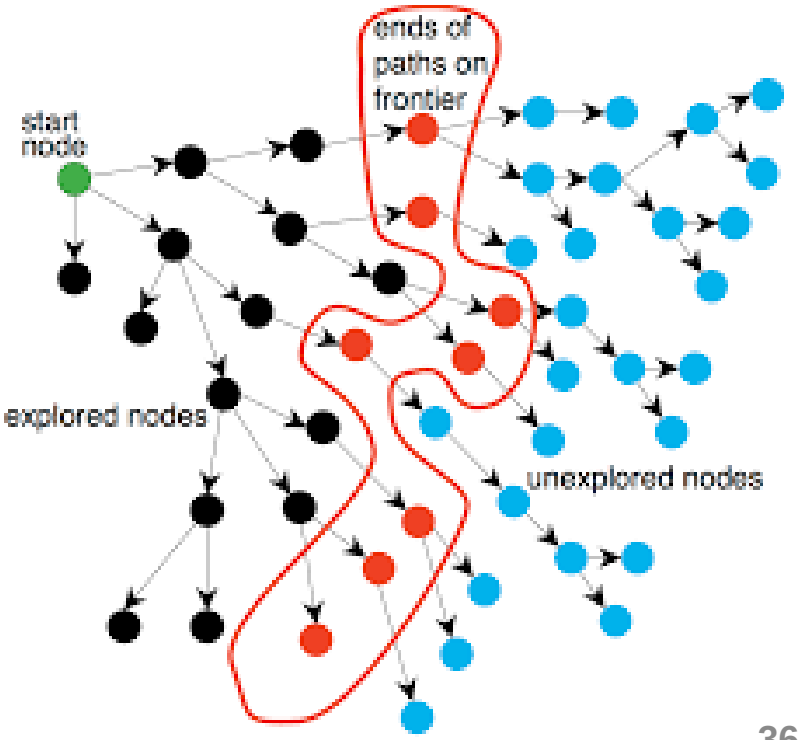


A sequence of search trees generated by a graph search on the Romania problem. At each stage, we have expanded every node on the frontier, extending every path with all applicable actions that do not result in a state that has already been reached.

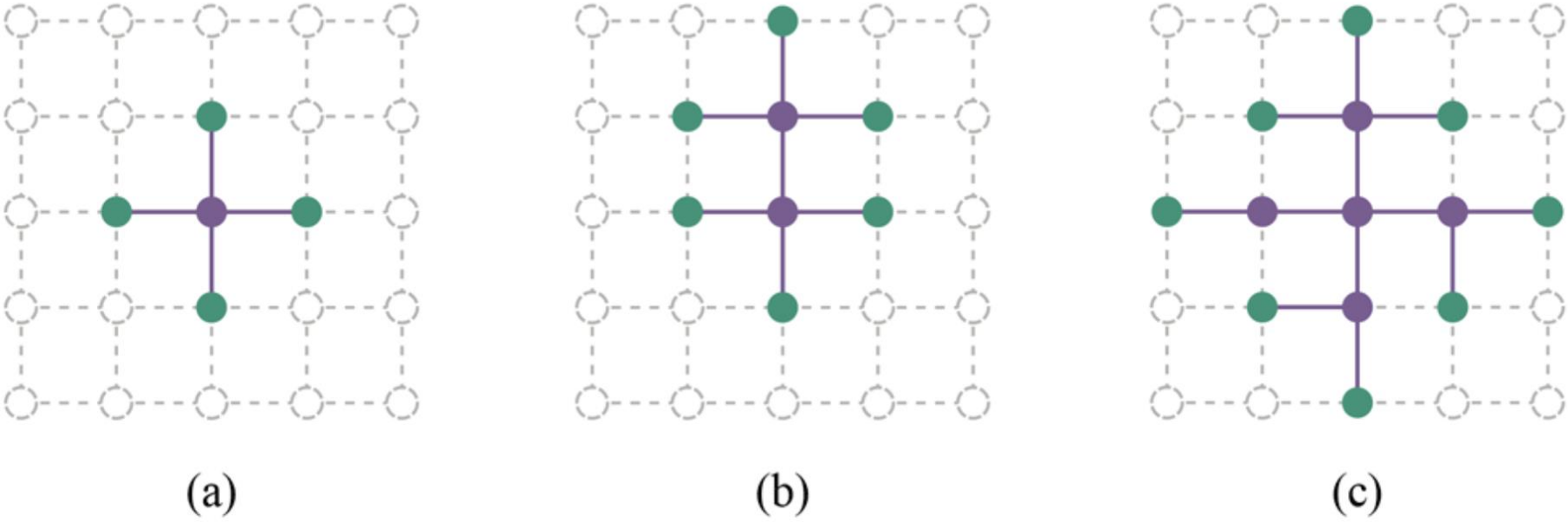
Graph-search separation property

- The frontier separates two regions of the state-space graph.
- An interior region is where every state has been expanded, and an exterior region for states not yet been reached.

- The algorithm examines the states in a systematic fashion, until it finds a solution.



Graph-search separation property



The separation property of graph search, illustrated on a rectangular-grid problem. The frontier (green) separates the interior (lavender) from the exterior (faint dashed). The frontier is the set of nodes (and corresponding states) that have been reached but not yet expanded; the interior is the set of nodes (and corresponding states) that have been expanded; and the exterior is the set of states that have not been reached. In (a), just the root has been expanded. In (b), the top frontier node is expanded. In (c), the remaining successors of the root are expanded in clockwise order.

Best-first search

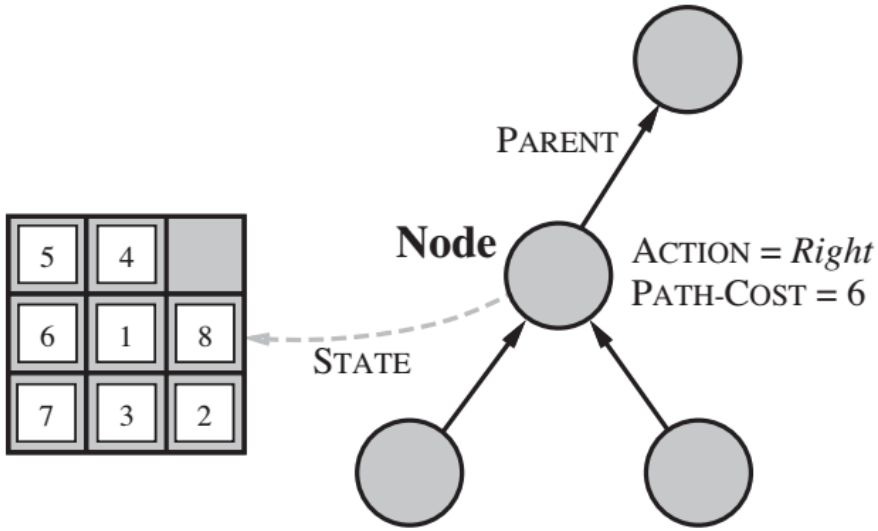
- Best-first search is a general approach that chooses a node n with minimum value of some evaluation function, $f(n)$.
- The choice of f determines the specific search strategy.

```
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 ← s.PATH-COST + problem.ACTION-COST(s, action, s')
        yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Search data structures

- We need some data structure to keep track of the search tree.
- A node *node* in the tree is represented by four components.



- The solution is formed by following the PARENT pointers back from the goal node to the initial node.

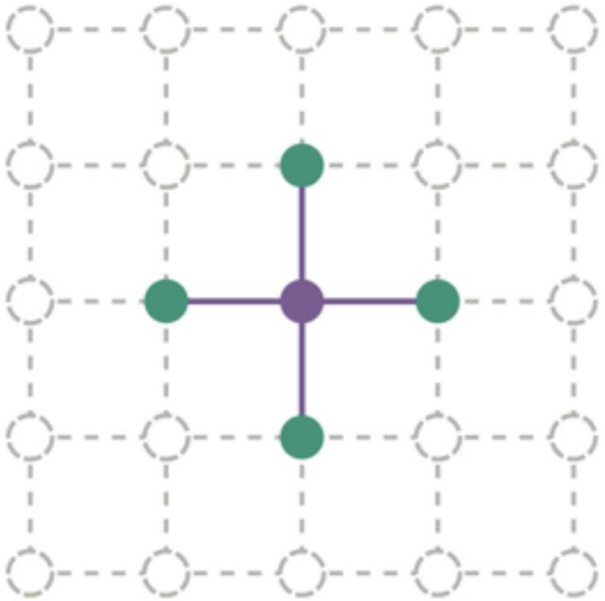
Search data structures

- The frontier can be implemented with a priority queue (UCS and A*), FIFO queue (BFS), or LIFO queue (DFS).
- The reached states can be stored in a lookup table (e.g., a hash table).
  - Each element has its state as key and node for that state as value.

Redundant paths

- A redundant path can be either a loopy or suboptimal path, which are both unavoidable.
- Following these paths may turn a tractable problem into an **intractable** one.

Each state has four successors.
A search tree of depth d: 4^d leaves with repeated states, about $2d^2$ distinct states within d steps of any given state
For $d = 20$: a trillion nodes but only about 800 distinct states.



Approaches to redundant paths

- Tree-search algorithms: No check of redundant paths.
  - There are some problem formulations where it is rare or impossible for two paths to reach the same state.
- Graph-search algorithms: Remember all previously reached states and keep only the best path to each state.
  - Appropriate for situations where there are many redundant paths, and the table of reached states will fit in memory.

"Tree-search" refers to the fact that the state space is treated as if it were a tree, with only one path from each node back to the root. The state space is still the same graph no matter how we search it.

Approaches to redundant paths

- We can compromise and check for cycles, but not for redundant paths in general.
- **Idea:** Follow up the chain of parents to see if the state at the end of the path has appeared earlier in the path.
  - This can be done with no need for additional memory.
- Some implementations follow this chain all the way up to eliminate all cycles.
- Others trace only a few links (e.g., $\rightarrow$ parent $\rightarrow$ grandparent $\rightarrow$ great-grandparent), and thus just detects short cycles.

Measuring problem-solving performance

- **Completeness:** Does it always find a solution if one exists?
- **Optimality:** Does it always find a least-cost solution?
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

- Time and space complexities are abstractly measured by the number of states and actions considered.
  - b: number of successors of a node that need to be considered
  - d: number of actions in an optimal solution
  - m: the maximum number of actions in any path (may be ∞)

Quiz 02: The Towers of Hanoi

- Draw the first two levels of the search tree for the Towers of Hanoi problem with three pegs and two disks (identical repeated states on a path can be ignored).


...the end.