CS 581

Advanced Artificial Intelligence

January 10, 2024

Announcements / Reminders

- **Please follow the Week 01 To Do List instructions (if you haven't already):**
  - **Go through the Syllabus,**
  - **Setup Python environment on your computer**

Plan for Today

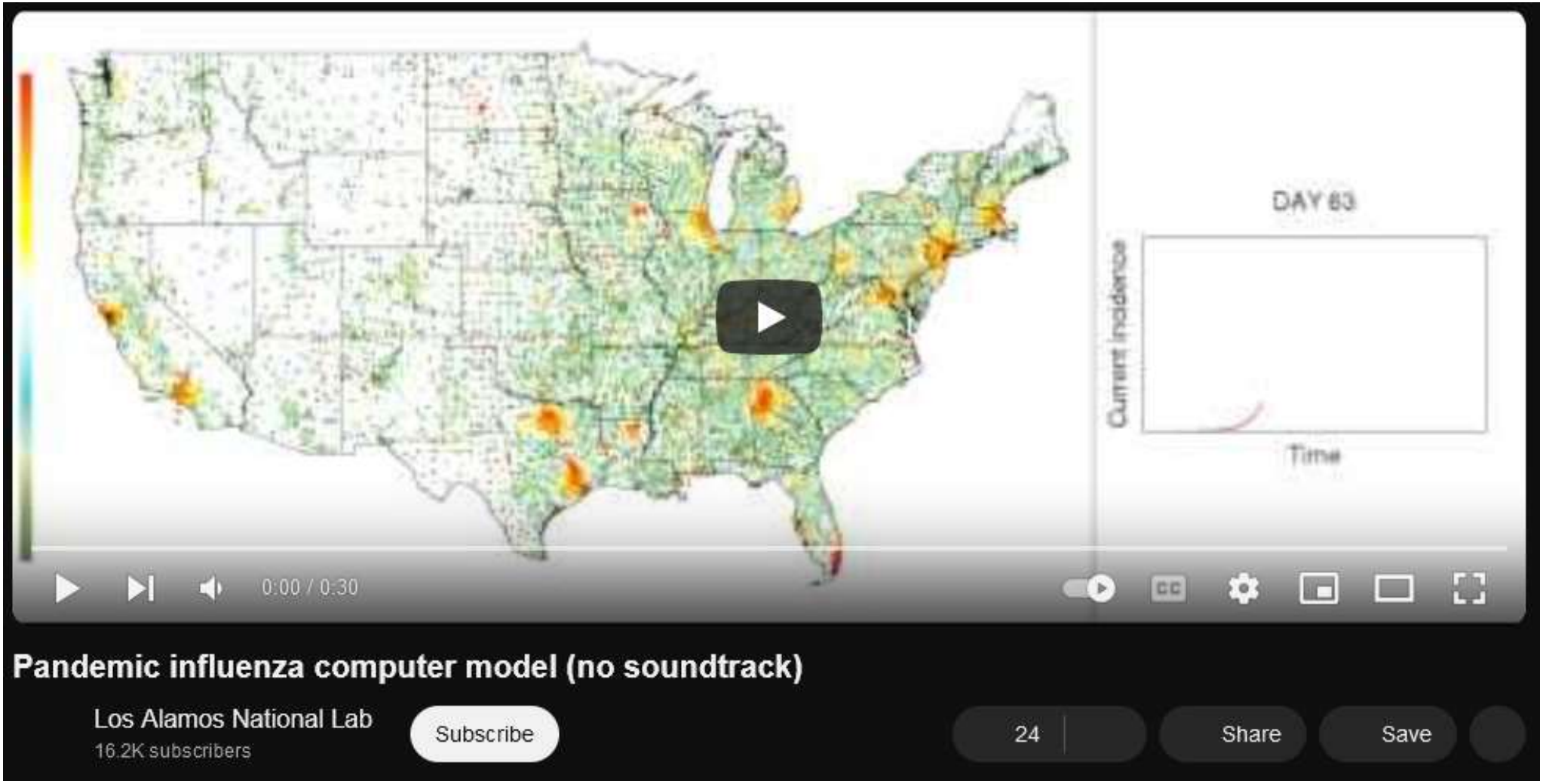
- Intelligent Agents
- Solving problems by Searching

Identyfing Problems Suitable for AI

Most AI problems will exhibit the following three characteristics:

- tend to be large,
- computationally complex and cannot be solved by a straightforward algorithm,
- tend to require a significant amount of human expertise to be solved

Agent-Based Modeling



Source: https://www.youtube.com/watch?v=E_-9hFzmxkw

Intelligent (Autonomous) Agents

Intelligent Agents in Action

Source: https://www.youtube.com/watch?v=kopoLzvh5jY

Agent

Agent:

An **agent** is just **something that acts** (from the Latin agere, to do).

Of course, we would prefer "acting" to be:

- autonomous
- situated in some environment (that could be really complex)
- adaptive
- creative and goal-oriented

Rational Agent

Rational Agent:

A **rational agent** is one that acts so as **to achieve the best outcome**, or when there is uncertainty, **the best expected outcome**.*

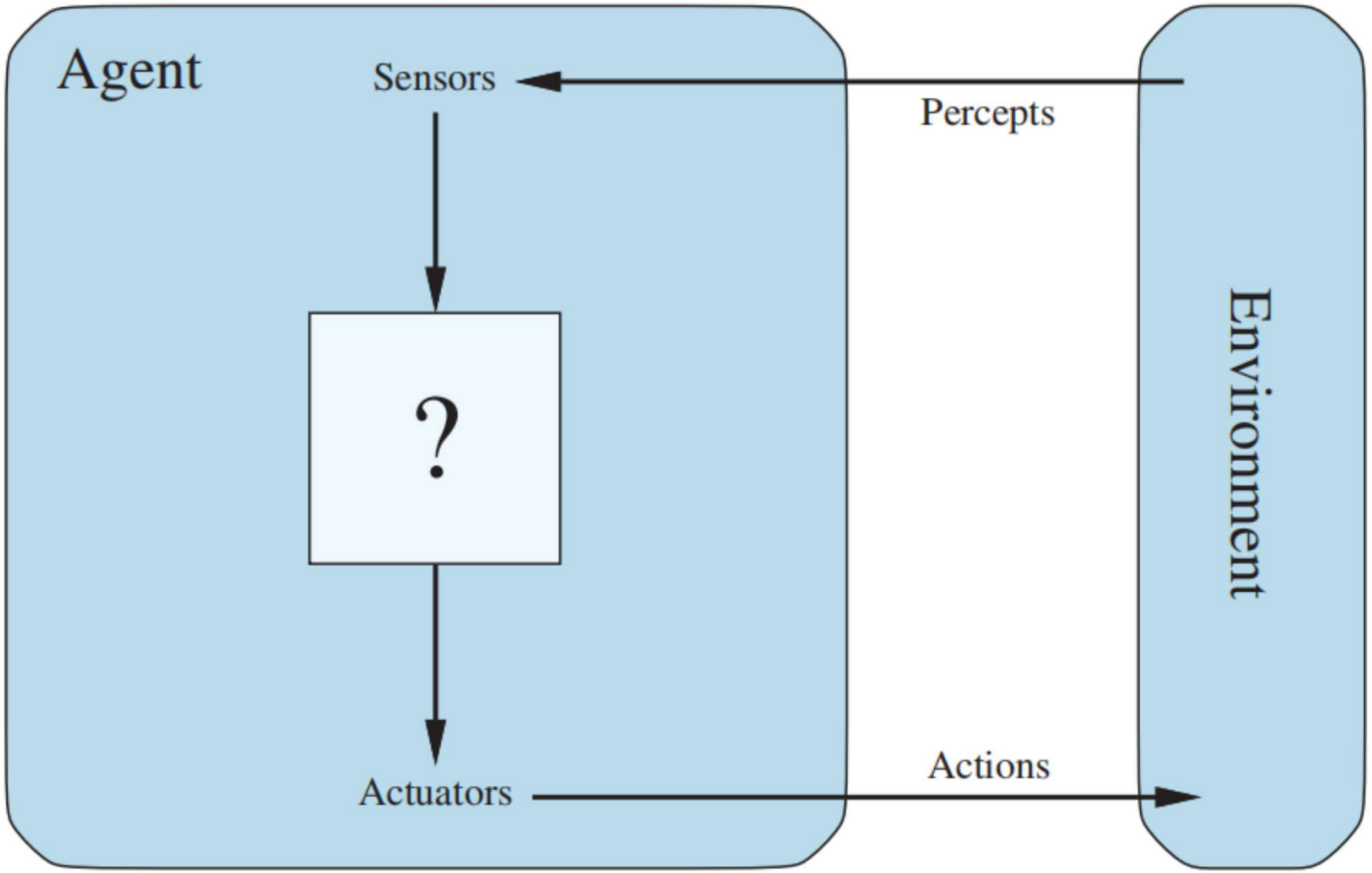
* no worries, we will make it a little less vague soon

AI: Constructing Agents

You can say that:

AI is focused on the **study and construction of agents that do the right thing**.

Agent



Agent

Agent:

An **agent** is **anything** that can be viewed as **perceiving** its **environment** through **sensors** and acting upon that environment through **actuators**.

Percepts and Percept Sequences

- **Percept: content / information that agent's sensors are perceiving / capturing currently**

- **Percept Sequence: a complete history of everything that agent has ever perceived**
  - **any practical issues that you can see here?**
  - **what can a percept sequence be used for?**

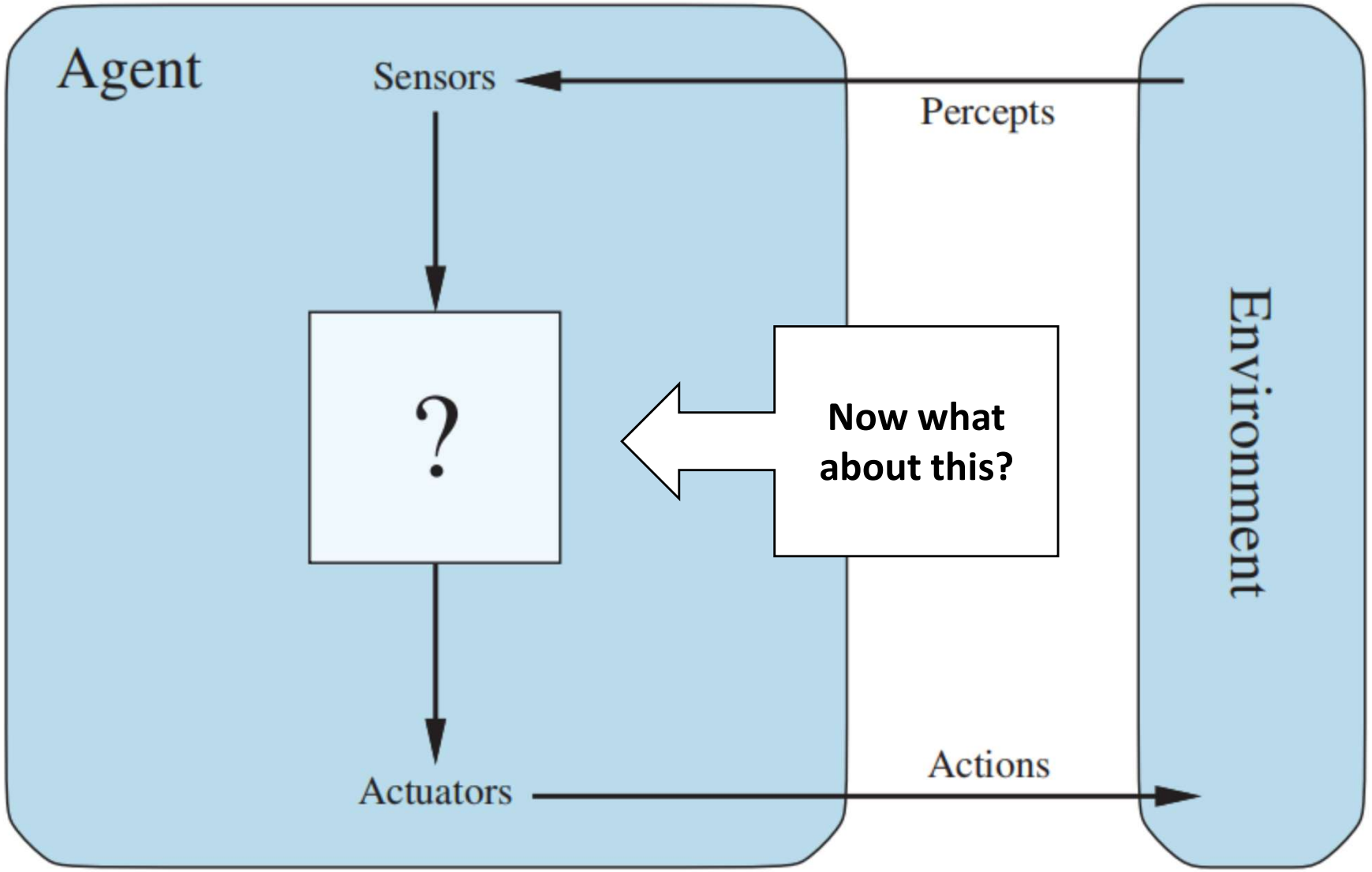
Percepts, Knowledge, Actions, States

- Agent's choice of action / decision at any given moment:
  - CAN depend on:
    - built-in knowledge
    - entire percept sequence
  - CANNOT depend anything it hasn't perceived

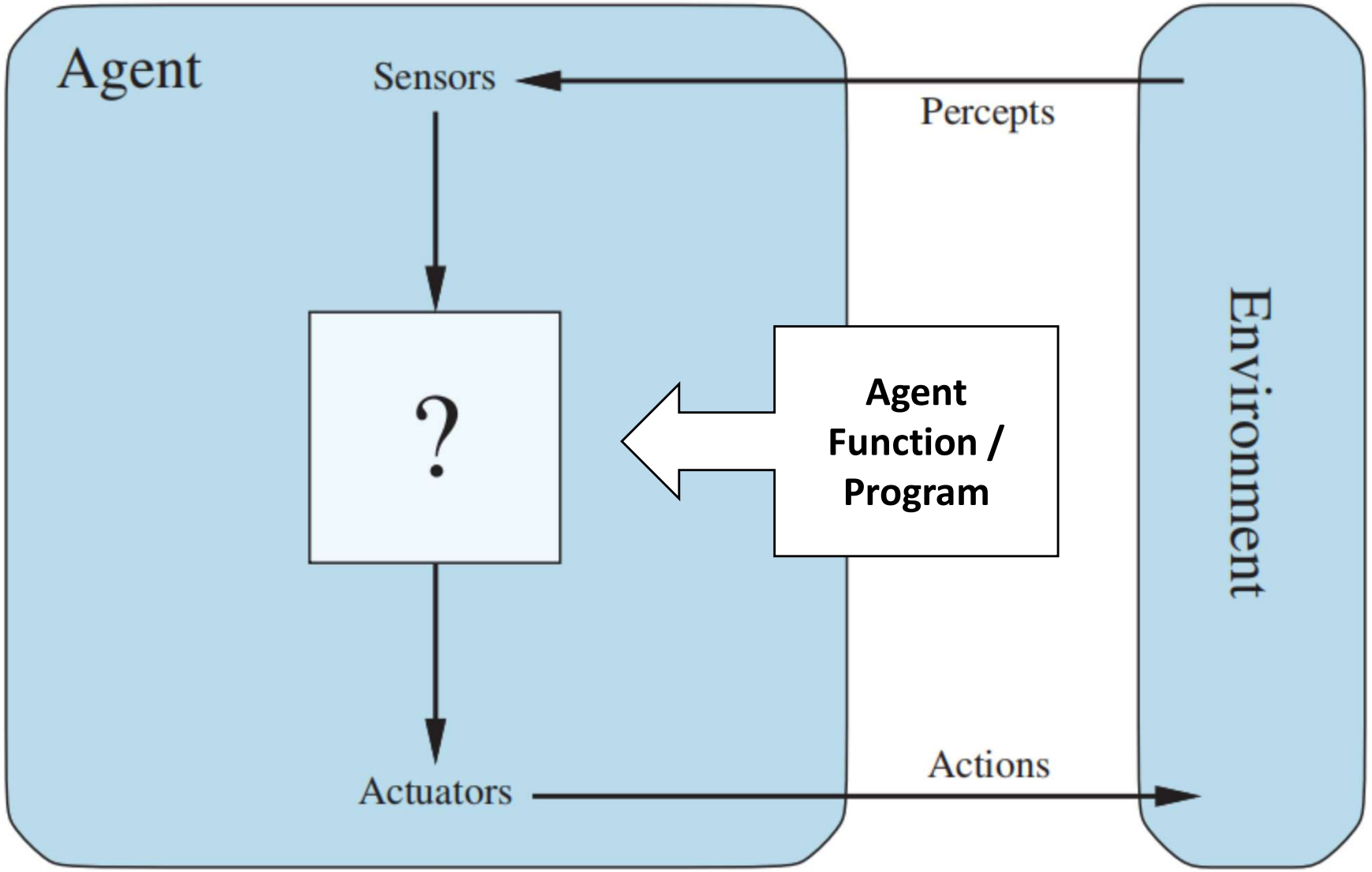
- Agent's action CAN change the environment state

Knowledge is power, right?

Agent


Agent
Sensors
Percepts
Environment
?
Now what about this?
Actuators
Actions

Agent Function / Program

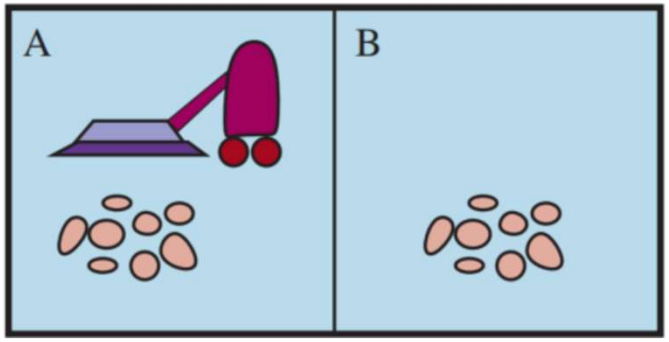


Agent Function / Program

- Specifying an action choice for every possible percept sequence would define an agent

- Action <-> percept sequence mapping IS the agent function
- Agent function describes agent behavior
- Agent function is an abstract concept
- Agent program implements agent function

Vacuum Cleaner Agent Example



Percept sequence	Action
$[A, Clean]$	$Right$
$[A, Dirty]$	$Suck$
$[B, Clean]$	$Left$
$[B, Dirty]$	$Suck$
$[A, Clean], [A, Clean]$	$Right$
$[A, Clean], [A, Dirty]$	$Suck$
⋮	⋮
$[A, Clean], [A, Clean], [A, Clean]$	$Right$
$[A, Clean], [A, Clean], [A, Dirty]$	$Suck$
⋮	⋮

Vacuum Cleaner Agent Example

function TABLE-DRIVEN-AGENT(*percept*) **returns** an action
  persistent: *percepts*, a sequence, initially empty
    table, a table of actions, indexed by percept sequences, initially fully specified

  append *percept* to the end of *percepts*
  action ← LOOKUP(*percepts*, *table*)
  return *action*

function REFLEX-VACUUM-AGENT([*location*, *status*]) **returns** an action

  if *status* = *Dirty* **then return** *Suck*
  else if *location* = *A* **then return** *Right*
  else if *location* = *B* **then return** *Left*

Actions Have Consequences

- An agent can act upon its environment, but how do we know if the end result is "right"?
- After all, actions have consequences: either good or bad.
- Recall that agent actions change environment state!
- If state changes are desirable, an agent performs well.
- Performance measure evaluates state changes.

Performance Measure: A Tip

It is better to design performance measures according to what one actually wants to be achieved in the environment, rather than according to how one thinks the agent should behave.

Performance Measure: A Warning

If it is difficult to specify the performance measure, agents may end up optimizing a wrong objective. Handle uncertainty well in such cases.

Rationality

Rational decisions at the moment depend on:

- **The performance measure that defines success criteria**
- **The agent's prior knowledge of the environment**
- **The actions that the agent can perform**
- **The agent's percept sequence so far**

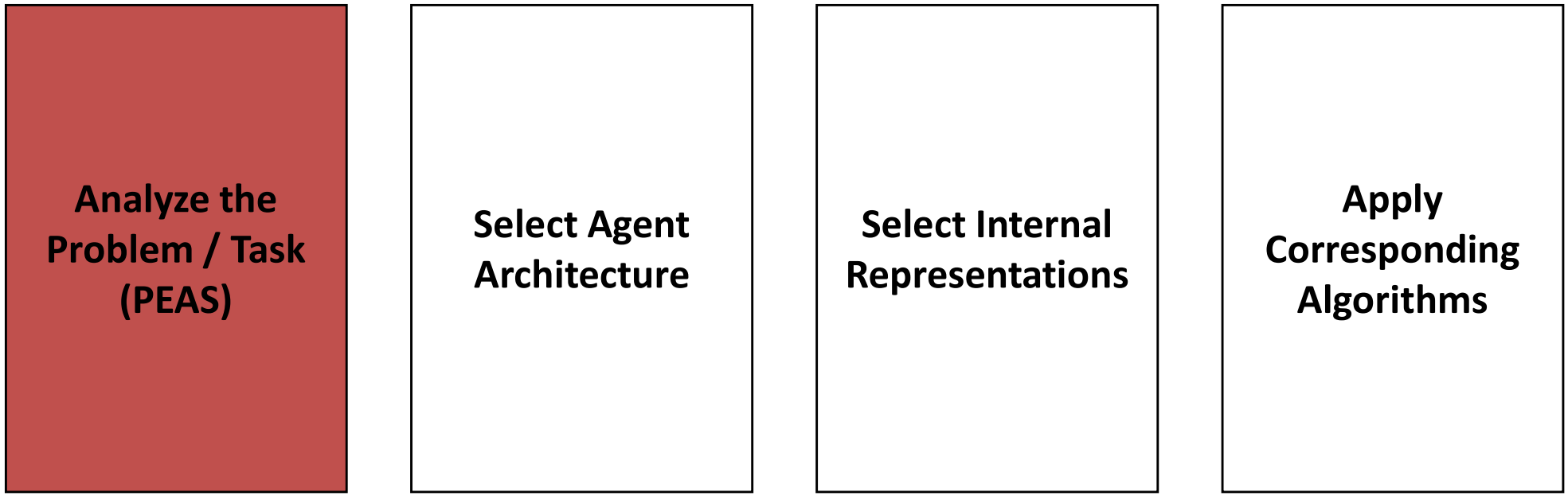
Rational Agent

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

Rationality in Reality

- An omniscient agent will ALWAYS know the final outcome of its action. Impossible in reality. That would be perfection.
- Rationality maximizes what is EXPECTED to happen
- Perfection maximizes what WILL happen
- Performance can be improved by information gathering and learning

Designing the Agent for the Task



Task Environment | PEAS

In order to start the agent design process we need to specify / define:

- **The Performance measure**
- **The Environment in which the agent will operate**
- **The Actuators that the agent will use to affect the environment**
- **The Sensors that the agent will use to perceive the environment**

PEAS: Taxi Driver Example

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits, minimize impact on other road users	Roads, other traffic, police, pedestrians, customers, weather	Steering, accelerator, brake, signal, horn, display, speech	Cameras, radar, speedometer, GPS, engine sensors, accelerometer, microphones, touchscreen

PEAS: Other Examples

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments	Touchscreen/voice entry of symptoms and findings
Satellite image analysis system	Correct categorization of objects, terrain	Orbiting satellite, downlink, weather	Display of scene categorization	High-resolution digital camera
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, tactile and joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, raw materials, operators	Valves, pumps, heaters, stirrers, displays	Temperature, pressure, flow, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, feedback, speech	Keyboard entry, voice

Task Environment Properties

Key dimensions by which task environments can be categorized:

- **Fully vs partially observable (can be unobservable too)**
- **Single agent vs multiagent**
  - **multiagent: competitive vs. cooperative**
- **Deterministic vs. nonderministic (stochastic)**
- **Episodic vs. sequential**
- **Static vs. dynamic**
- **Discrete vs. continuous**
- **Known vs. unknown (to the agent)**

Fully Observable Environment

Source: Pixabay (www.pexels.com)

Partially Observable Environment

Source: https://en.wikipedia.org/wiki/Fog_of_war

Partially Observable Environment

Source: https://en.wikipedia.org/wiki/Fog_of_war

Single-agent System

Source: cottonbro (www.pexels.com)

Single-agent System

Source: cottonbro (www.pexels.com)

Multiagent System

Source: Vlada Karpovich (www.pexels.com)

Multiagent System

Source: Vlada Karpovich (www.pexels.com)

Deterministic vs. Nondeterministic

- **Deterministic environment:**
  - **next state is completely determined by the current state and agent action**
  - **deterministic AND fully observable environment: no need to worry about uncertainty**
  - **deterministic AND partially observable ***may*** appear nondeterministic**
- **Nondeterministic (stochastic) environment:**
  - **next state is NOT completely determined by the current state and agent action**

Episodic vs. Sequential

- **Episodic environment:**
  - **agent experience is divided into individual, independent, and atomic episodes**
  - **one percept - one action.**
  - **next action is not a function of previous action: not necessary to memorize it**
- **Sequential environment:**
  - **current decision / action COULD affect all future decisions / actions**
  - **better keep track of it**

Static vs. Dynamic

- **Static environment:**
  - **environment CANNOT change while the agent is taking its time to decide**
- **Dynamic environment:**
  - **environment CAN change while the agent is taking its time to decide -> decision / action may be dated**
  - **speed is important**

Discrete vs. Continuous

- **Discrete environment:**
  - state changes are discrete
  - time changes are discrete
  - percepts are discrete
- **Continuous environment:**
  - state changes are continuous (“fluid”)
  - time changes are continuous
  - percepts / actions can be continuous

Known vs. Unknown (to Agent)

- **Known environment:**
  - **agent knows all outcomes to its actions (or their probabilities)**
  - **agent "knows how the environment works"**
- **Unknown environment:**
  - **agent "doesn't know all the details about the inner workings of the environment"**
  - **learning and exploration can be necessary**

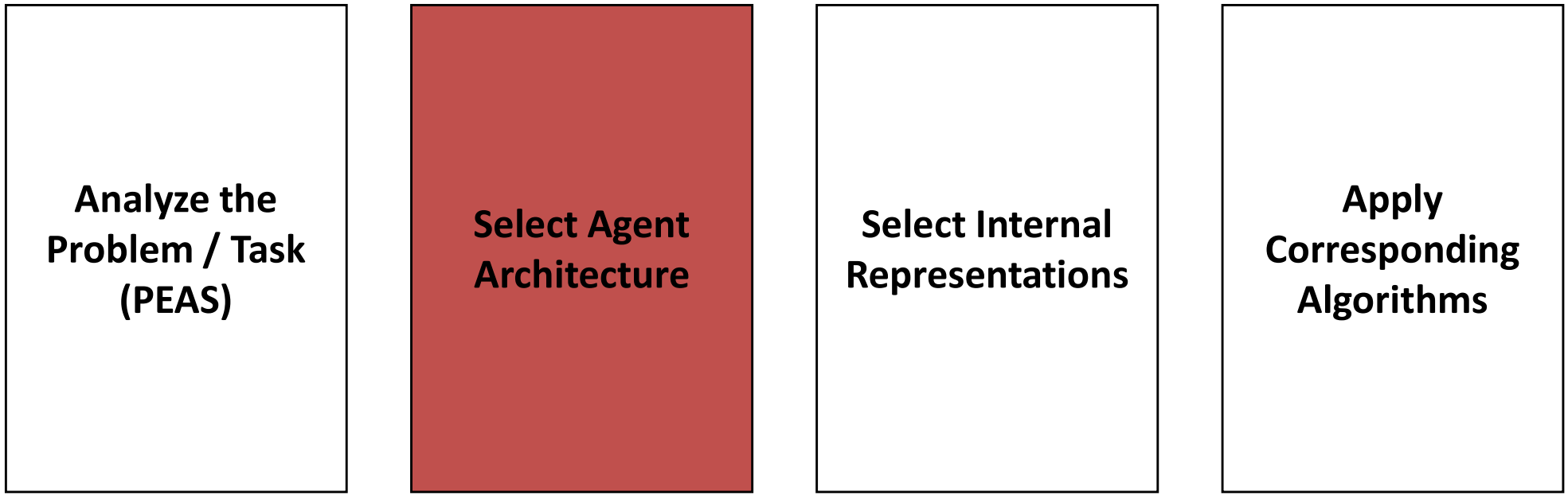
Task Environment Characteristics

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Hardest Case / Problem

- Partially observable (incomplete information, uncertainty)
- Multiagent (complex interactions)
- Nondeterministic (uncertainty)
- Sequential (planning usually necessary)
- Dynamic (changing environment, uncertainty)
- Continuous (infinite number of states)
- Unknown (agent needs to learn / explore, uncertainty)

Designing the Agent for the Task



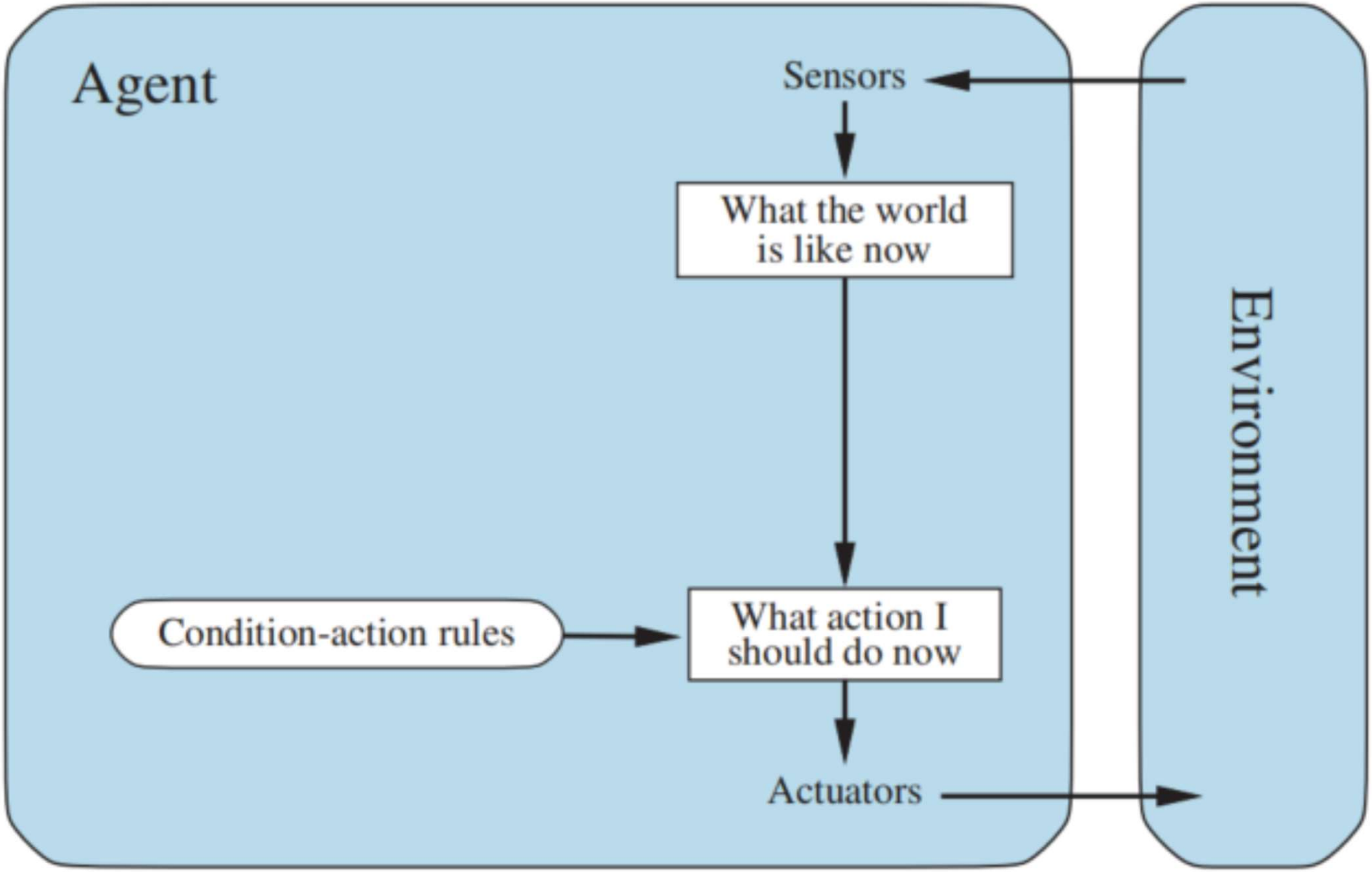
Agent Structure / Architecture

Agent = Architecture + Program

Typical Agent Architectures

- Simple reflex agent
- Model-based reflex agent:
- Goal-based reflex agent
- Utility-based reflex agent

Simple Reflex Agent

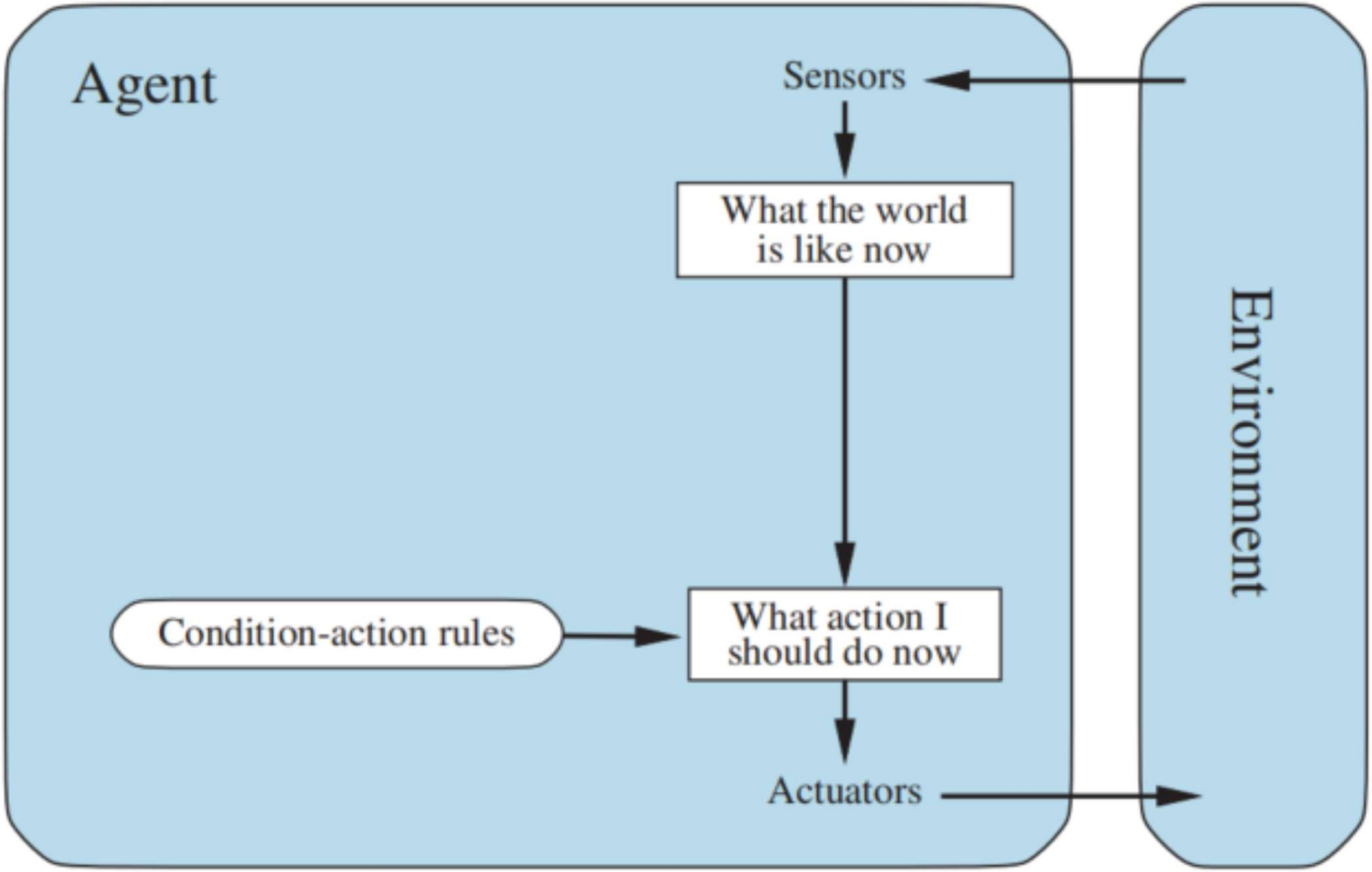


Simple Reflex Agent

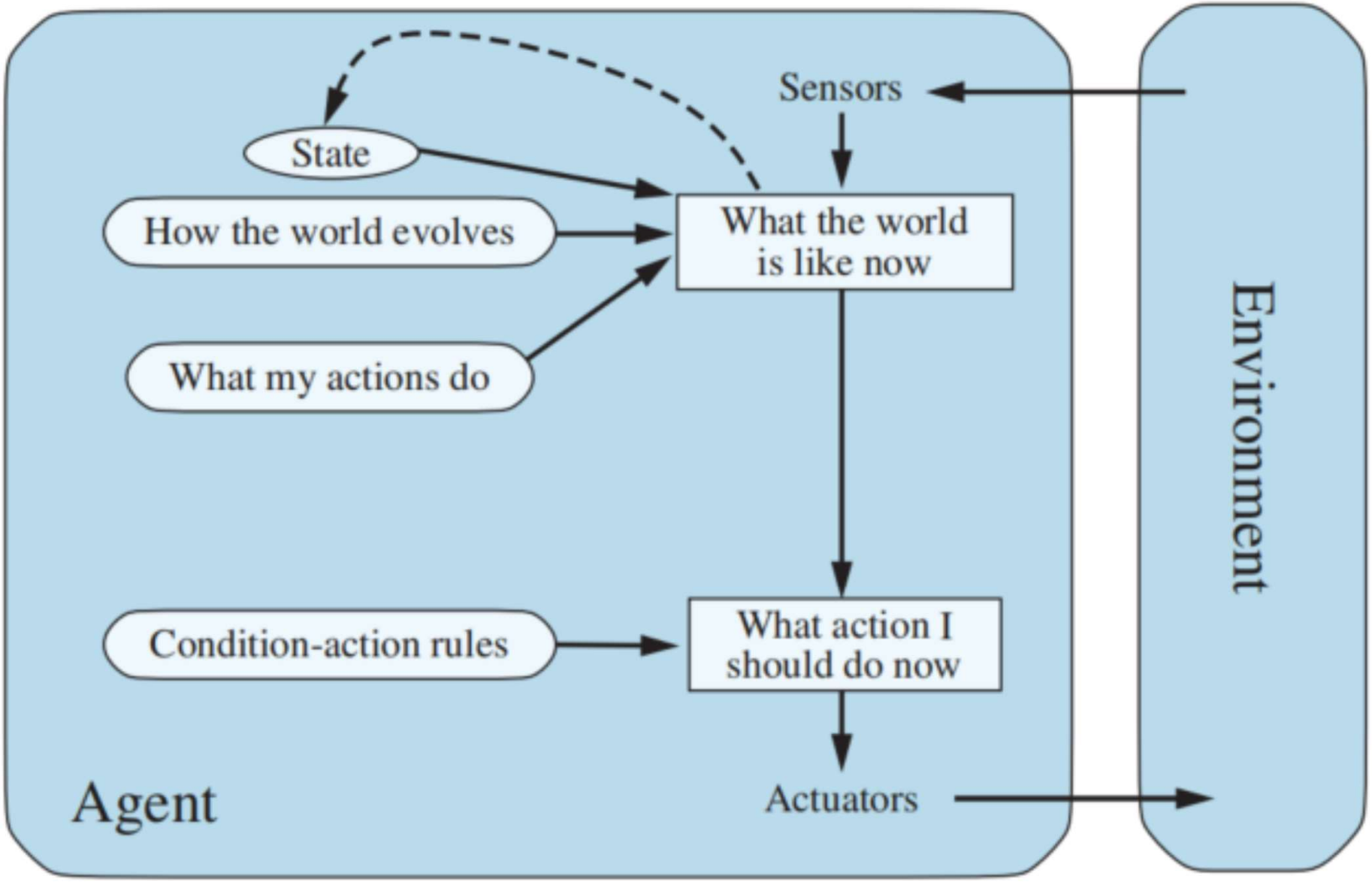
function SIMPLE-REFLEX-AGENT(*percept*) **returns** an action
  persistent: *rules*, a set of condition–action rules

  state ← INTERPRET-INPUT(*percept*)
  rule ← RULE-MATCH(*state*, *rules*)
  action ← *rule*.ACTION
  return *action*

Simple Reflex Agent: Challenges?



Model-based Reflex Agent

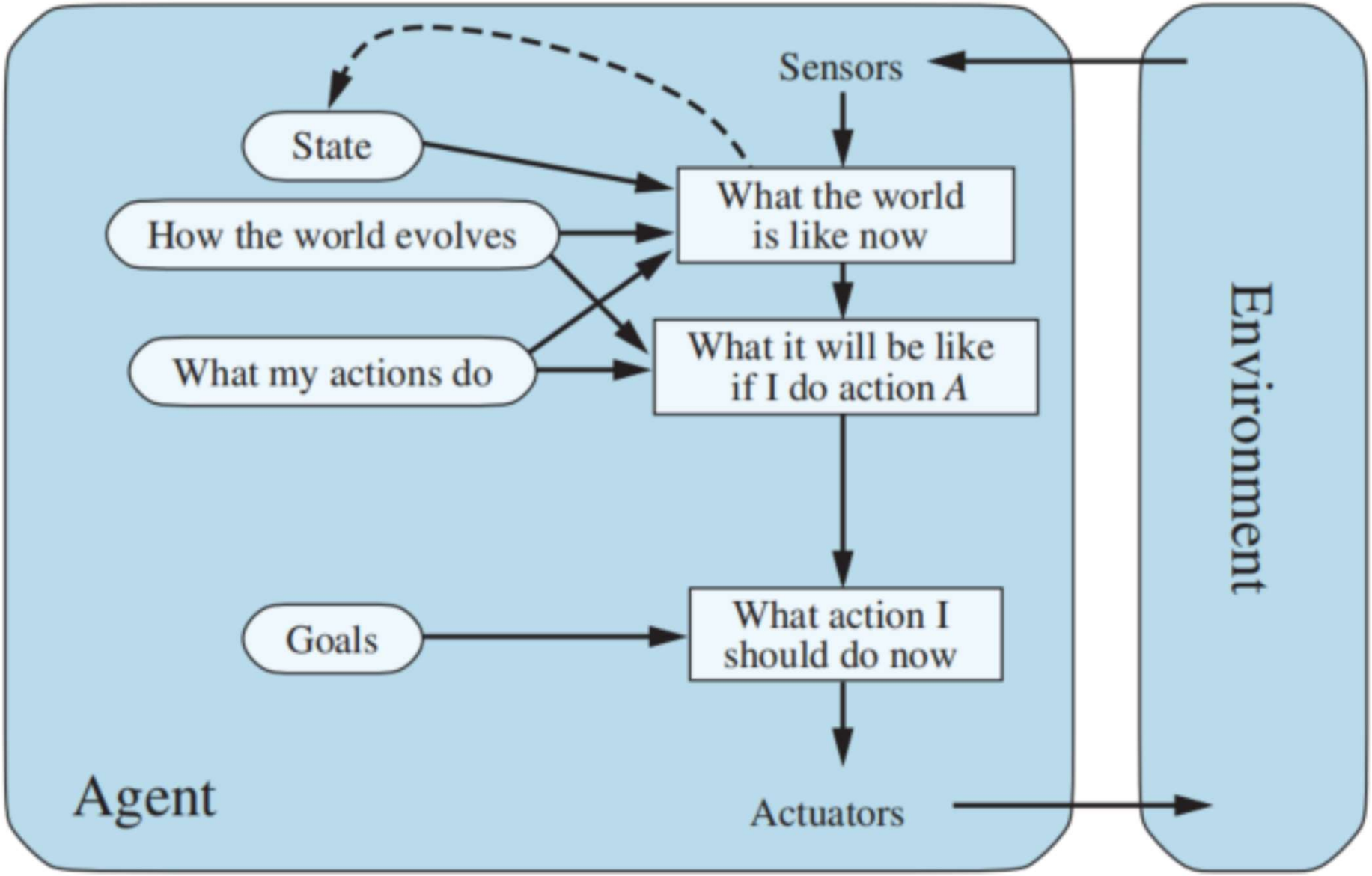


Model-based Reflex Agent

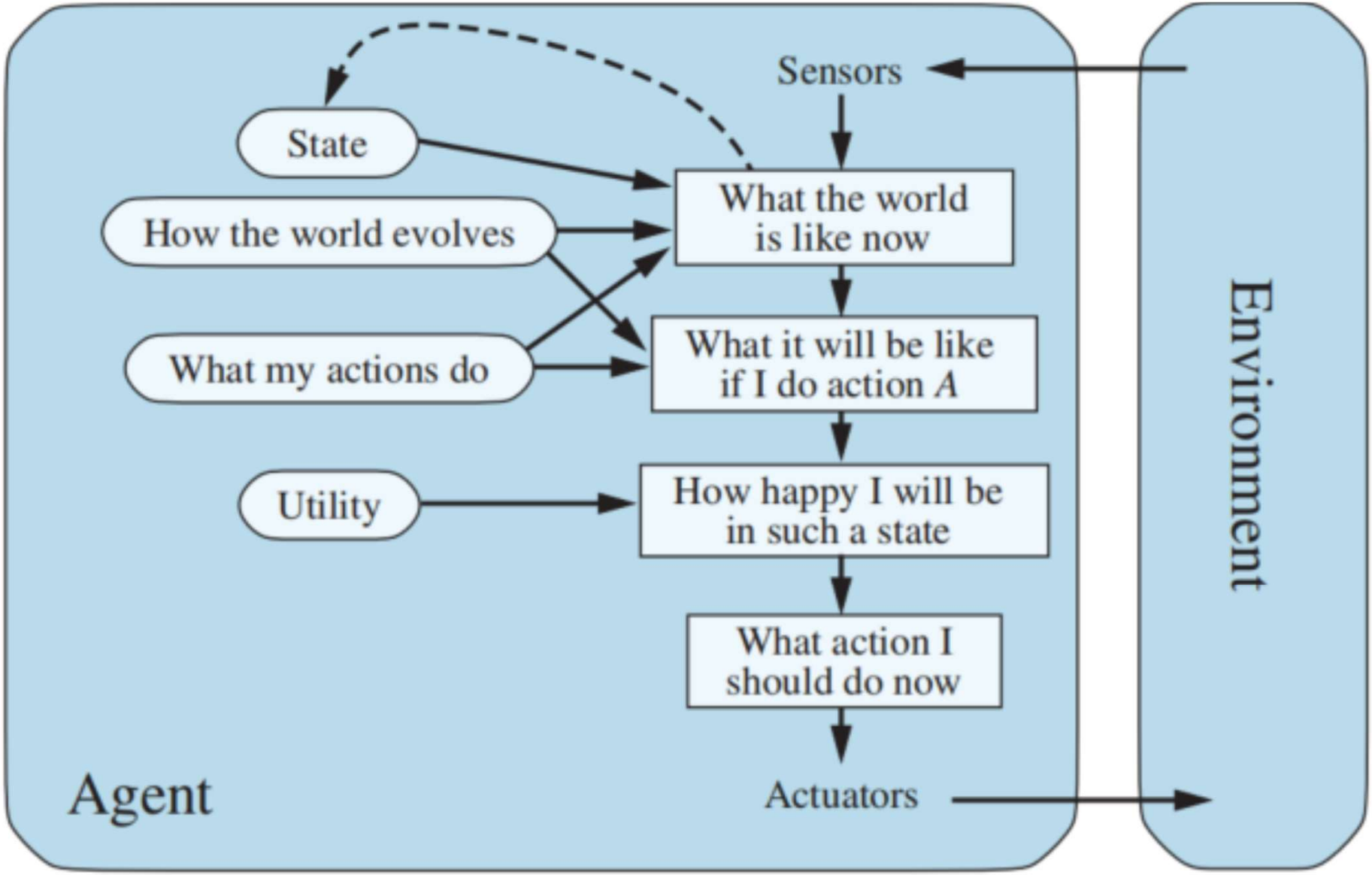
function MODEL-BASED-REFLEX-AGENT(*percept*) **returns** an action
  persistent: *state*, the agent's current conception of the world state
    transition_model, a description of how the next state depends on the current state and action
    sensor_model, a description of how the current world state is reflected in the agent's percepts
    rules, a set of condition–action rules
    action, the most recent action, initially none

  state ← UPDATE-STATE(*state*, *action*, *percept*, *transition_model*, *sensor_model*)
  rule ← RULE-MATCH(*state*, *rules*)
  action ← *rule*.ACTION
  return *action*

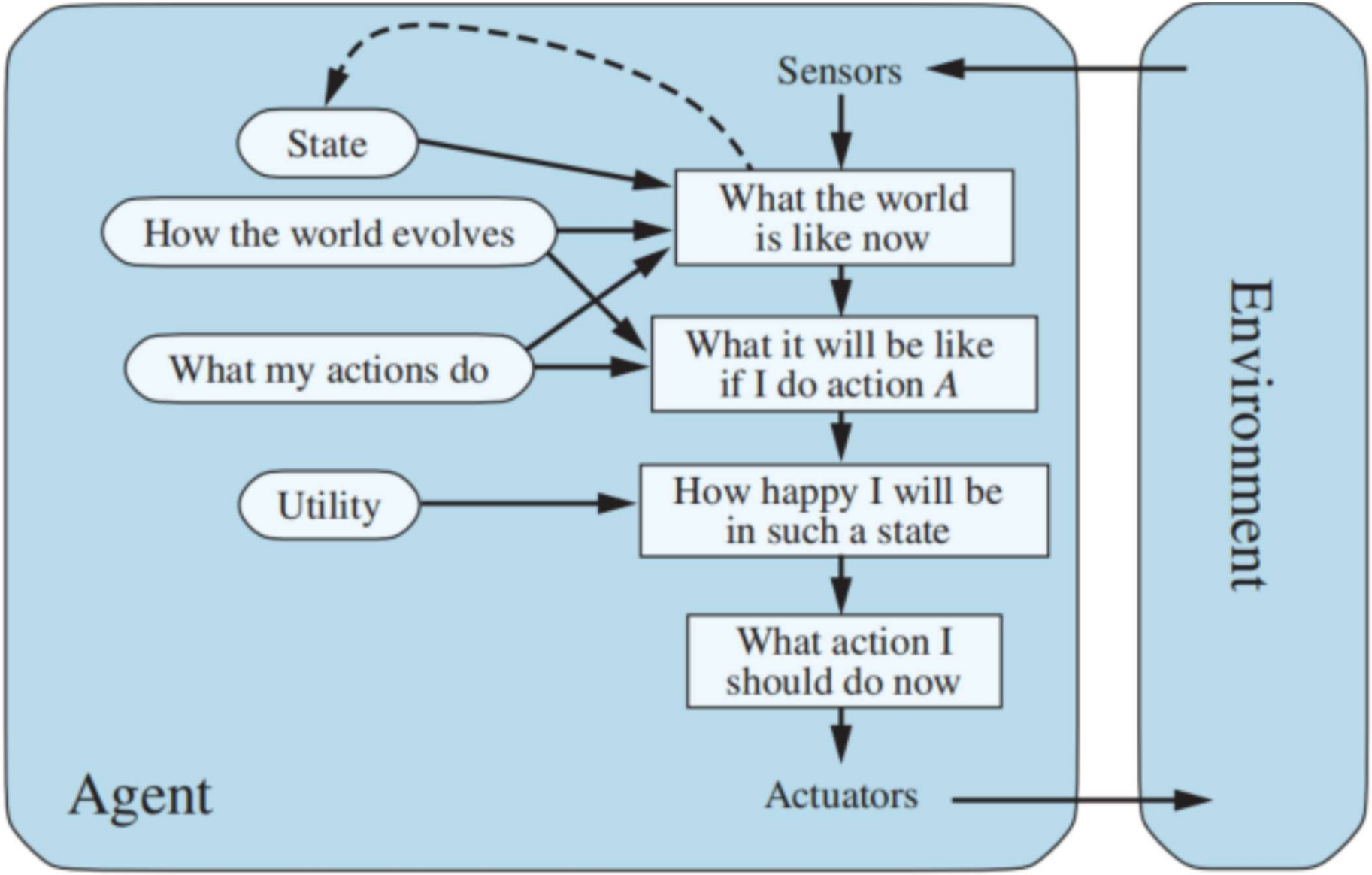
Model-based Goal-based Agent



Model-based Utility-based Agent



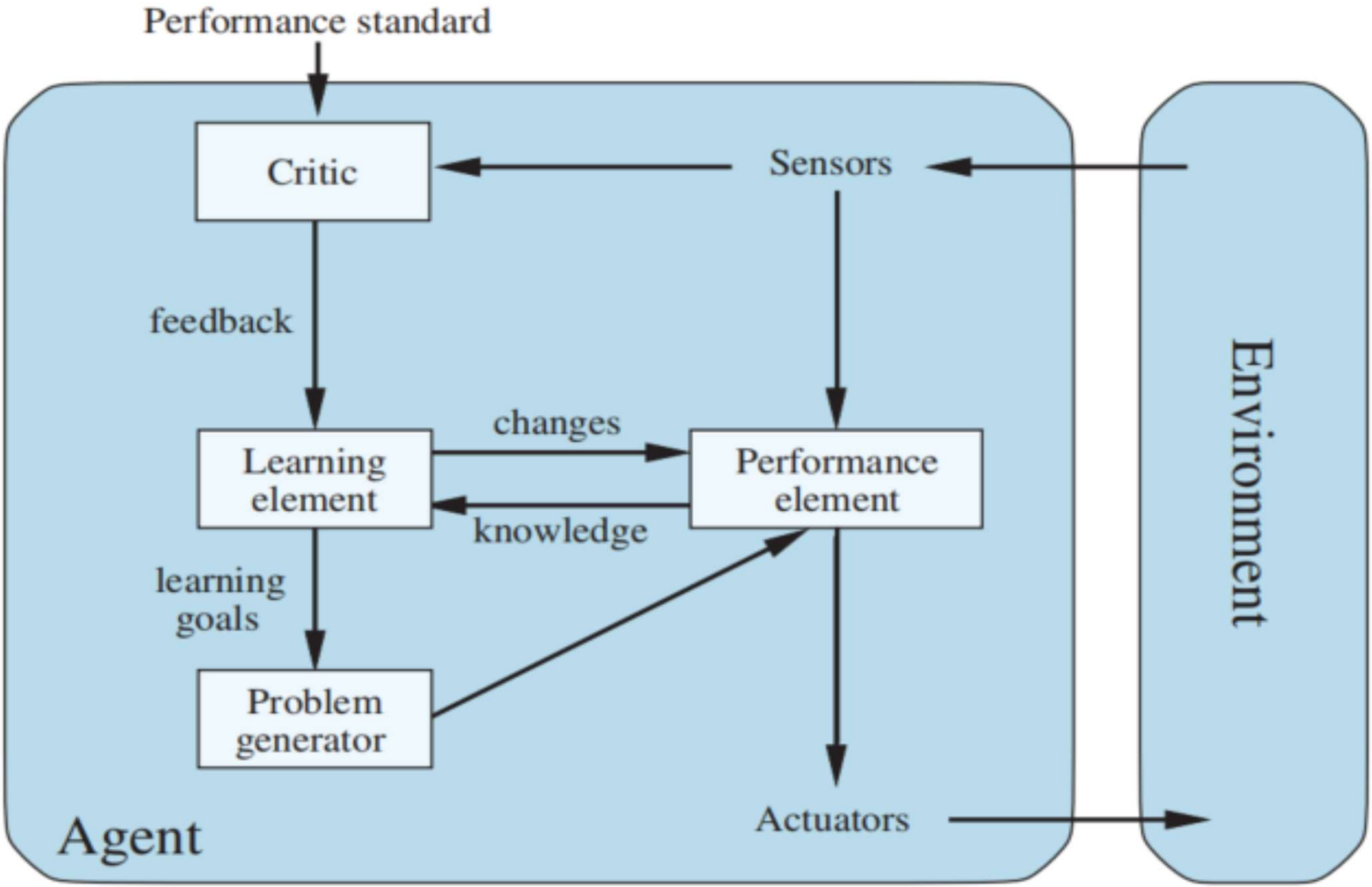
Model-based Agents: Challenges?



Typical Agent Architectures

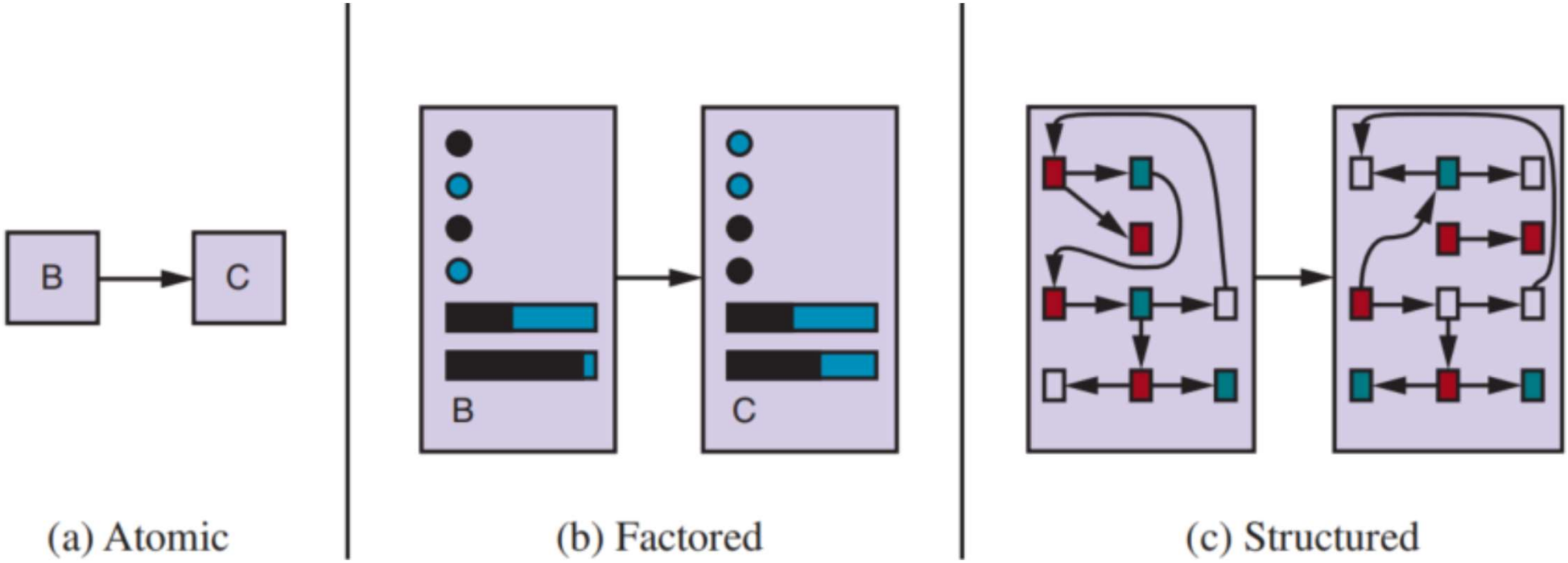
- Simple reflex agent: uses condition-action rules
- Model-based reflex agent: keeps track of the unobserved parts of the environment by maintaing internal state:
  - "how the world works": state transition model
  - how percepts and environment is related: sensor model
- Goal-based reflex agent: maintains the model of the world and goals to select decisions (that lead to goal)
- Utility-based reflex agent: maintains the model of the world and utility function to select PREFERRED decisions (that lead to the best expected utility: avg (EU * p))

Learning Agent



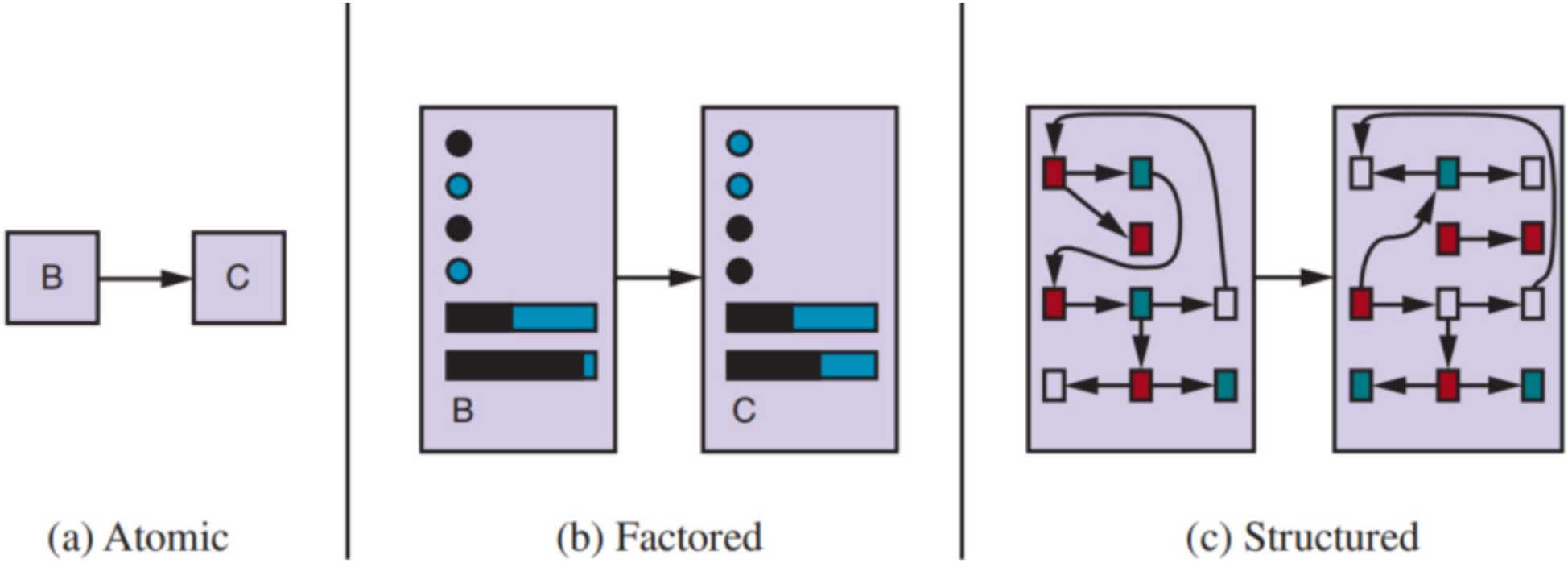
Designing the Agent for the Task

State and Transition Representations



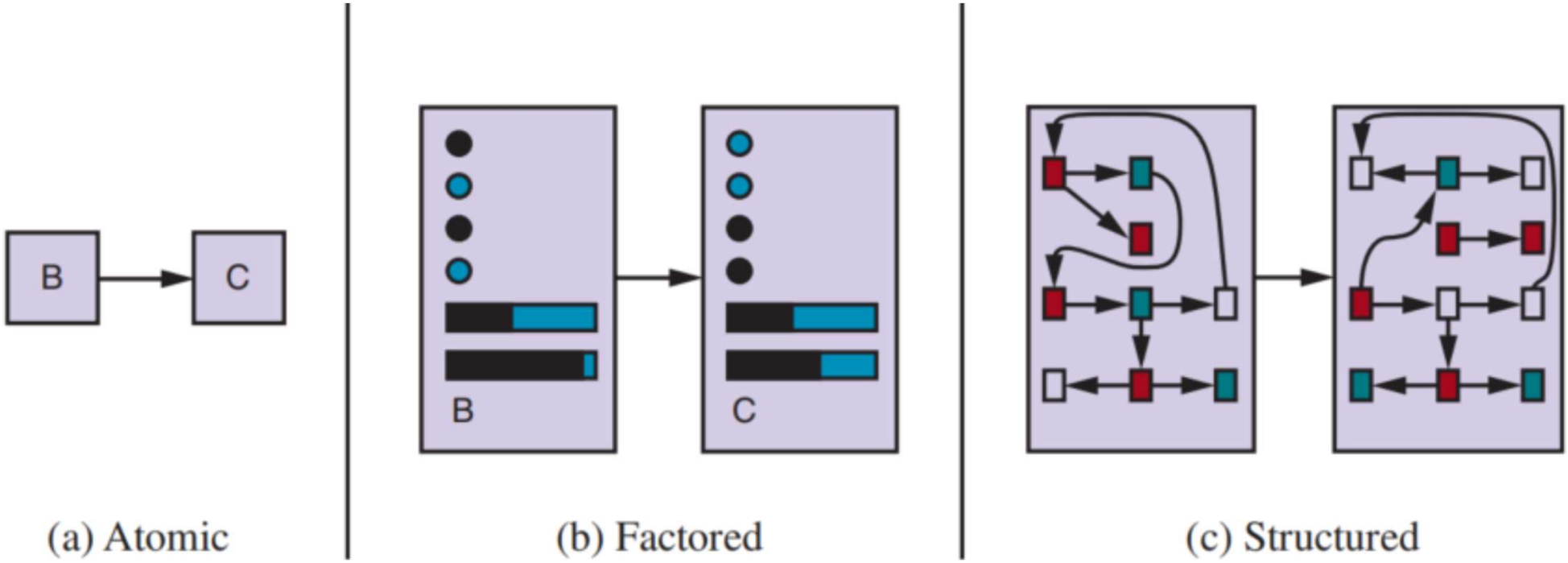
- **Atomic: state representation has NO internal structure**
- **Factored: state representation includes fixed attributes (which can have values)**
- **Structured: state representation includes objects and their relationships**

State and Transition Representations



Complexity, level of detail, expresiveness, more difficult to process

Representations and Algorithms

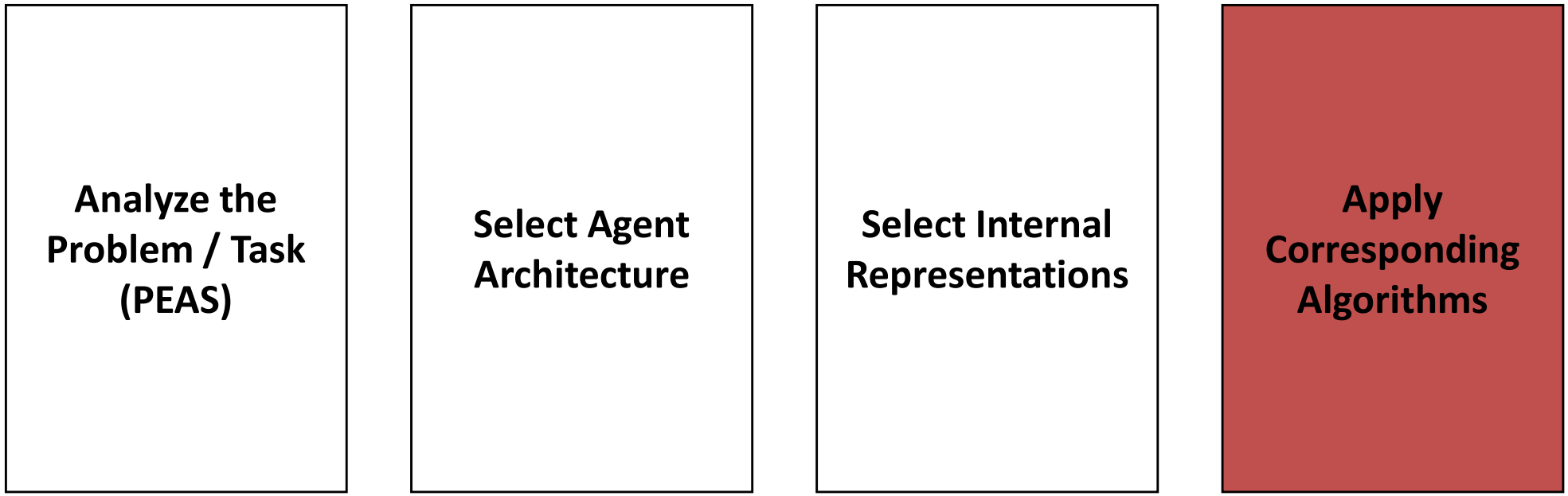


- Searching
- Hidden Markov models
- Markov decision process
- Finite state machines

- Constraint satisfaction algorithms
- Propositional logic
- Planning
- Bayesian algorithms
- Some machine learning algorithms

- Relational database algorithms
- First-order logic
- First-order probability models
- Natural language understanding (some)

Designing the Agent for the Task



Finite State Machine: A Turnstile

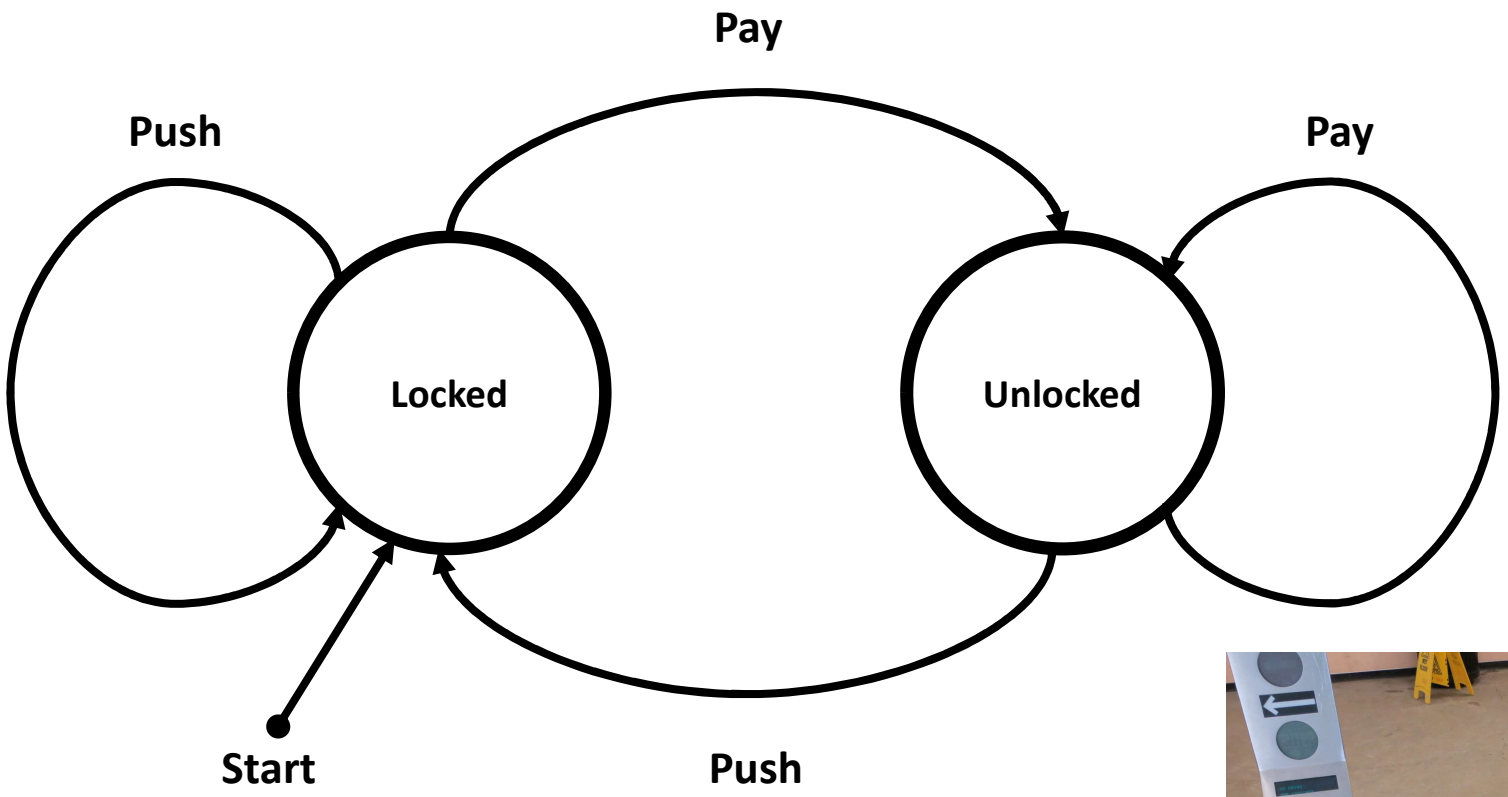
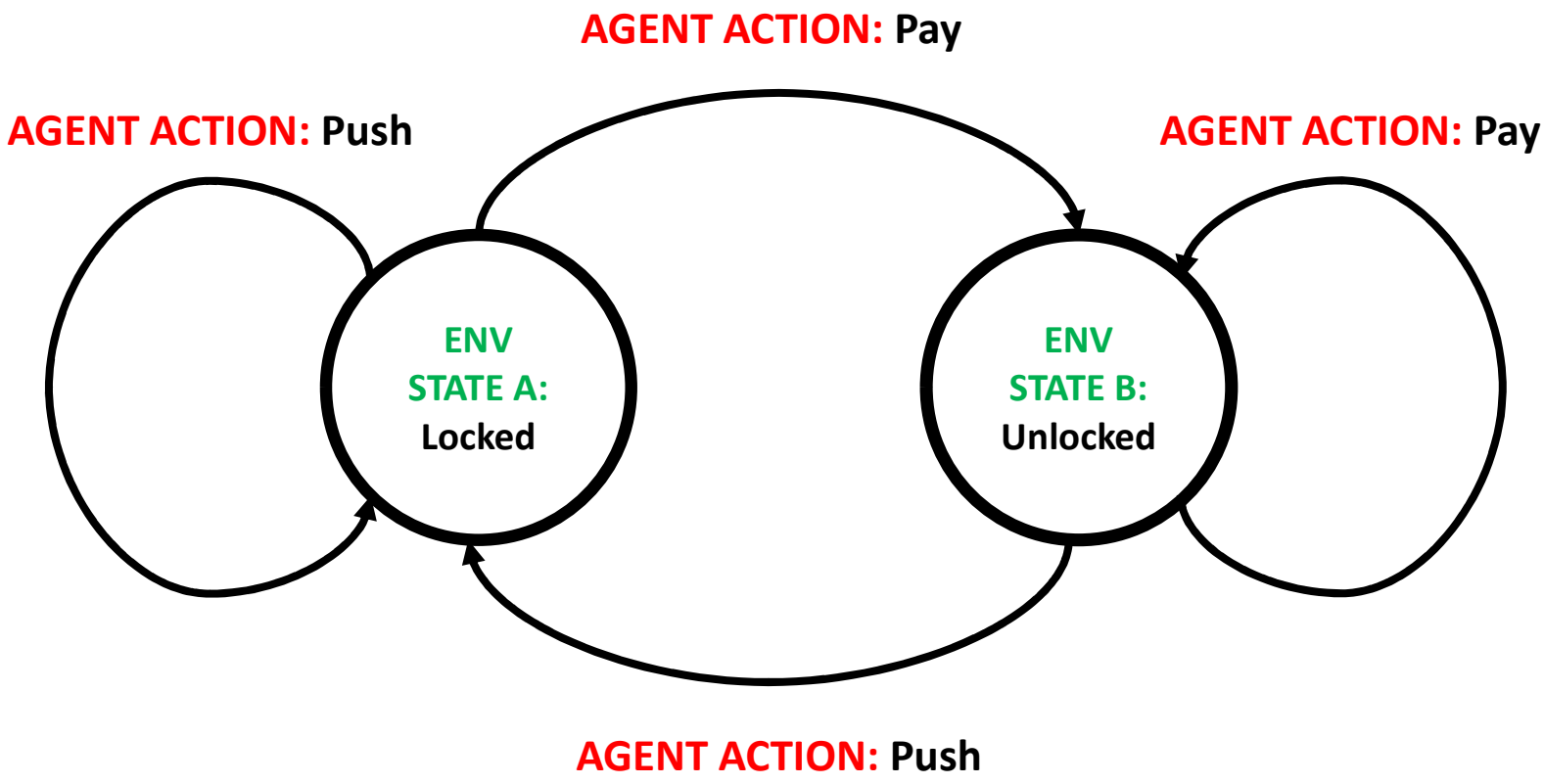
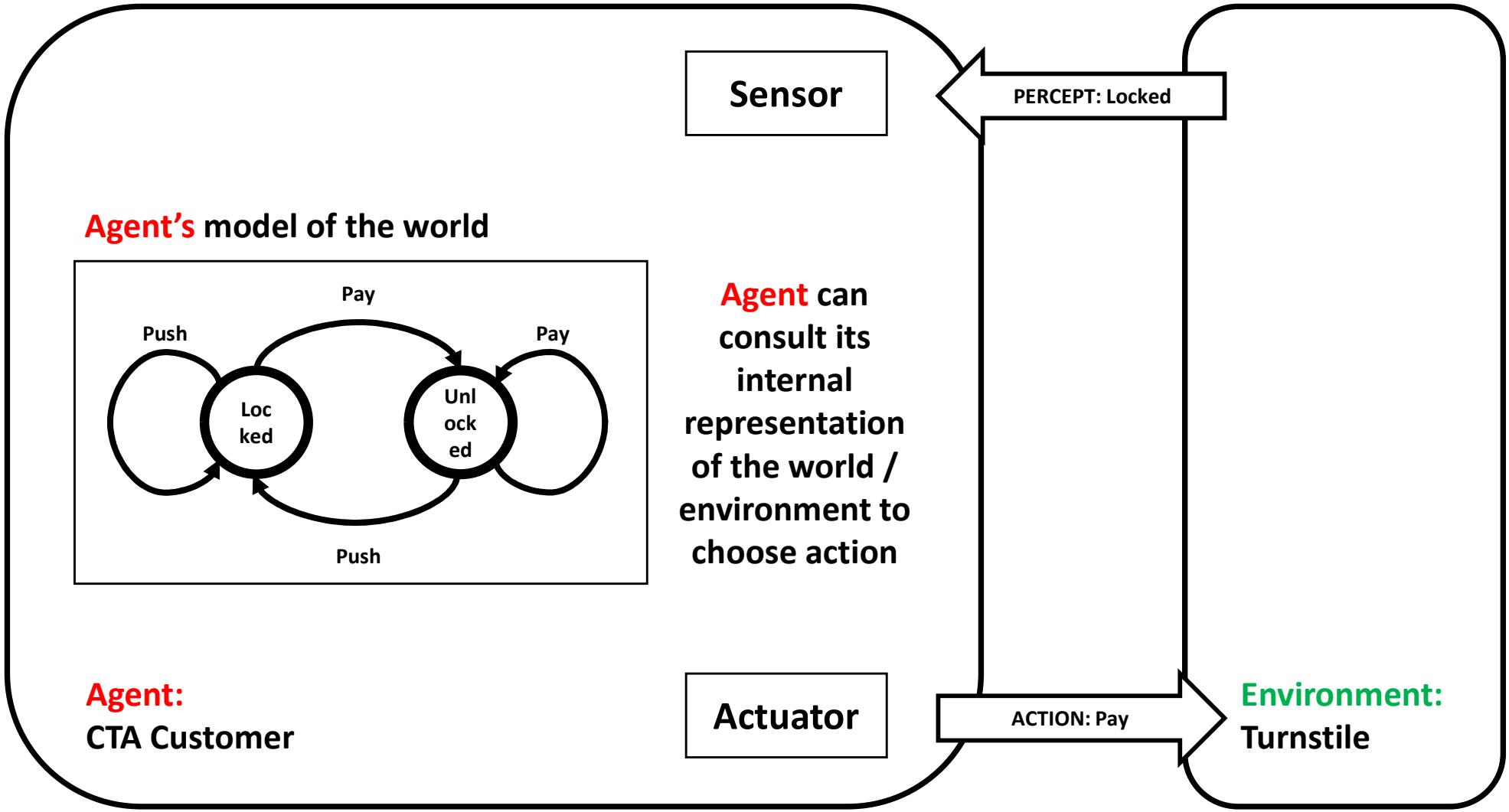


Image source: Wikipedia

Finite State Machine: A Turnstile

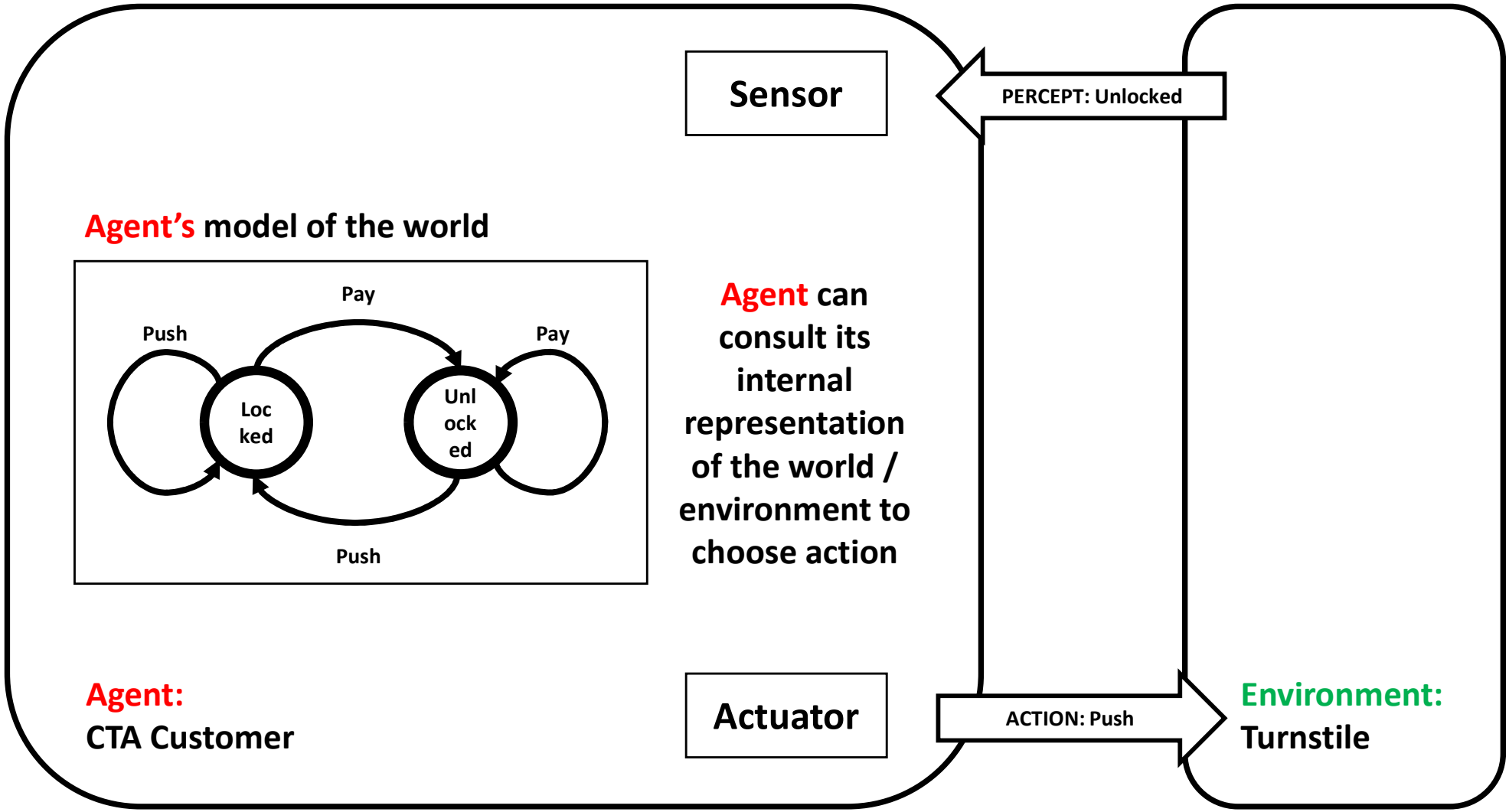


Model-based Reflex Agent Example



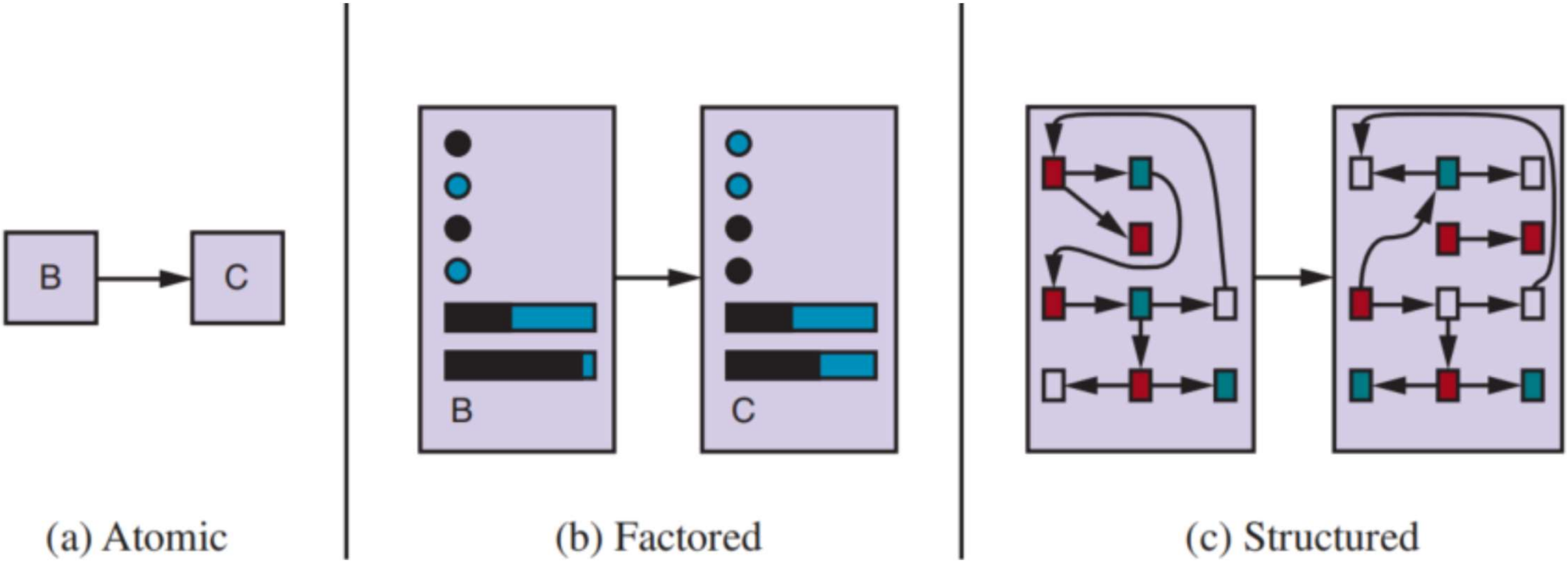
Note: This problem could be easily solved with a simple (without internal model) reflex agent.

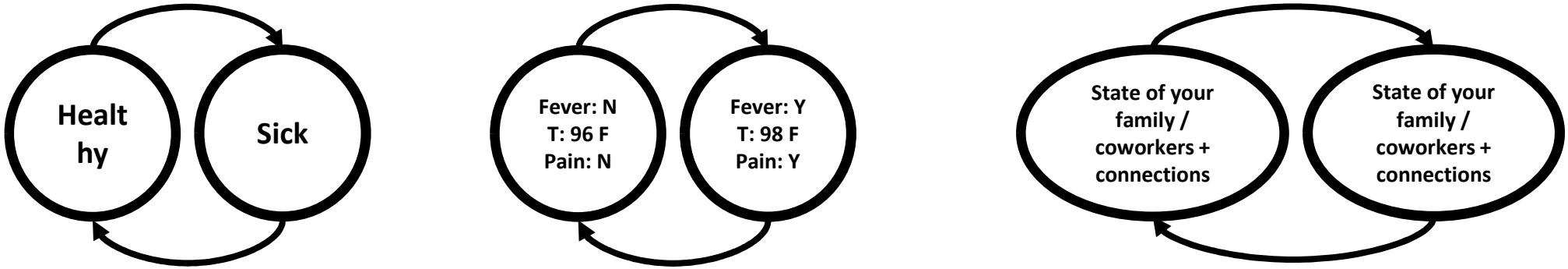
Model-based Reflex Agent Example



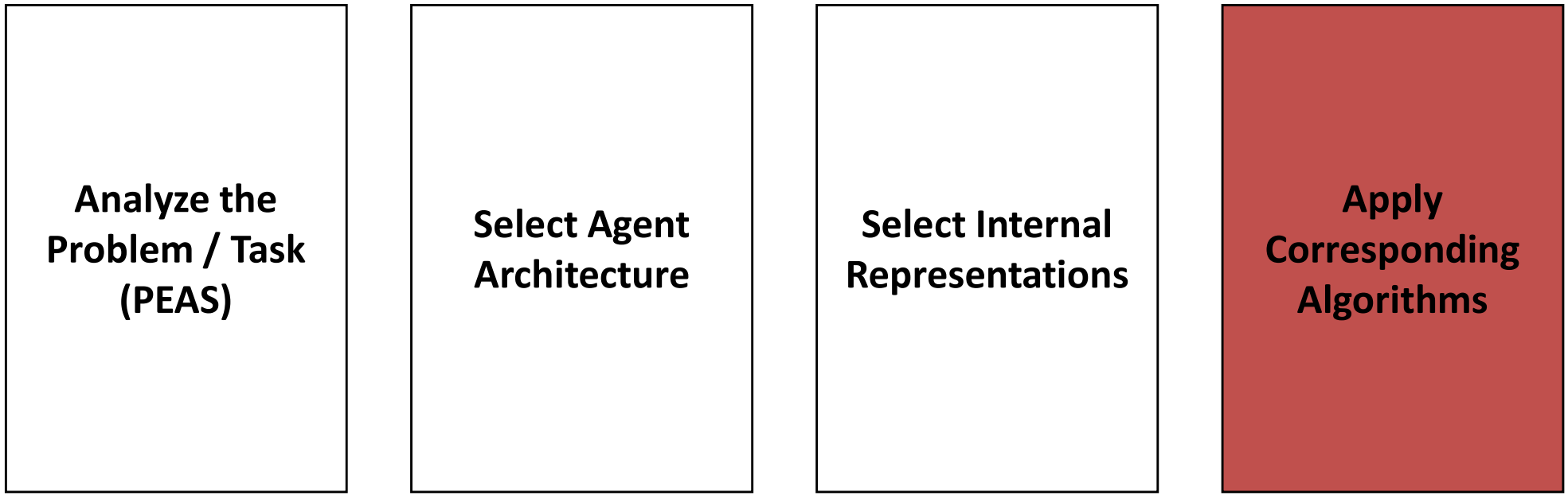
Note: This problem could be easily solved with a simple (without internal model) reflex agent.

Representations: Examples





Designing the Agent for the Task



BTW: How Would you Program it All?

Problem-Solving / Planning Agent

- **Context / Problem:**
  - **correct action is NOT immediately obvious**
  - **a plan (a sequence of actions leading to a goal) may be necessary**
- **Solution / Agent:**
  - **come up with a computational process that will search for that plan**
- **Planning Agent:**
  - **uses factored or structured representations of states**
  - **uses searching algorithms**

Planning: Environment Assumptions

Works with a "Simple Environment":

- Fully observable
- Single agent (for now -> it can be multiagent)
- Deterministic
- Static
- Episodic
- Discrete
- Known to the agent

Problem-Solving Process

- Goal formulation:
  - adopt a goal (think: desirable state)
  - a concrete goal should help you reduce the amount of searching
- Problem formulation:
  - an abstract representation of states and actions
- Search:
  - search for solutions within the abstract world model
- Execute actions in the solution

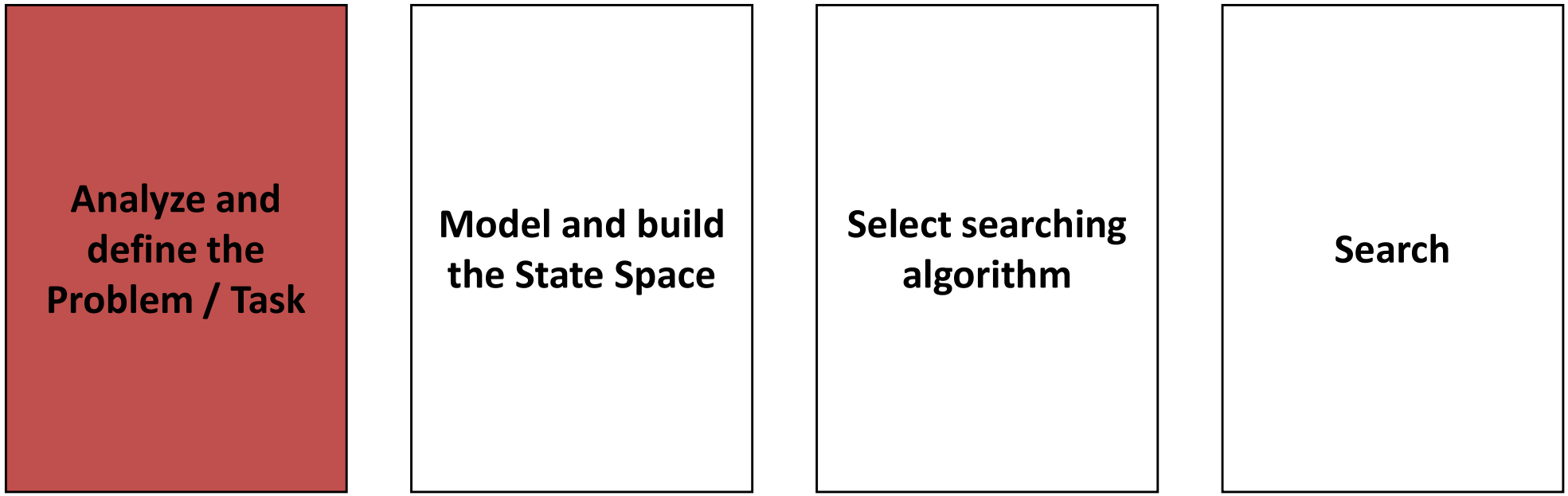
Planning: Environment Assumptions

Works with a "Simple Environment":

- **Fully observable**
- **Single agent (for now -> it can be multiagent)**
- **Deterministic**
- **Static**
- **Episodic**
- **Discrete**
- **Known to the agent**

Important and helpful:
Such assumptions **GUARANTEE** a **FIXED** sequence of actions as a solution
What does it mean?
You can execute the "plan" without worrying about incoming percepts (open-loop control)

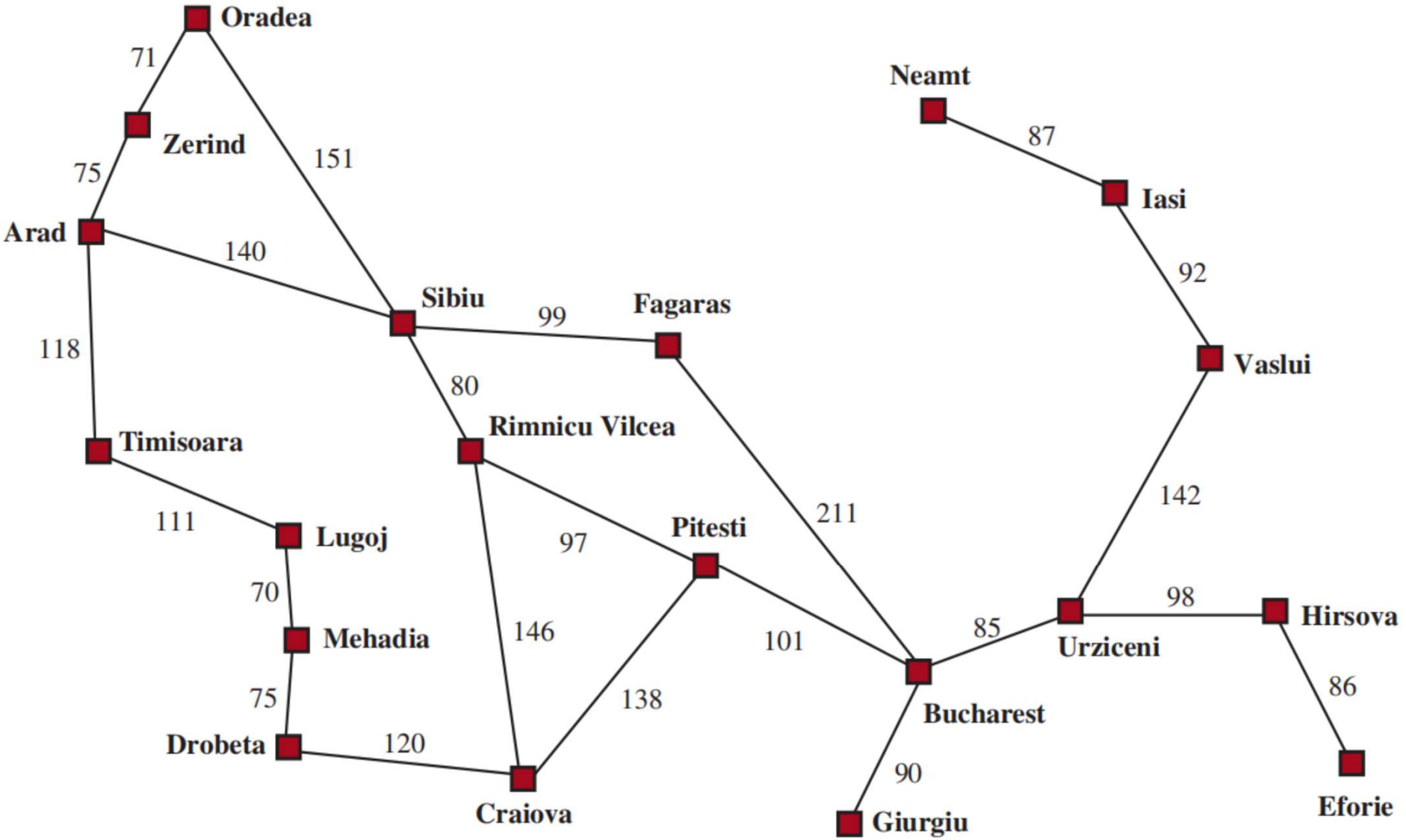
Designing the Searching Problem



Defining Search Problem

- Define a set of possible states: State Space
- Specify Initial State
- Specify Goal State(s) (there can be multiple)
- Define a FINITE set of possible Actions for EACH state in the State Space
- Come up with a Transition Model which describes what each action does
- Specify the Action Cost Function: a function that gives the cost of applying action *a* in state *s*

Sample Problem: Romanian Roadtrip

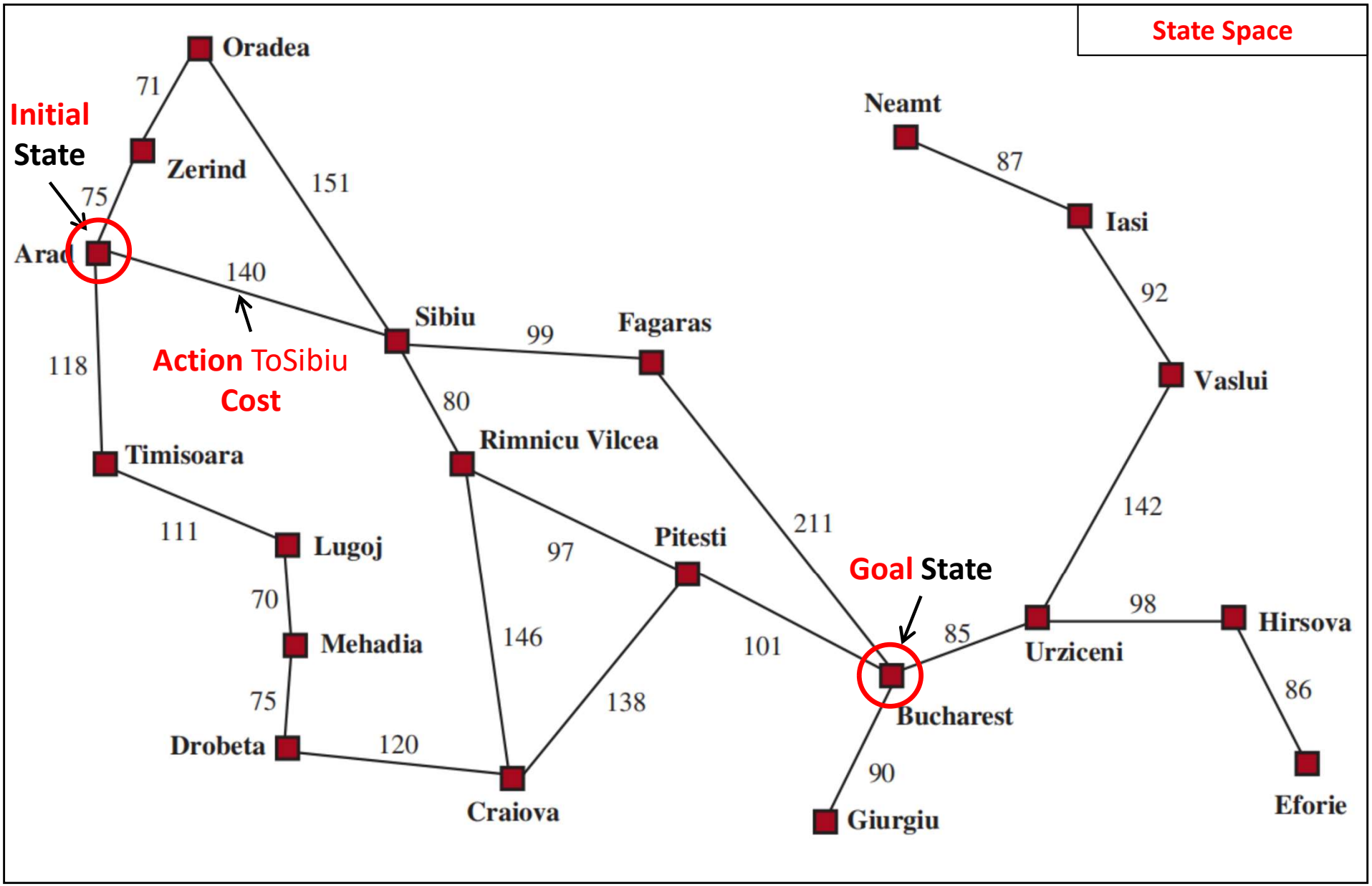


Problem: Get from Arad to Bucharest efficiently (for example: quickly or cheaply).

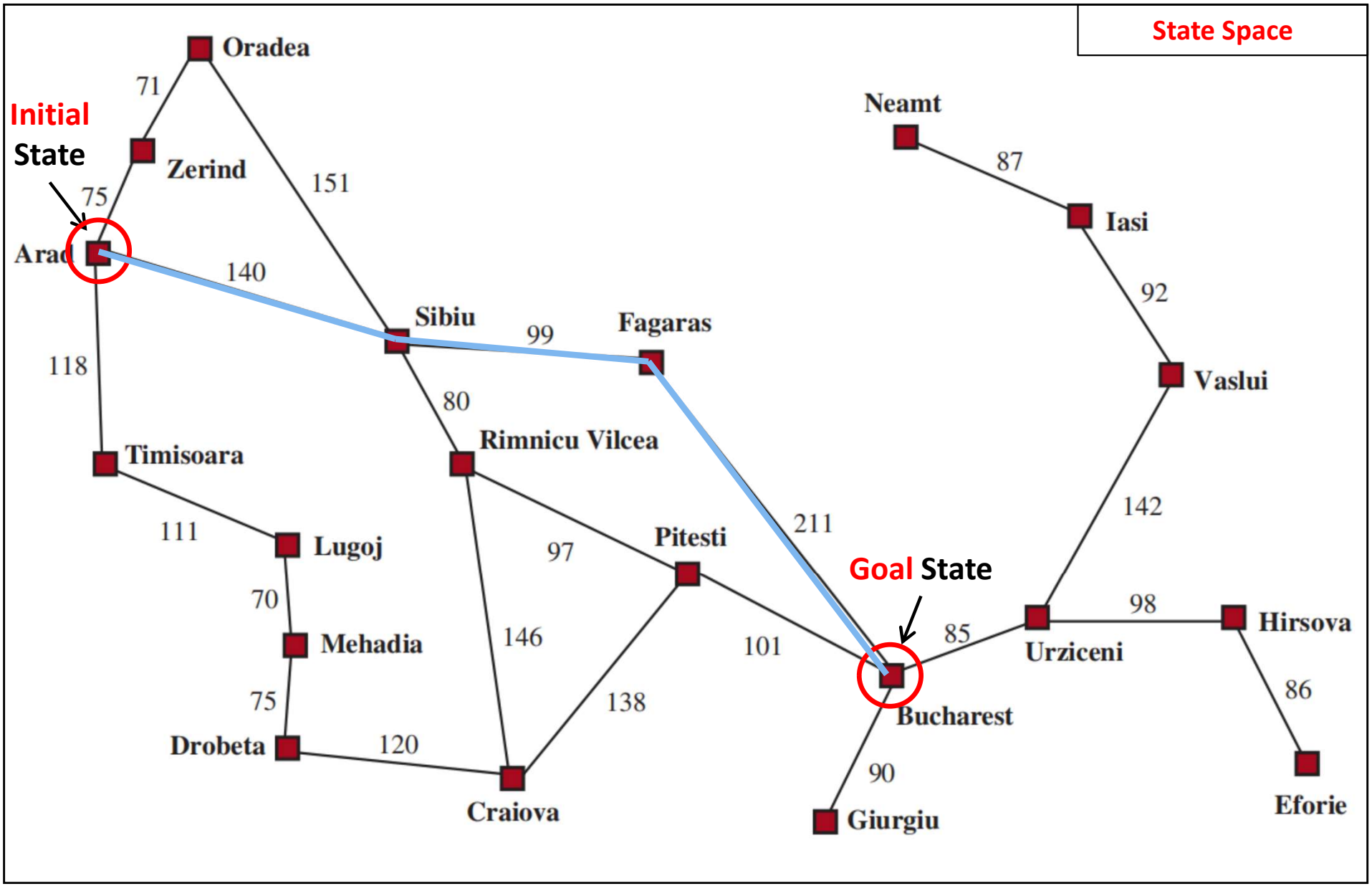
Search Problem: Romanian Roadtrip

- State Space: a map of Romania
- Initial State: Arad
- Goal State: Bucharest
- Actions:
  - for example: ACTIONS(Arad) = {ToSibiu,ToTimisoara,ToZerind}
- Transition Model:
  - for example: RESULT(Arad, ToZerind) = Zerind
- Action Cost Function [ActionCost($S_{current}$, a, S_{next})]
  - for example: ActionCost(Arad, ToSibiu, Sibiu) = 140

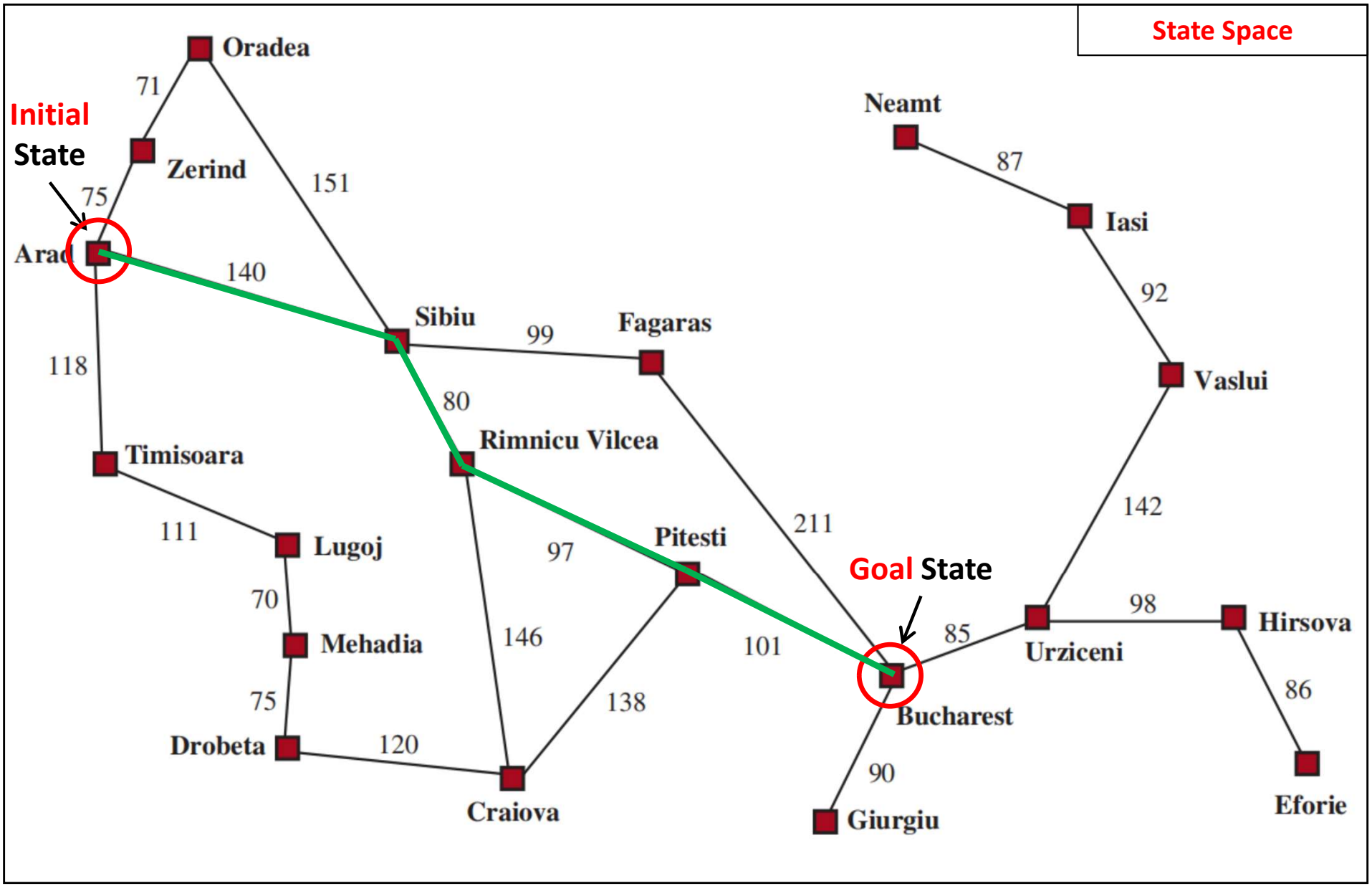
Sample Problem: Romanian Roadtrip



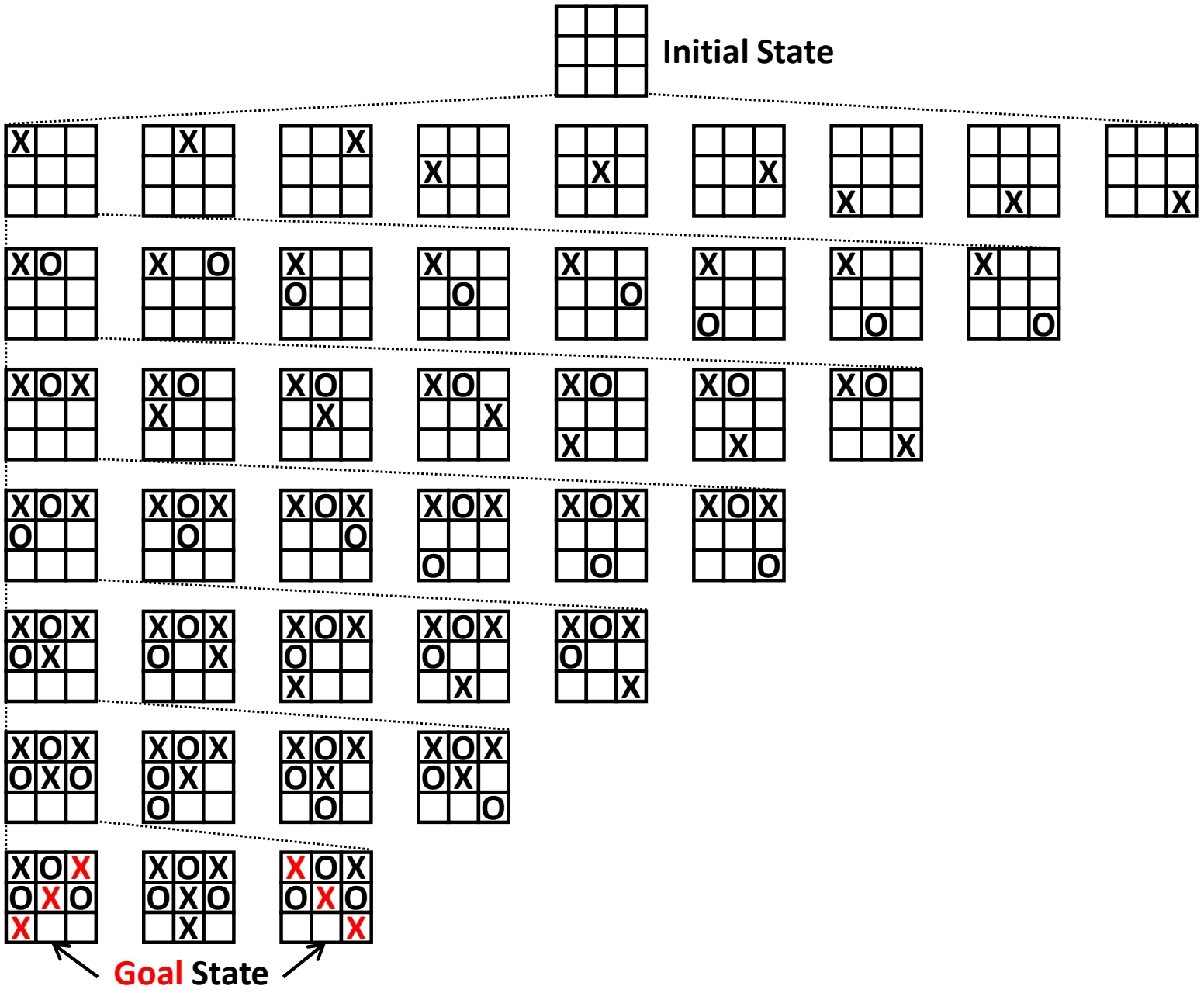
Romanian Roadtrip: Potential Solution



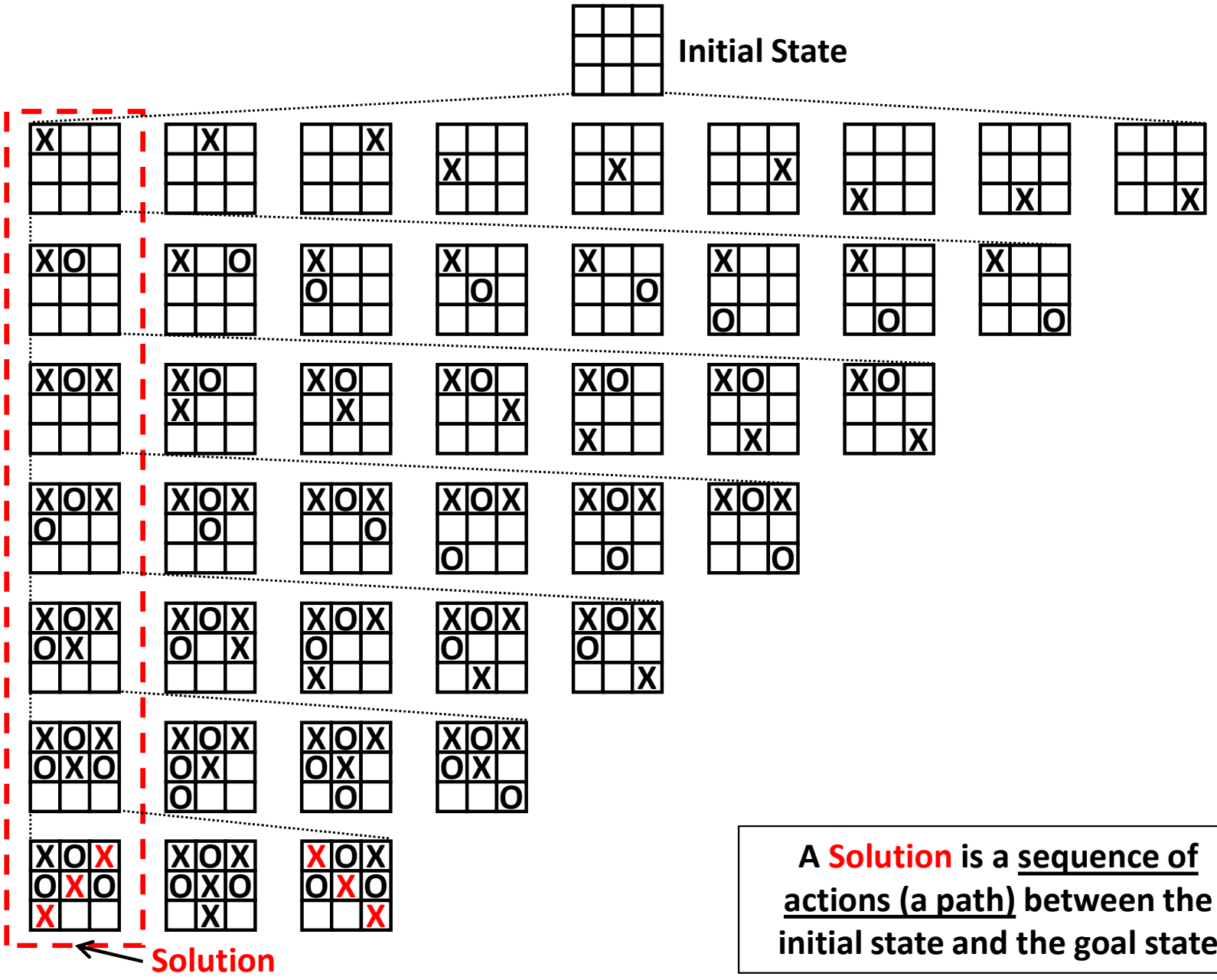
Romanian Roadtrip: Potential Solution



Tic Tac Toe: (Partial) State Space



Tic Tac Toe: Solution

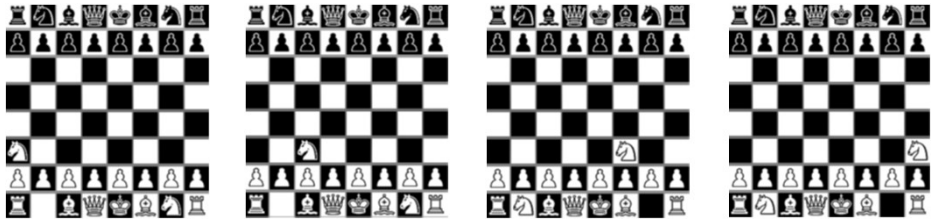


A Solution is a sequence of actions (a path) between the initial state and the goal state

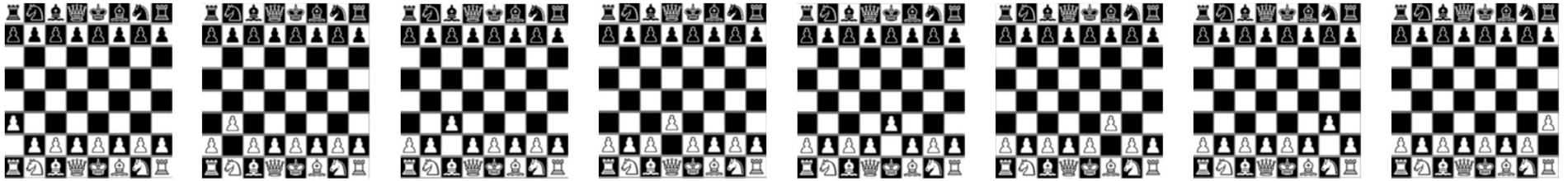
Chess: (First Move) State Space

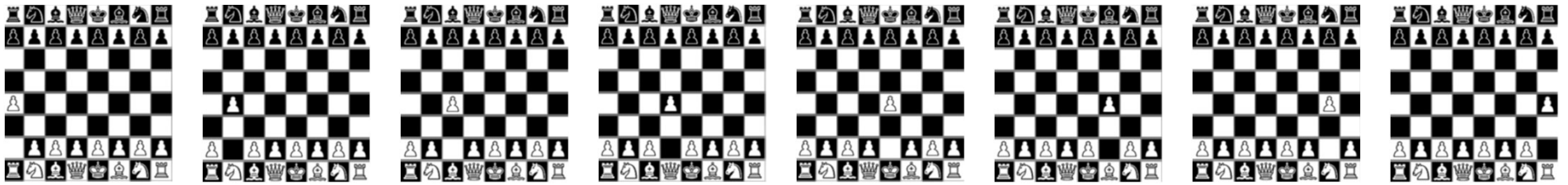
Initial State

Move

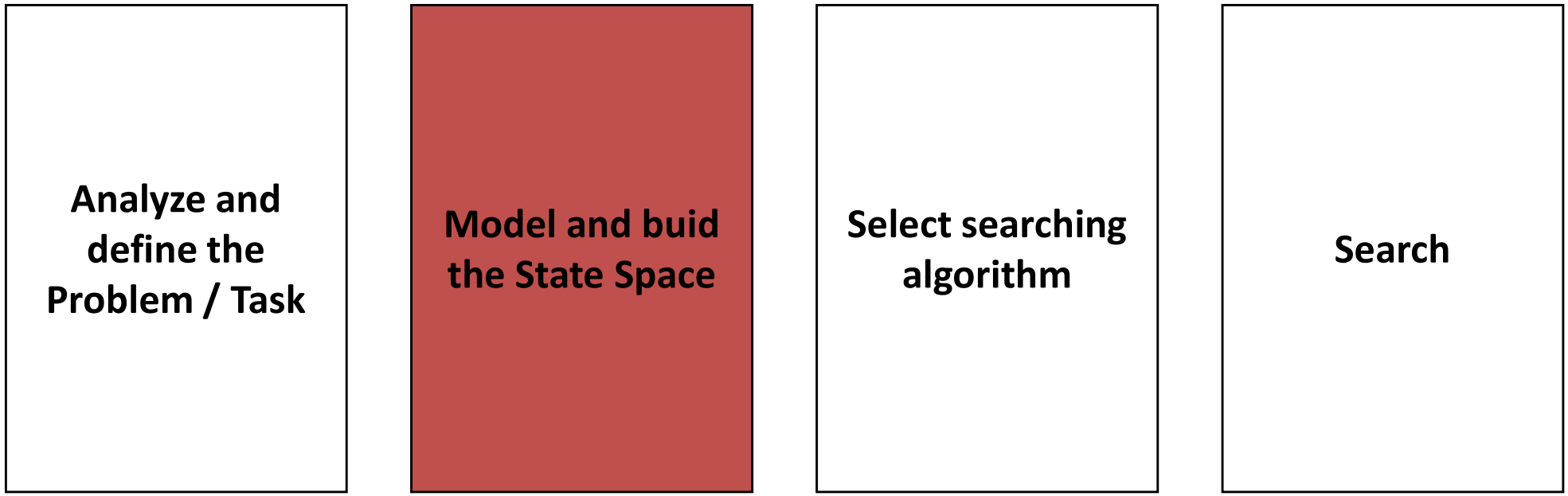


20 Possible legal first moves:
16 pawn moves
4 knight moves

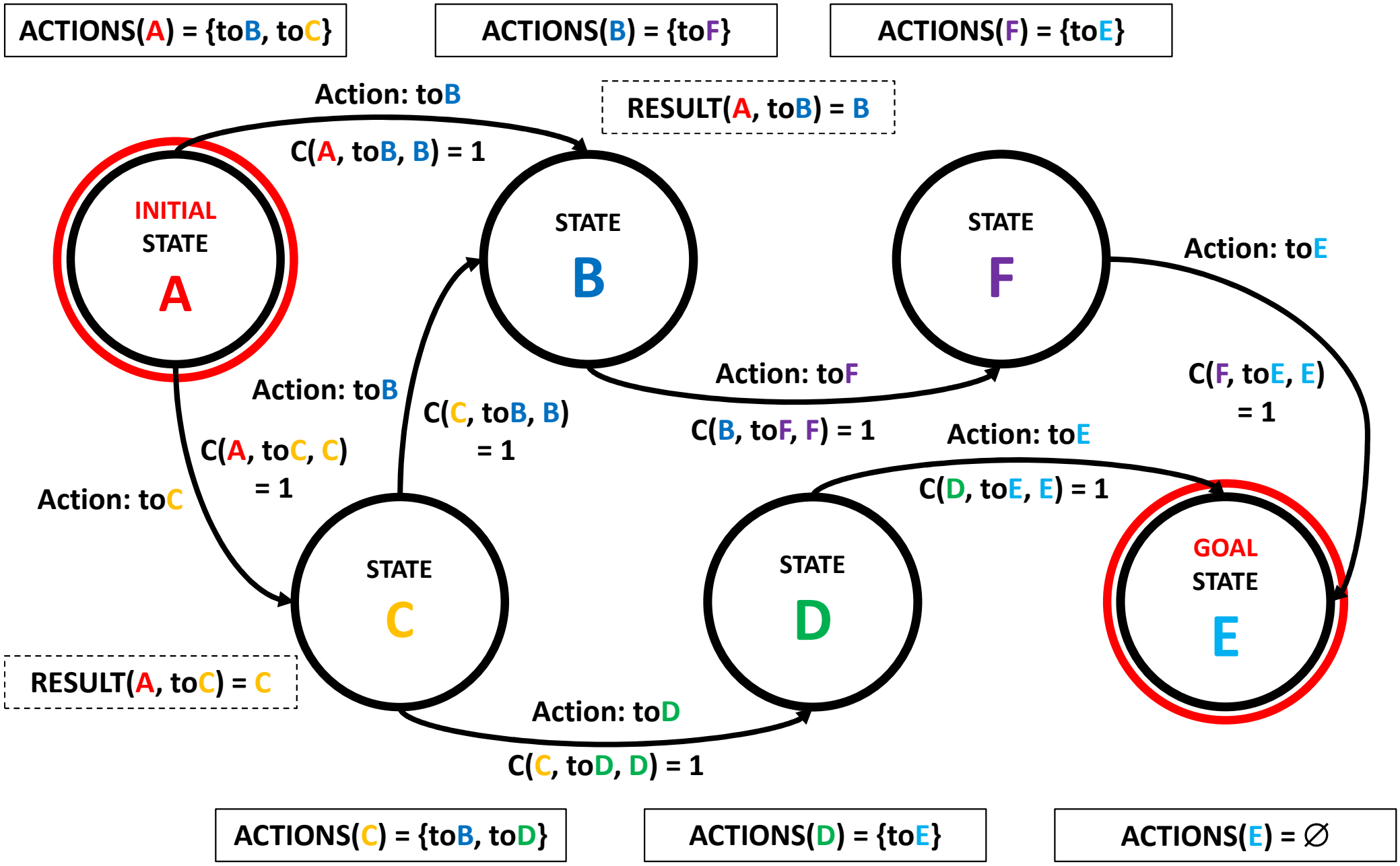




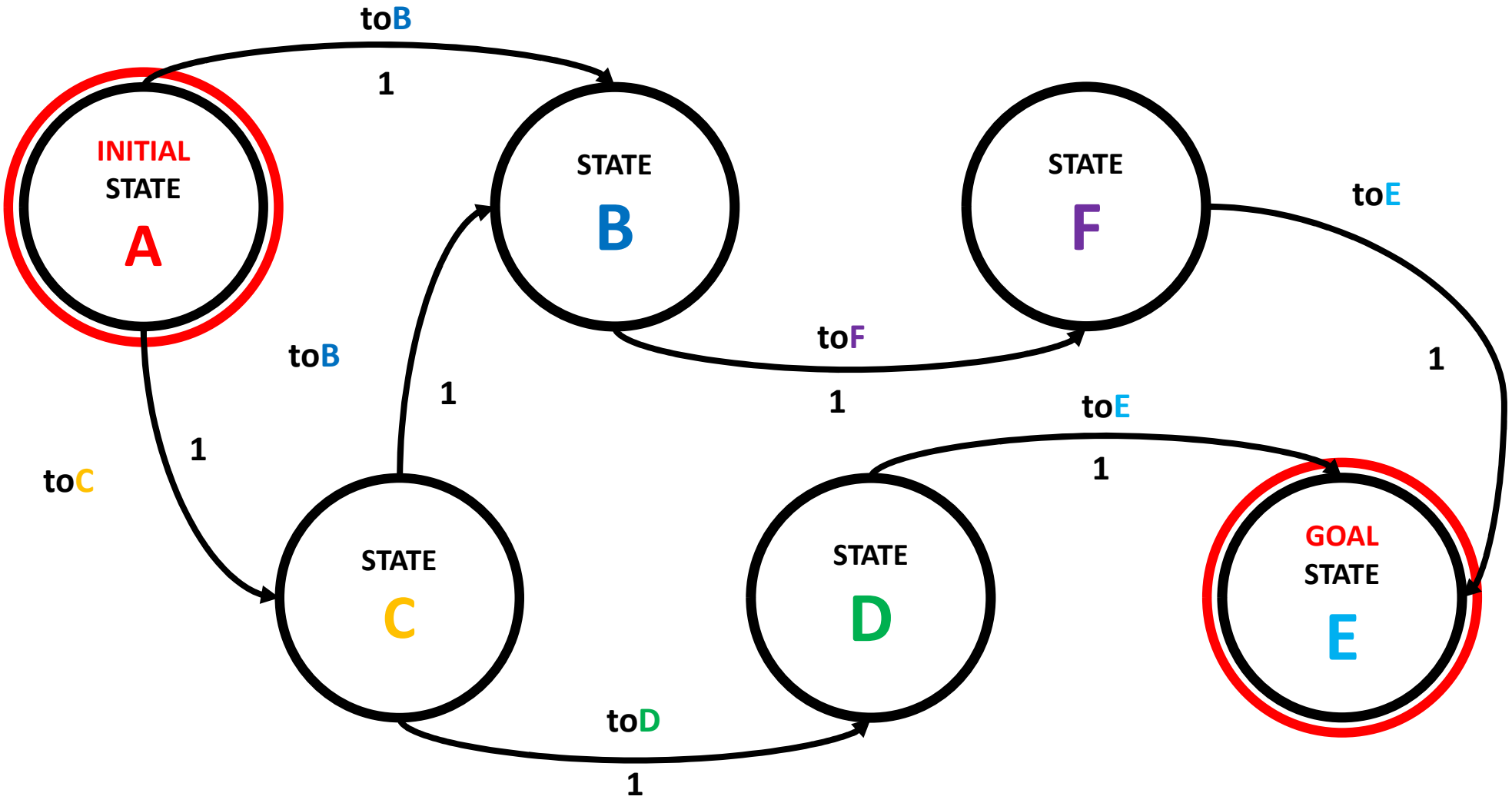
Designing the Searching Problem



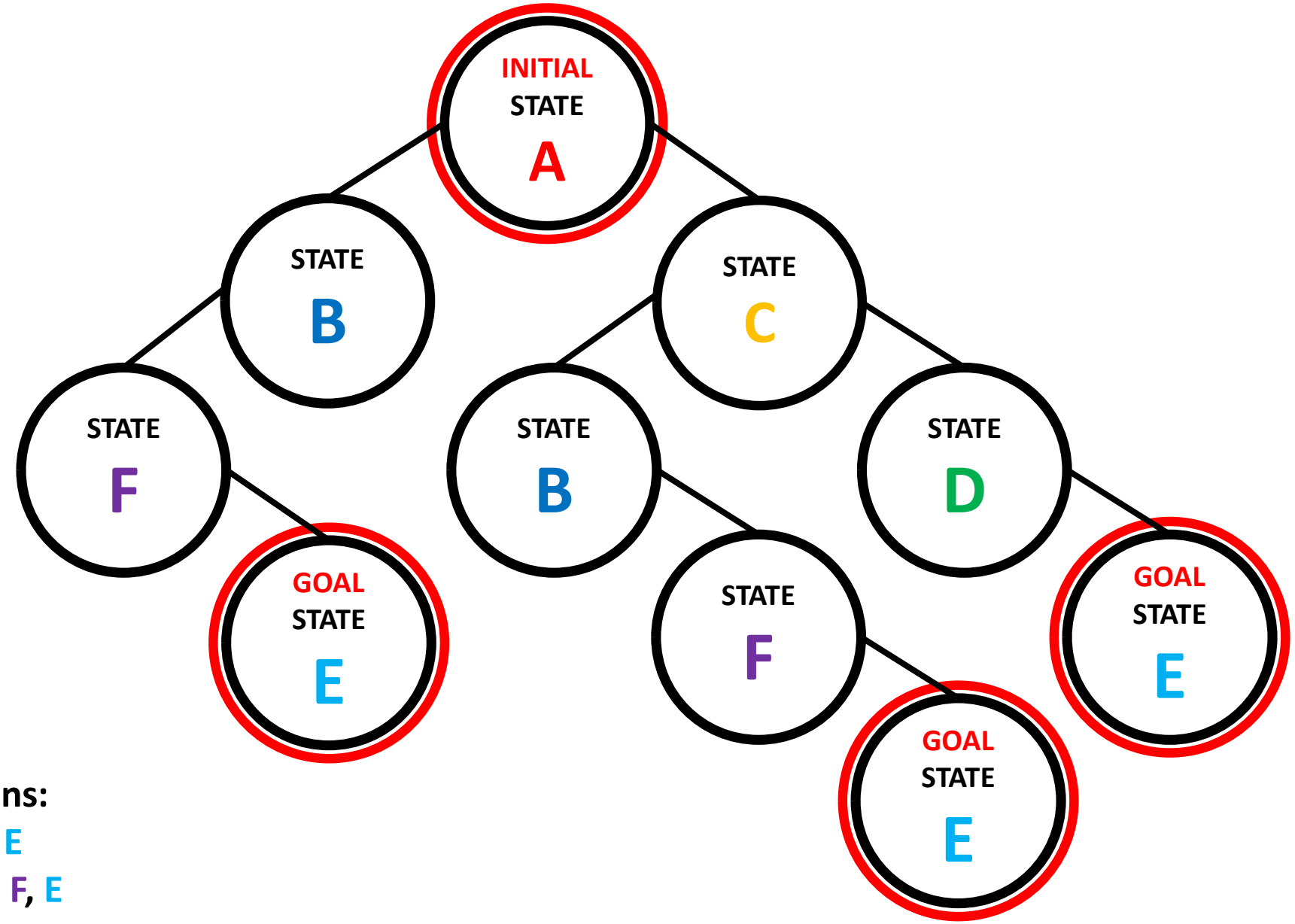
State Space Model: A Graph



State Space Model: A Graph



Searching State Space: Search Tree



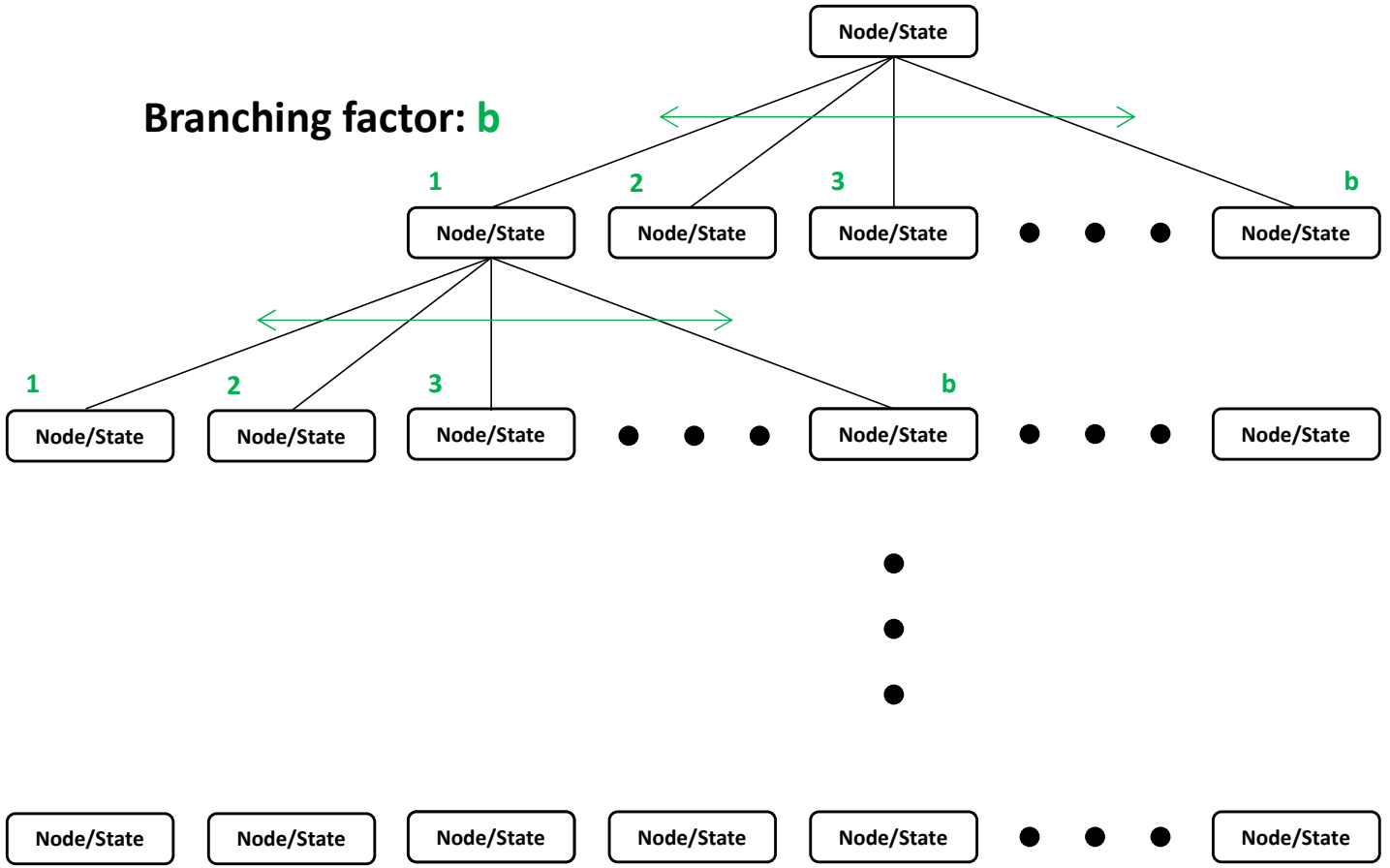
Solutions:

A, B, F, E

A, C, B, F, E

A, C, D, E

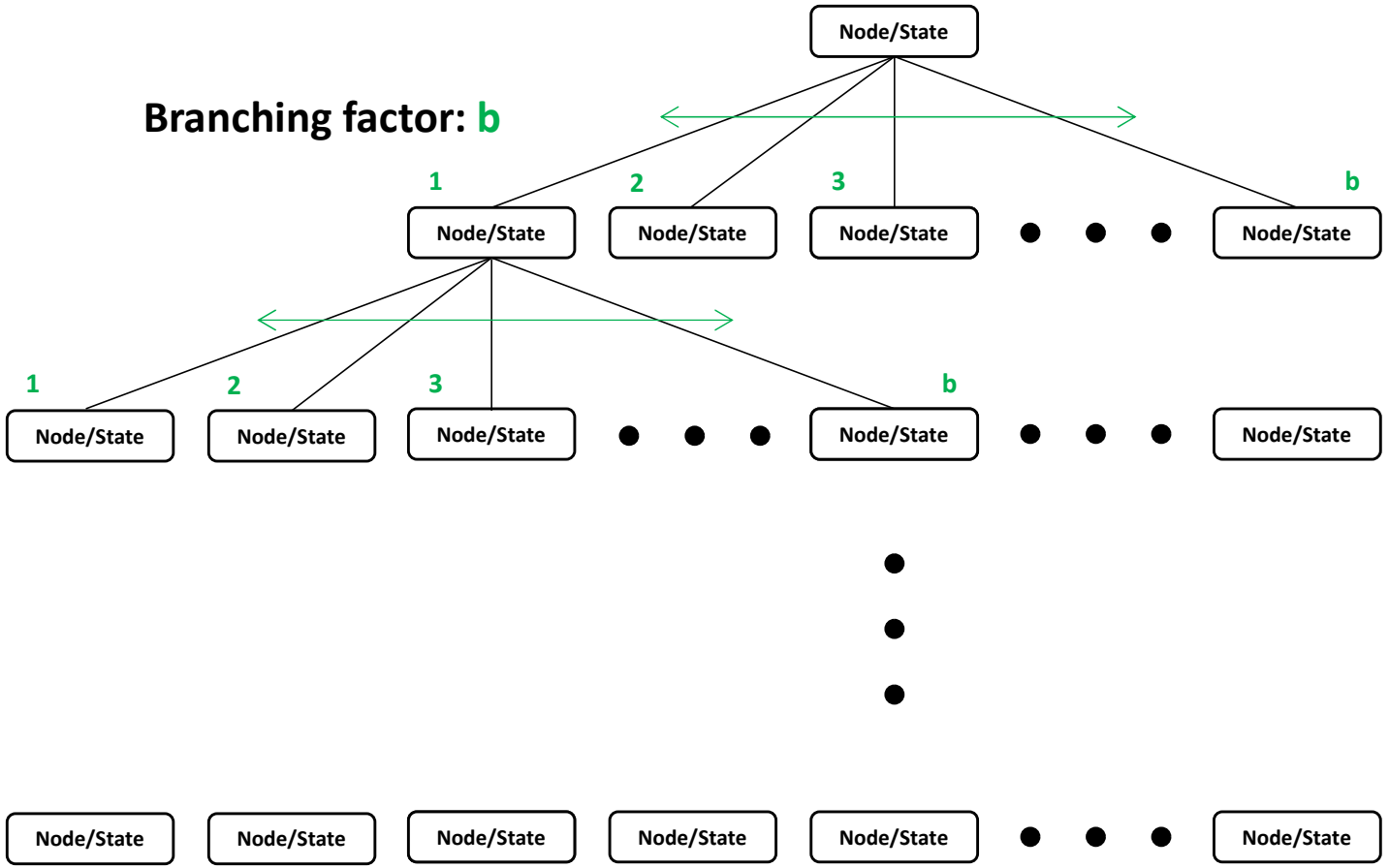
Search Tree Challenges: Size



Total number of nodes / states: $1 + b + b^2 + b^3 + \ldots + b^d \rightarrow O(b^d)$

Quickly becomes unmanageable and impossible to search with brute force!

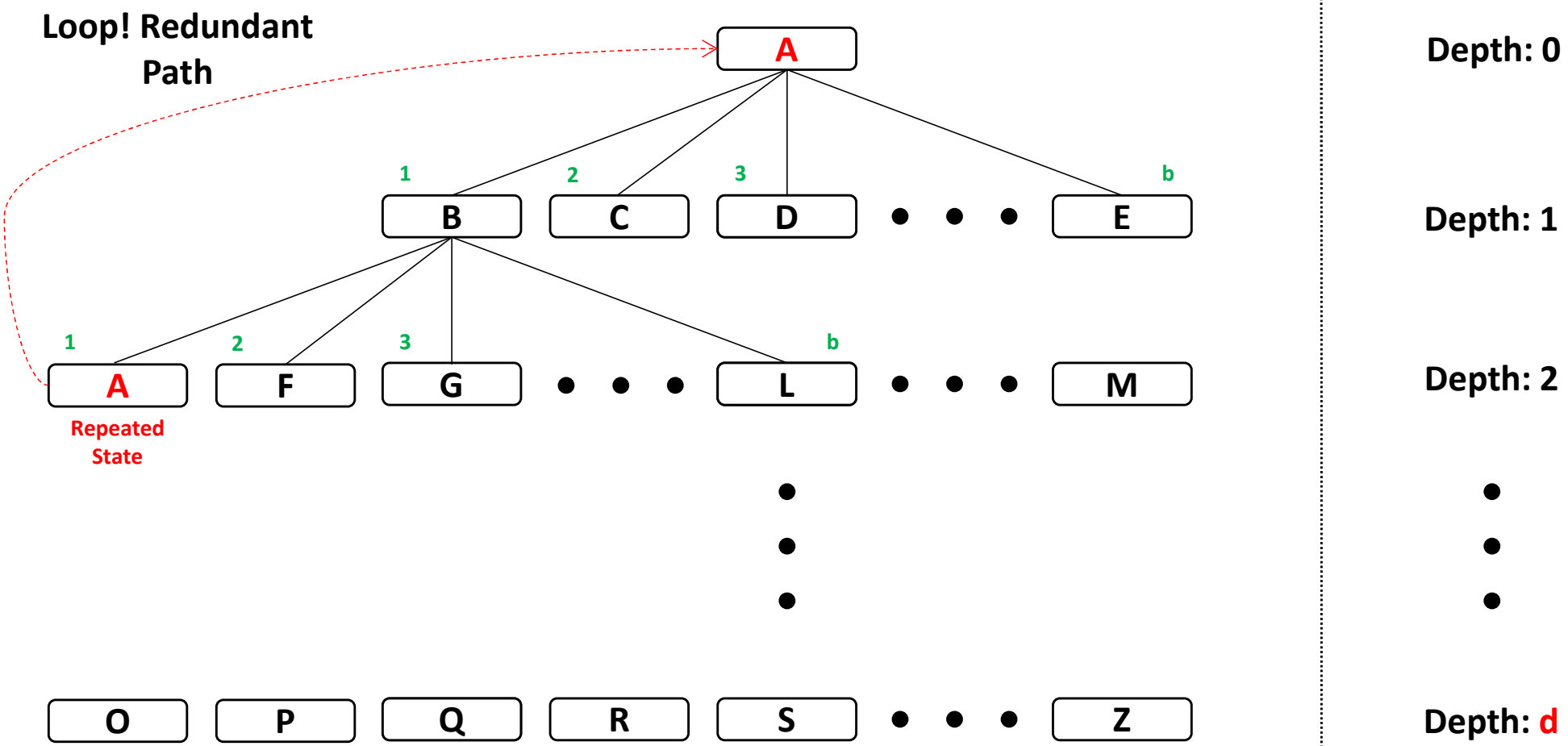
Search Tree Challenges: Infiniteness



Unmanageable and impossible to search with brute force!

Memory and time use grows quickly!

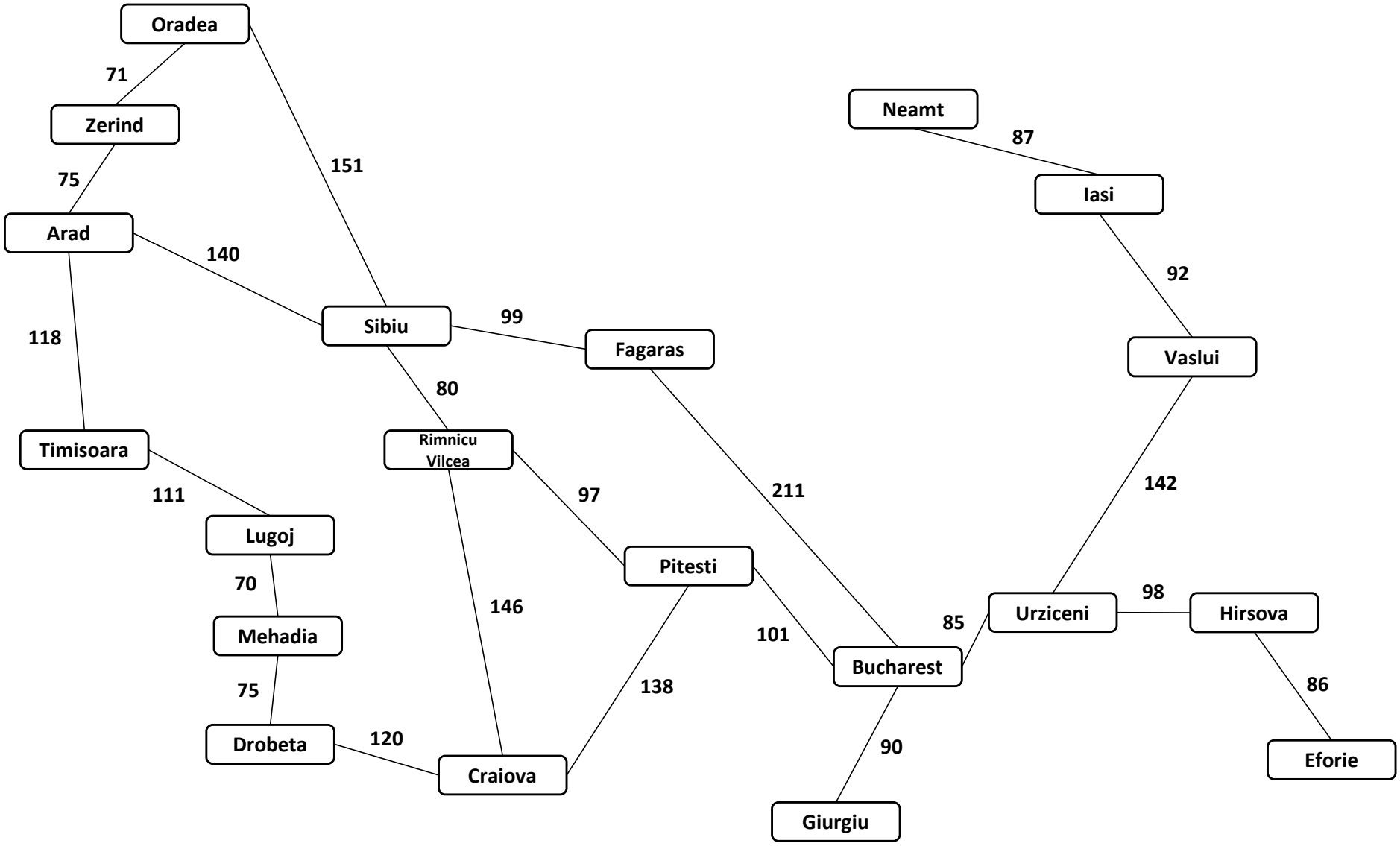
Search Tree Challenges: Loops



This would lead to an infinite state sequence repetition if not handled!

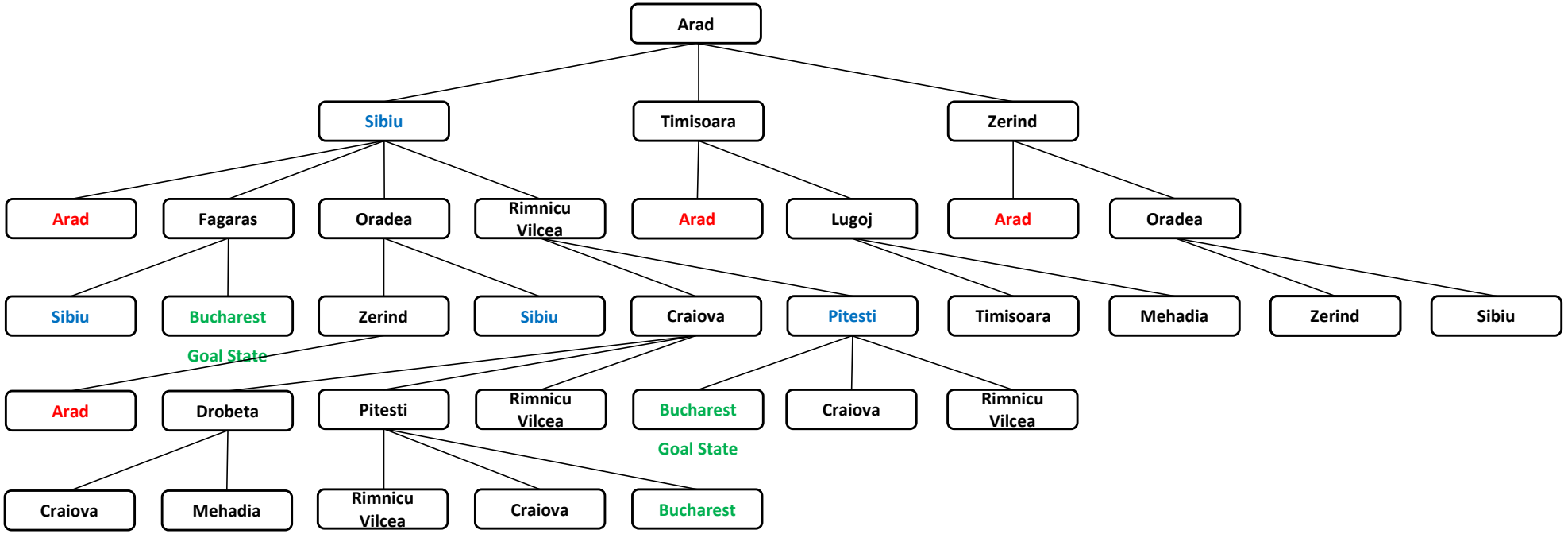
Memory and time use grows quickly!

Sample Problem: Romanian Roadtrip



Problem: Get from Arad to Bucharest efficiently (for example: quickly or cheaply).

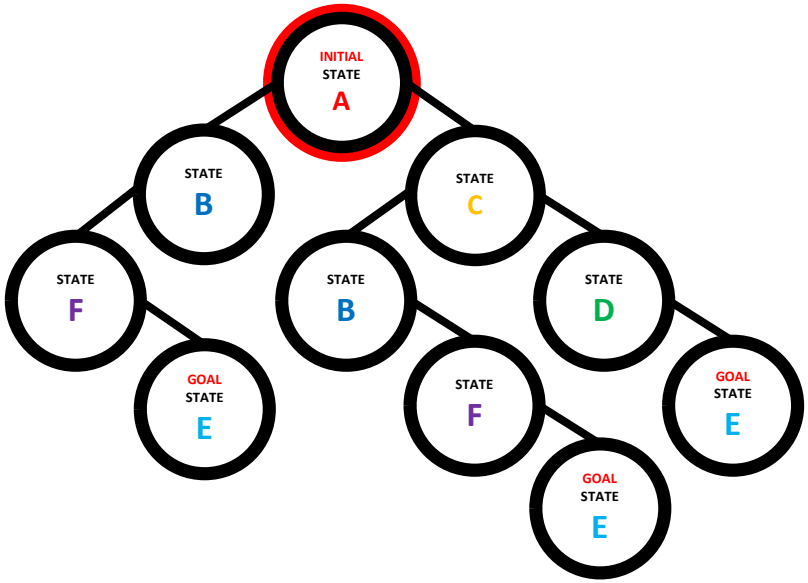
Romanian Roadtrip as a Tree



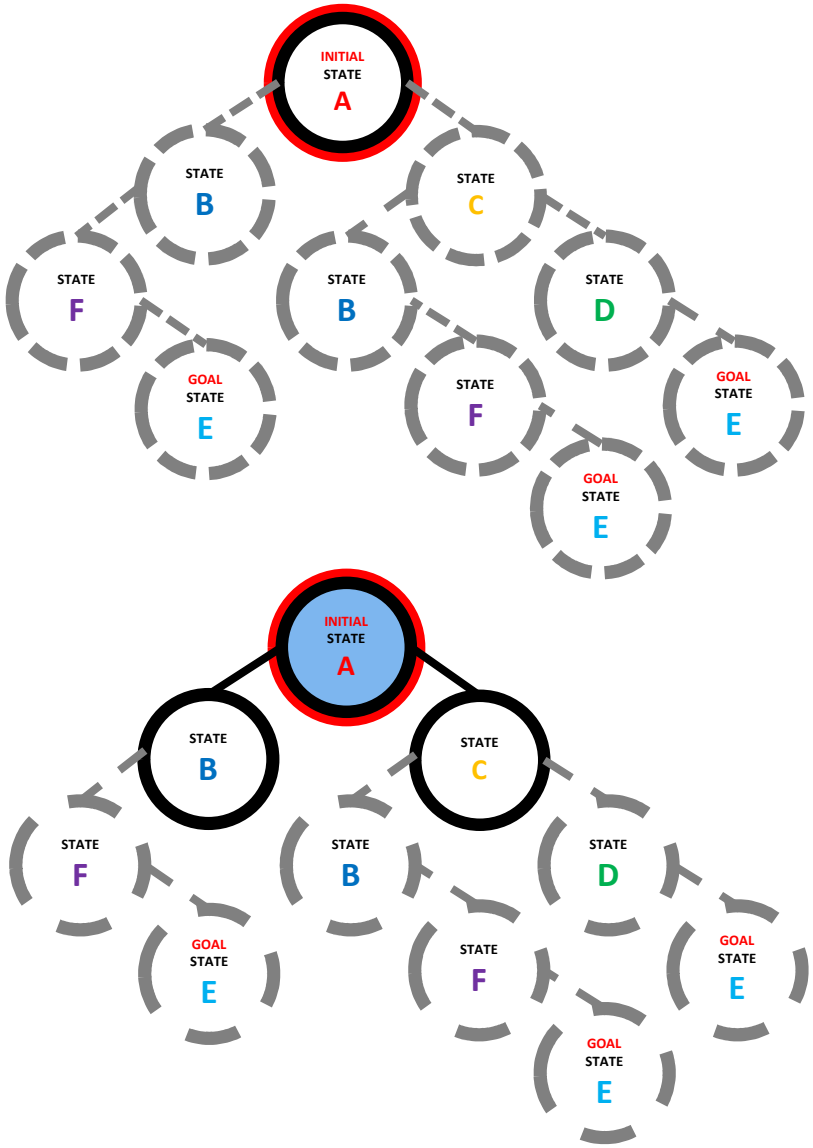
INCOMPLETE! I need to redraw it in smarter way

Search Tree: Implementations

Build entire search tree



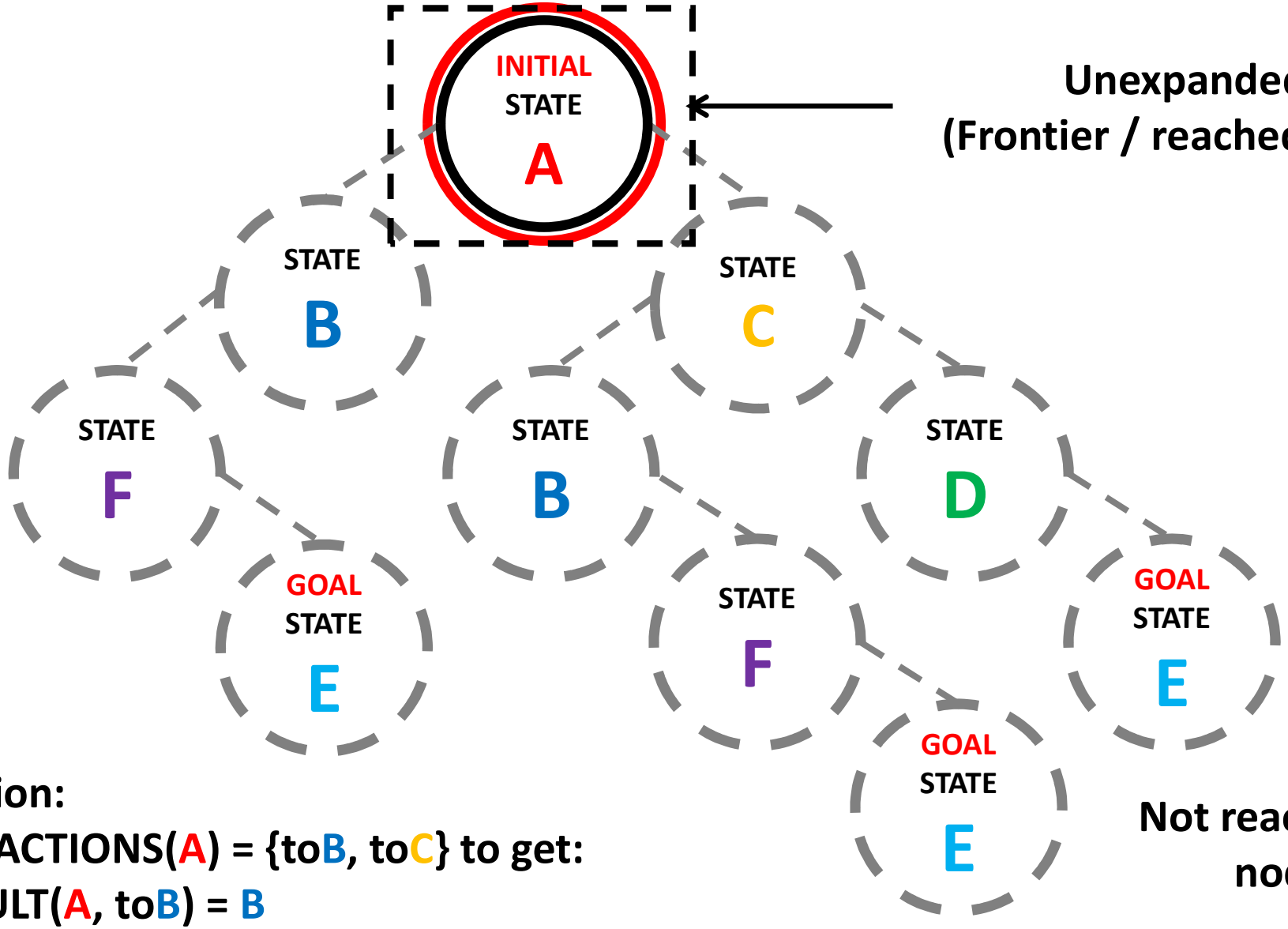
Expand/generate nodes as you go



Challenges:

- **memory requirements**
- **impossible for infinite number of states**

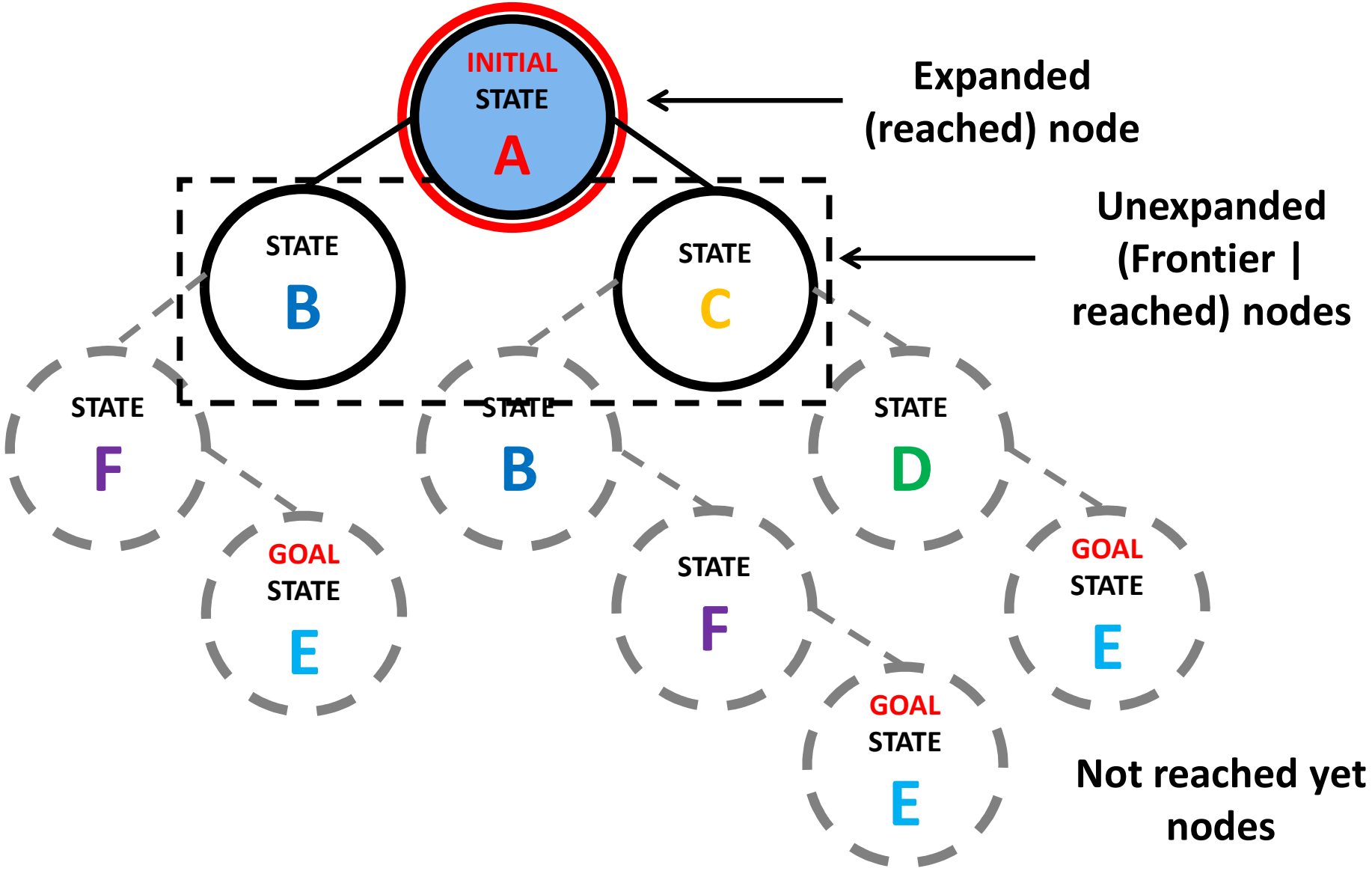
Search Tree: Node Expansion



Expansion:

- Use ACTIONS(A) = {toB, toC} to get:
- RESULT(A, toB) = B
- RESULT(A, toC) = C

Search Tree: Node Expansion

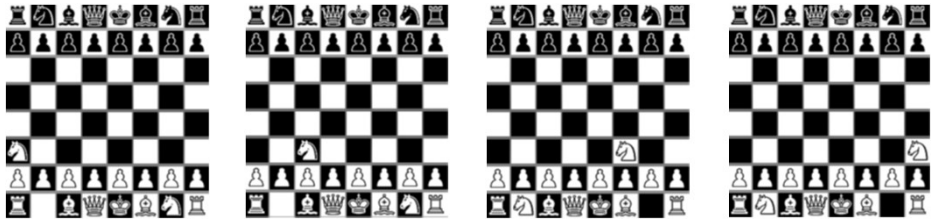


Chess: State Node Expansion

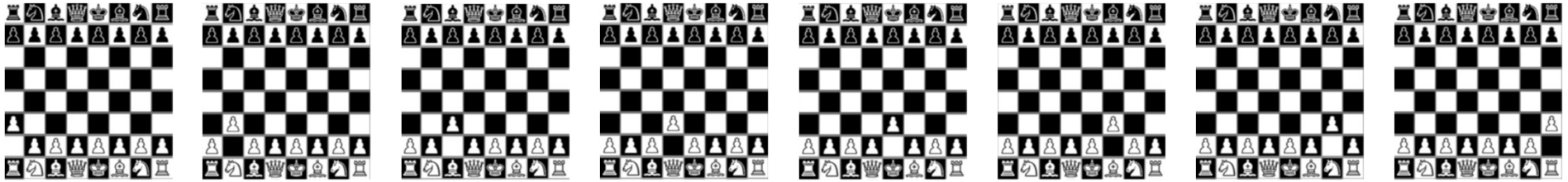
Initial State

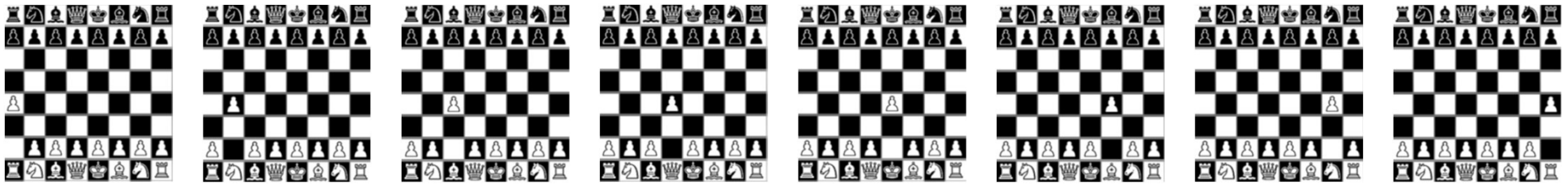
Move

Use game rules to generate subsequent possible game tree states / nodes!

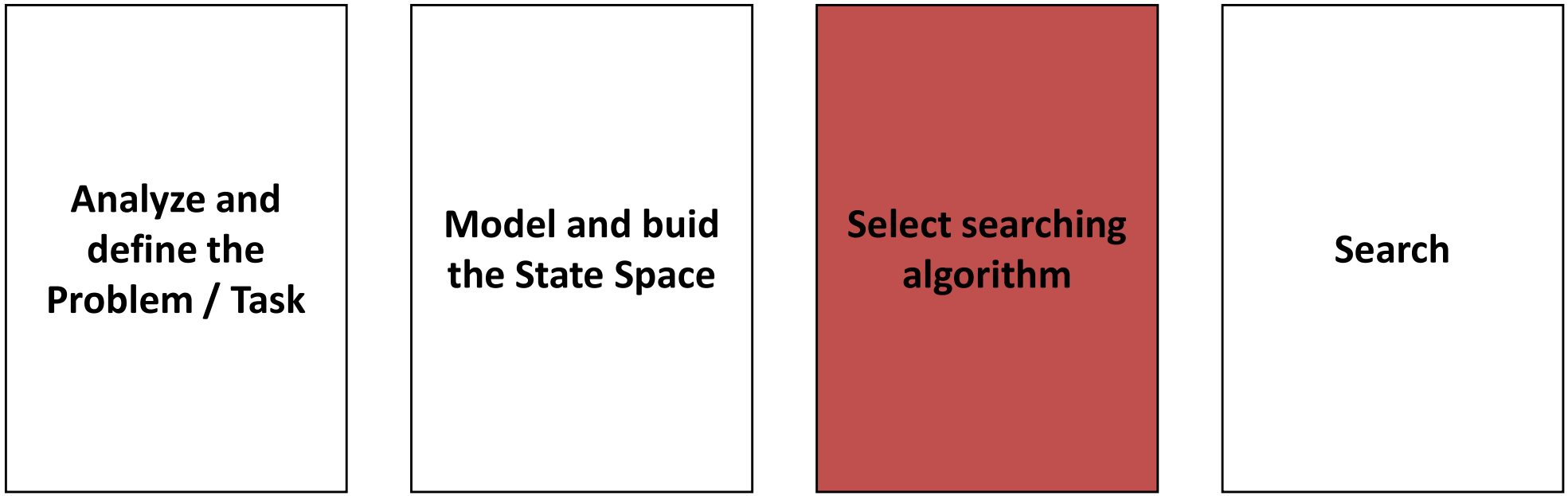


20 Possible legal first moves:
16 pawn moves
4 knight moves





Designing the Searching Problem

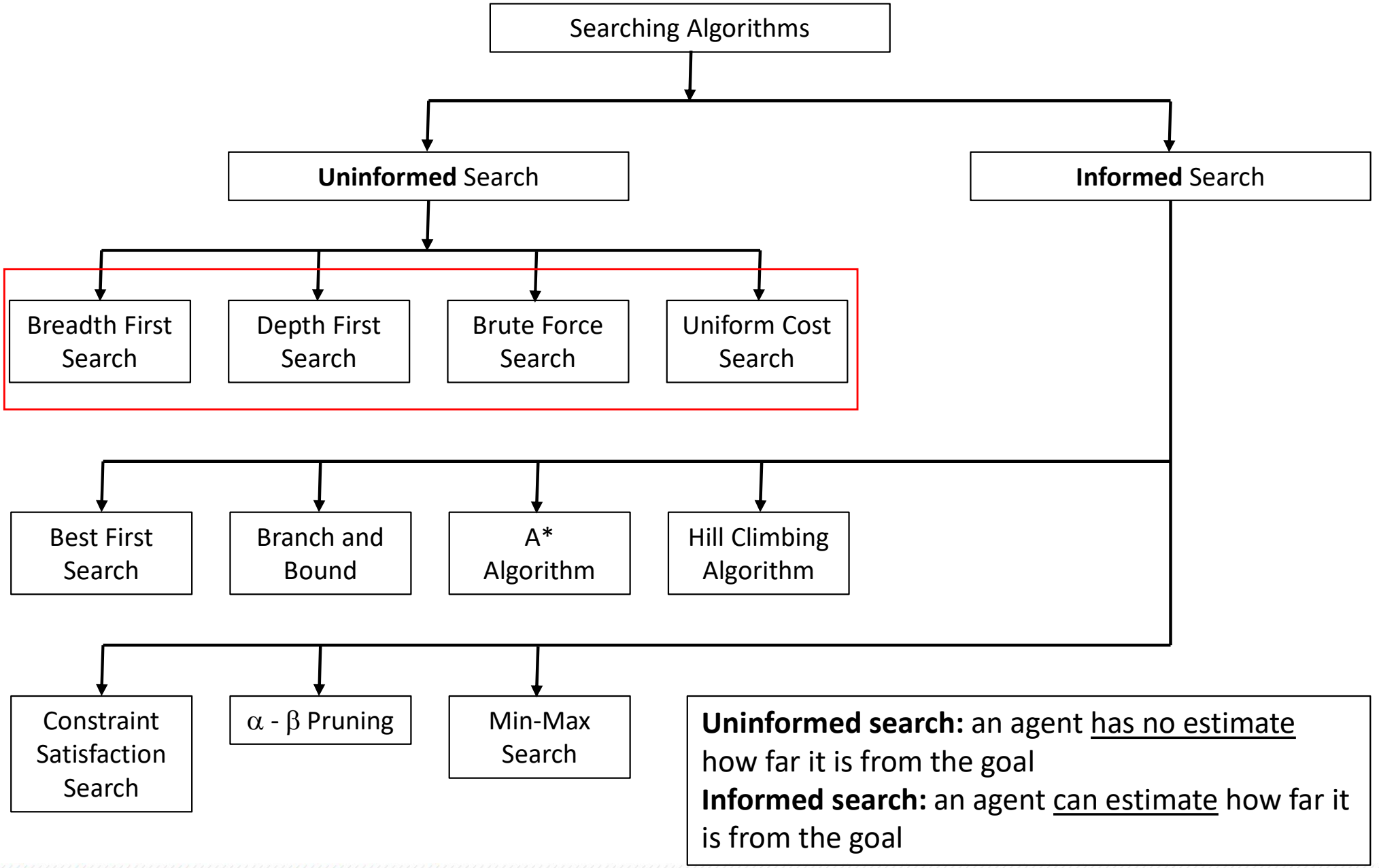


Measuring Searching Performance

Search algorithms can be evaluated in four ways:

- **Completeness: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?**
- **Cost optimality: Does it find a solution with the lowest path cost of all solutions?**
- **Time complexity: How long does it take to find a solution? (in seconds, actions, states, etc.)**
- **Space complexity: How much memory is needed to perform the search?**

Selected Searching Algorithms

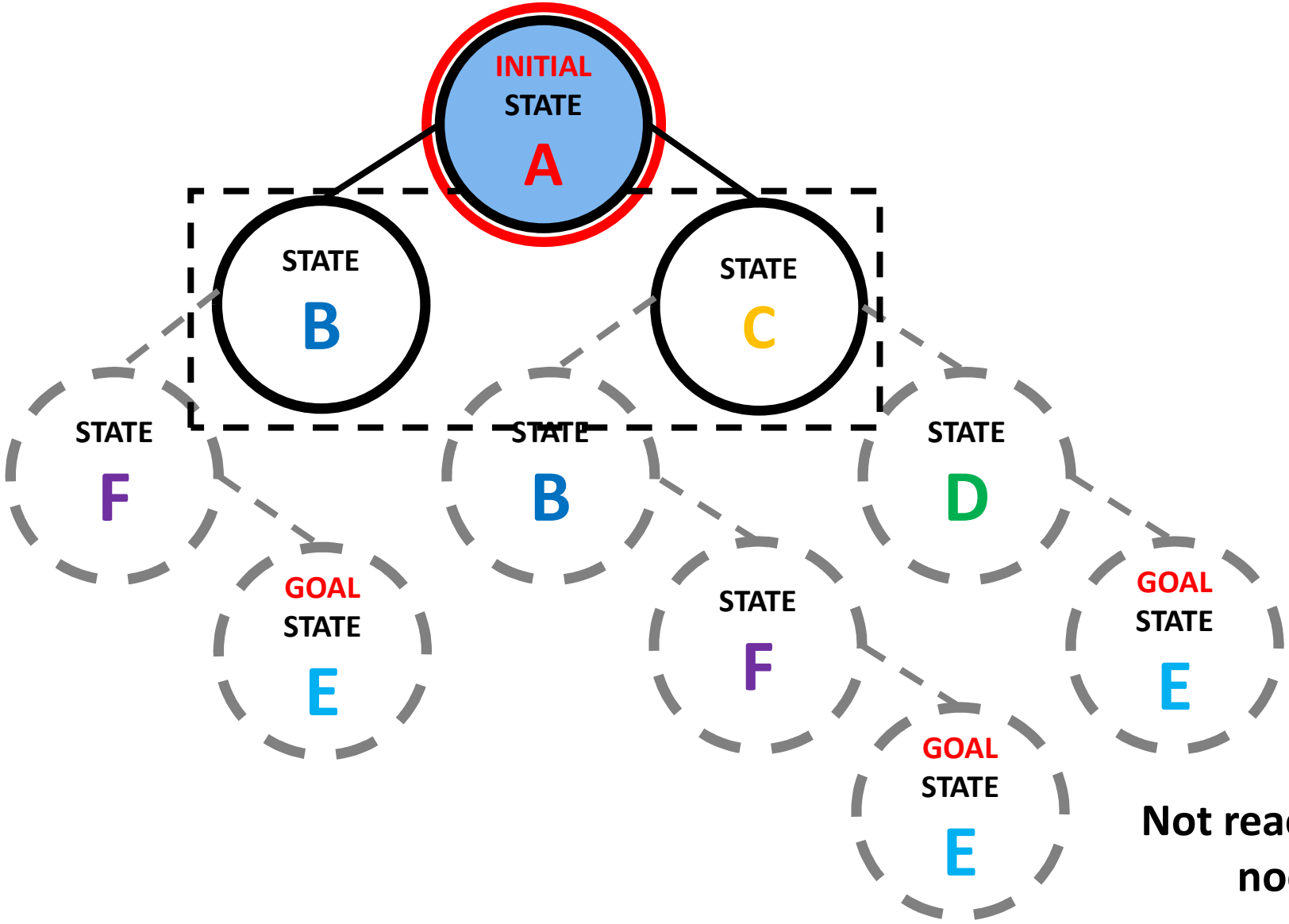


Uninformed search: an agent has no estimate how far it is from the goal
Informed search: an agent can estimate how far it is from the goal

Uninformed Searching

- Breadth First Search (BFS):
  - Will find a solution with a minimal number of actions
  - Large memory requirement
  - Only relatively small problem instances are tractable
- Depth First Search:
  - May NOT find a solution with a minimal number of actions
  - Requires less memory than BFS (for tree search
  - Backtracking (one child / successor generated at a time)
- Brute Force Search: depends on the approach -> bad
- Uniform Cost Search: minimize solution / path cost

Expansion: Which Node to Expand?



Evaluation function

Calculate / obtain:

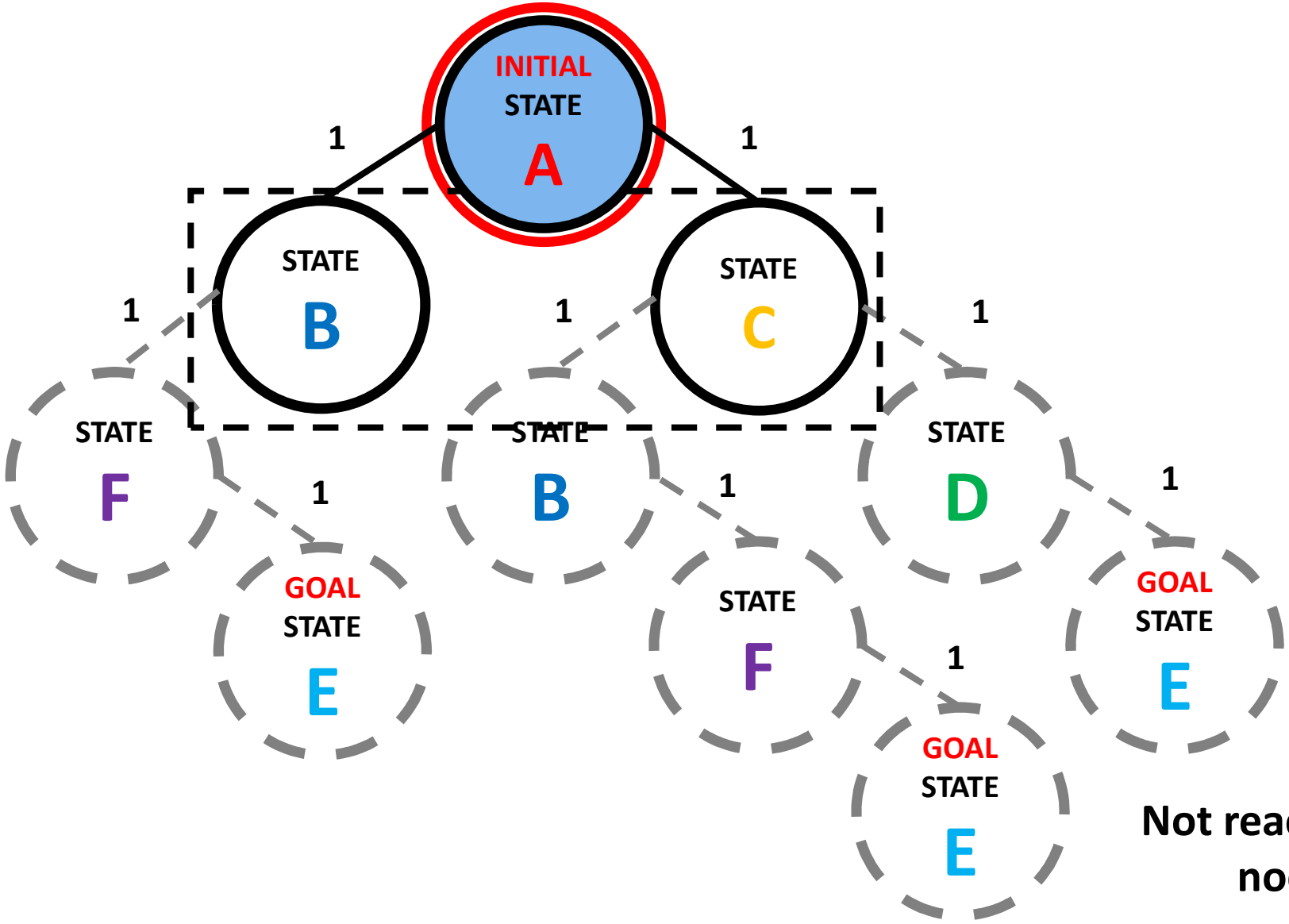
f(n) = f(State n)

f(n) = f(relevant information about State n)

A state n with minimum f(n) should be chosen for expansion

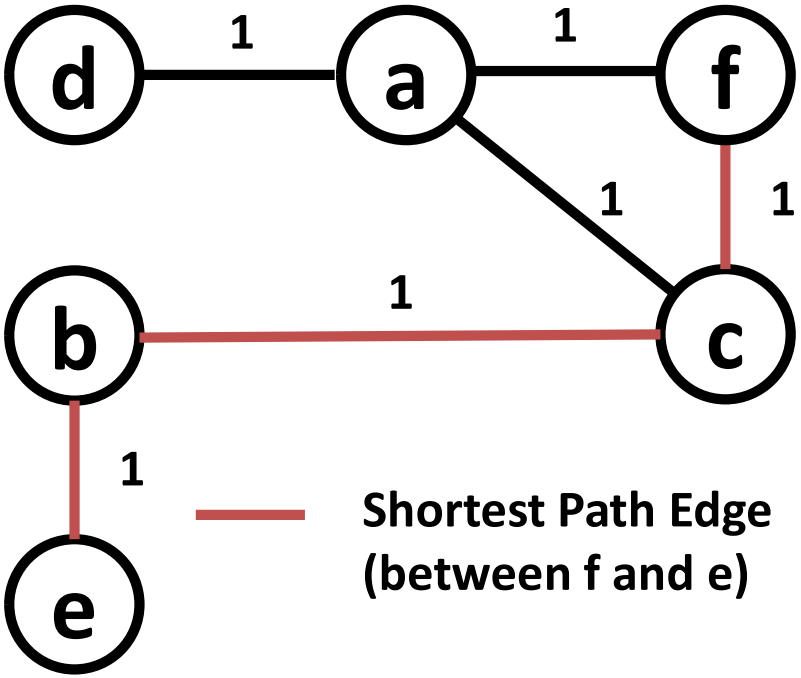
What about ties?

Search Tree: Uniform Action Cost



Uniform Cost Search | Dijkstra's Algo

Weighted Graph G



Popular algorithms:

- **Dijkstra's algorithm**

Shortest Path Problem

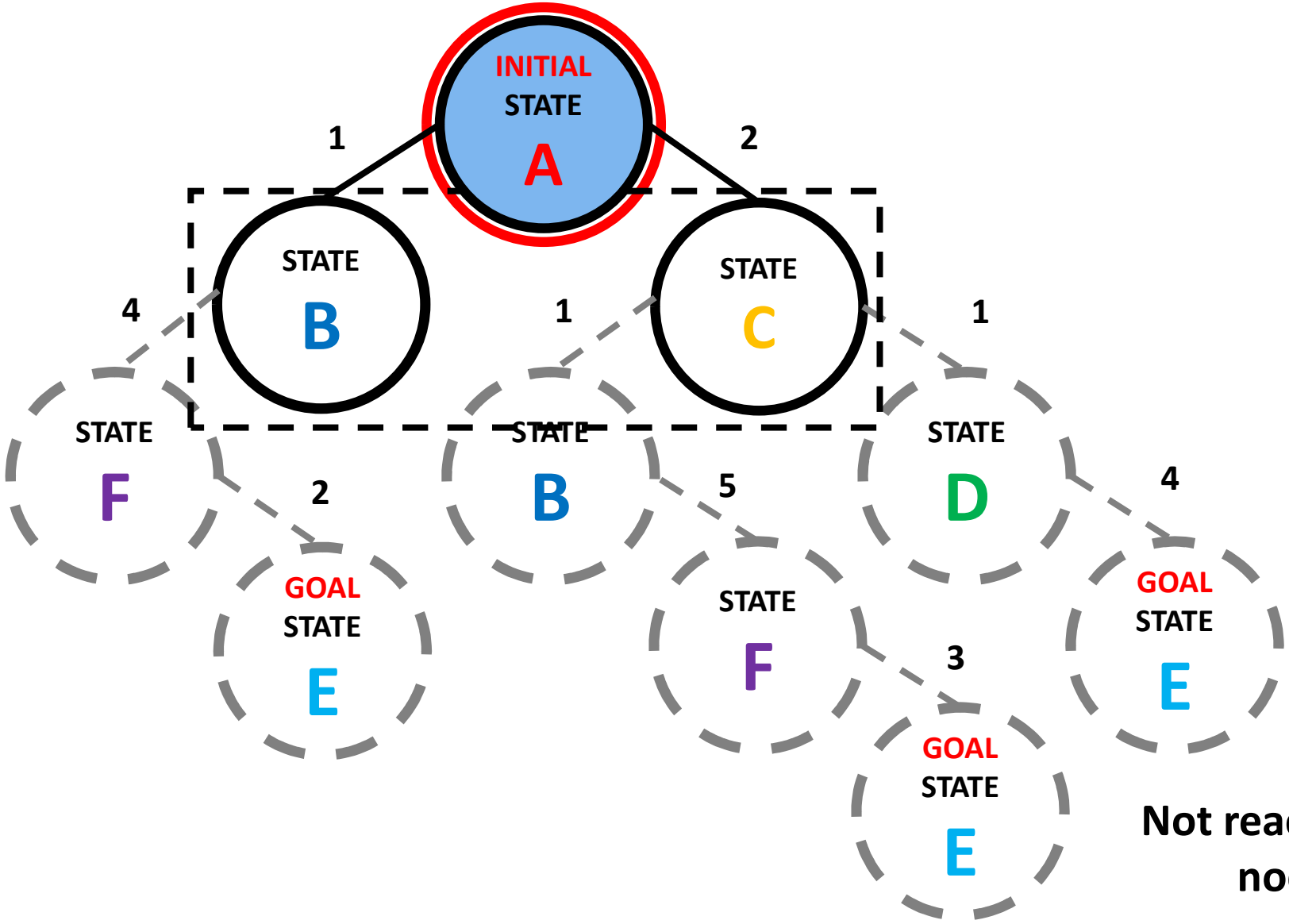
Shortest path problem:
Given a weighted graph $G(V, E, w)$ and two vertices a, b in V, find the shortest path between vertices a and b (**all edge weights are equal**).

BFS and UCS: Pseudocode

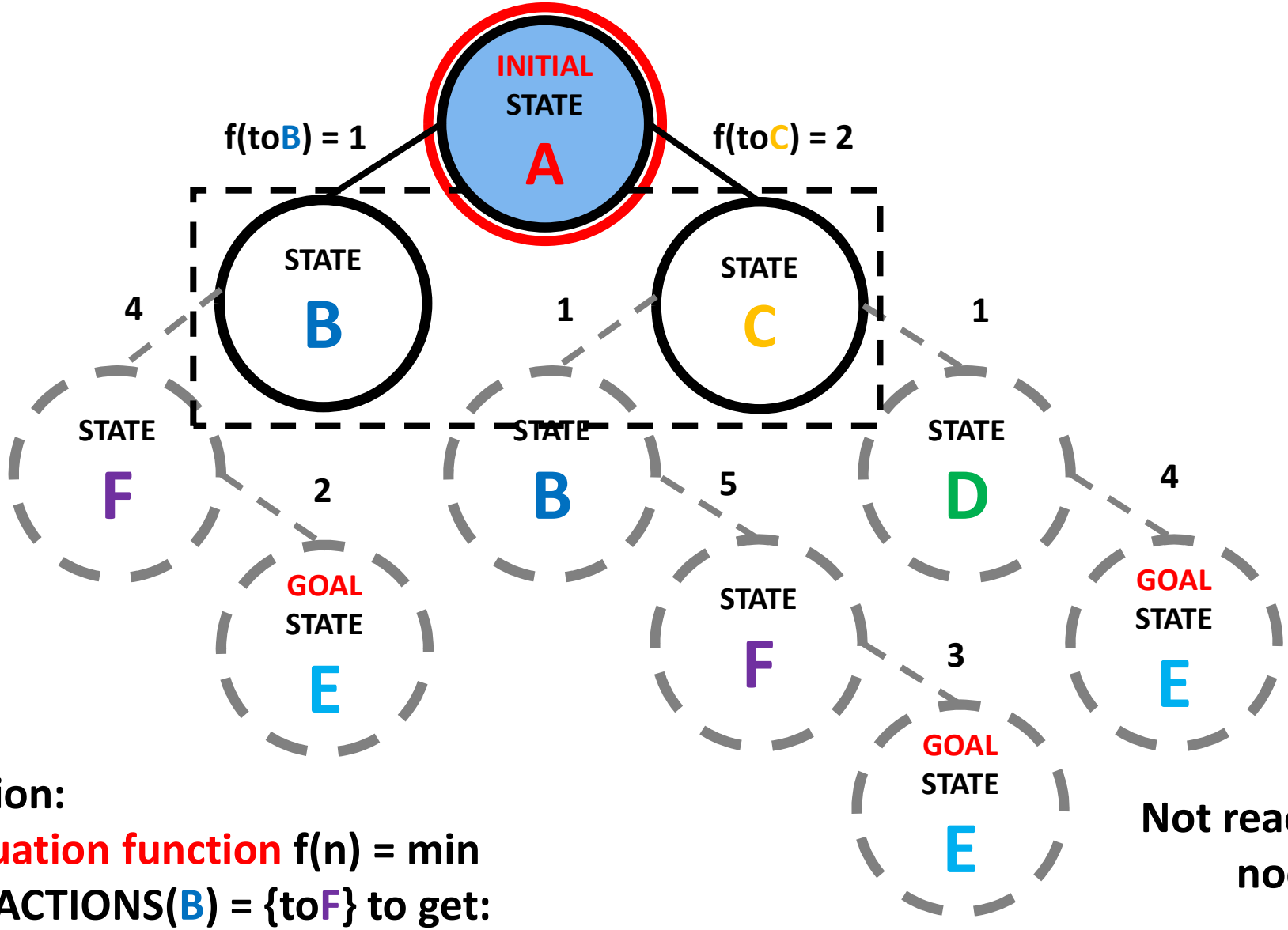
```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
 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

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

Search Tree: Variable Action Cost



Search Tree: Variable Action Cost

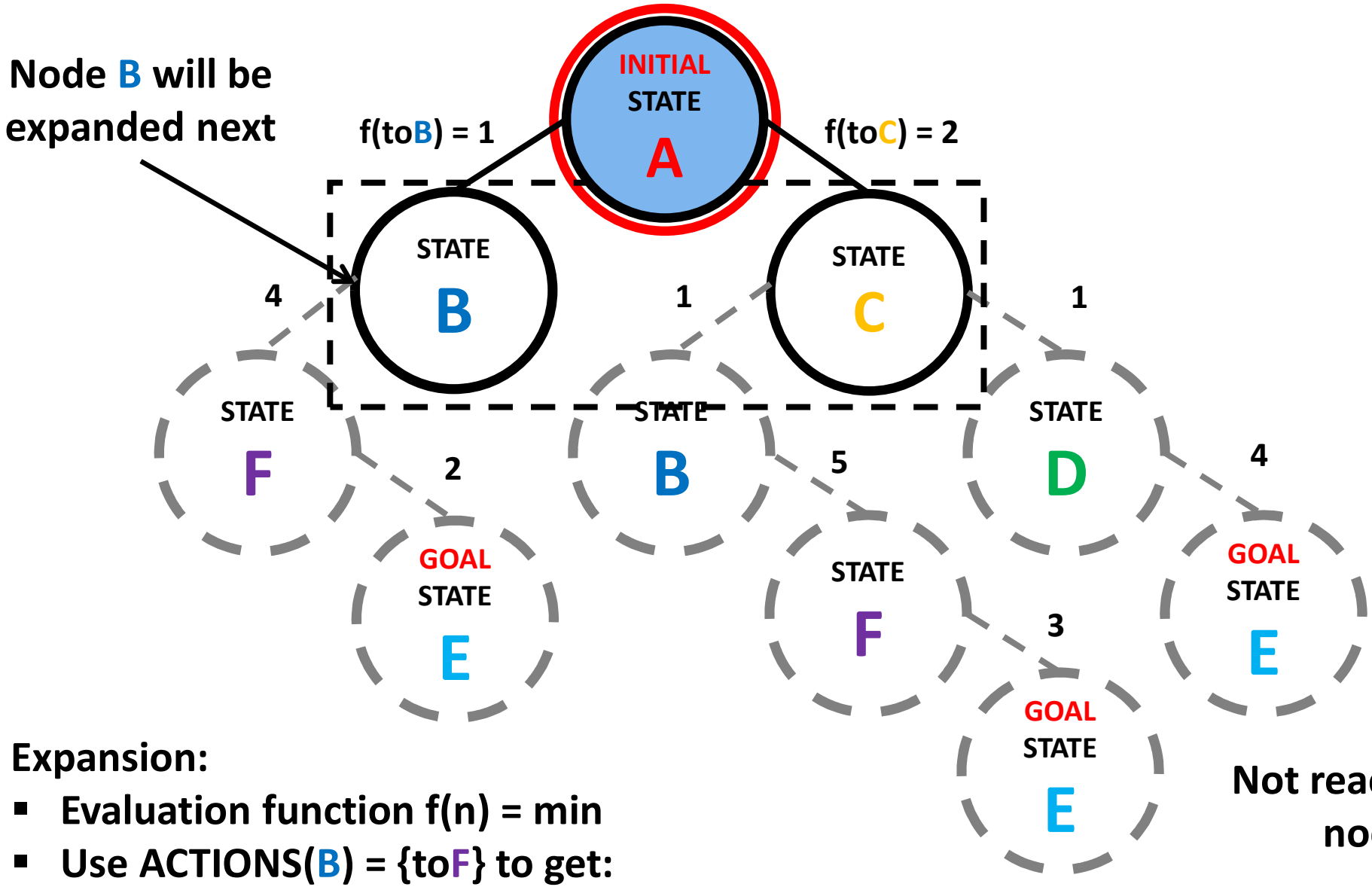


Expansion:

- Evaluation function f(n) = min
- Use ACTIONS(B) = {toF} to get:
- RESULT(B, toF) = F

Not reached yet nodes

Search Tree: Best-First Search

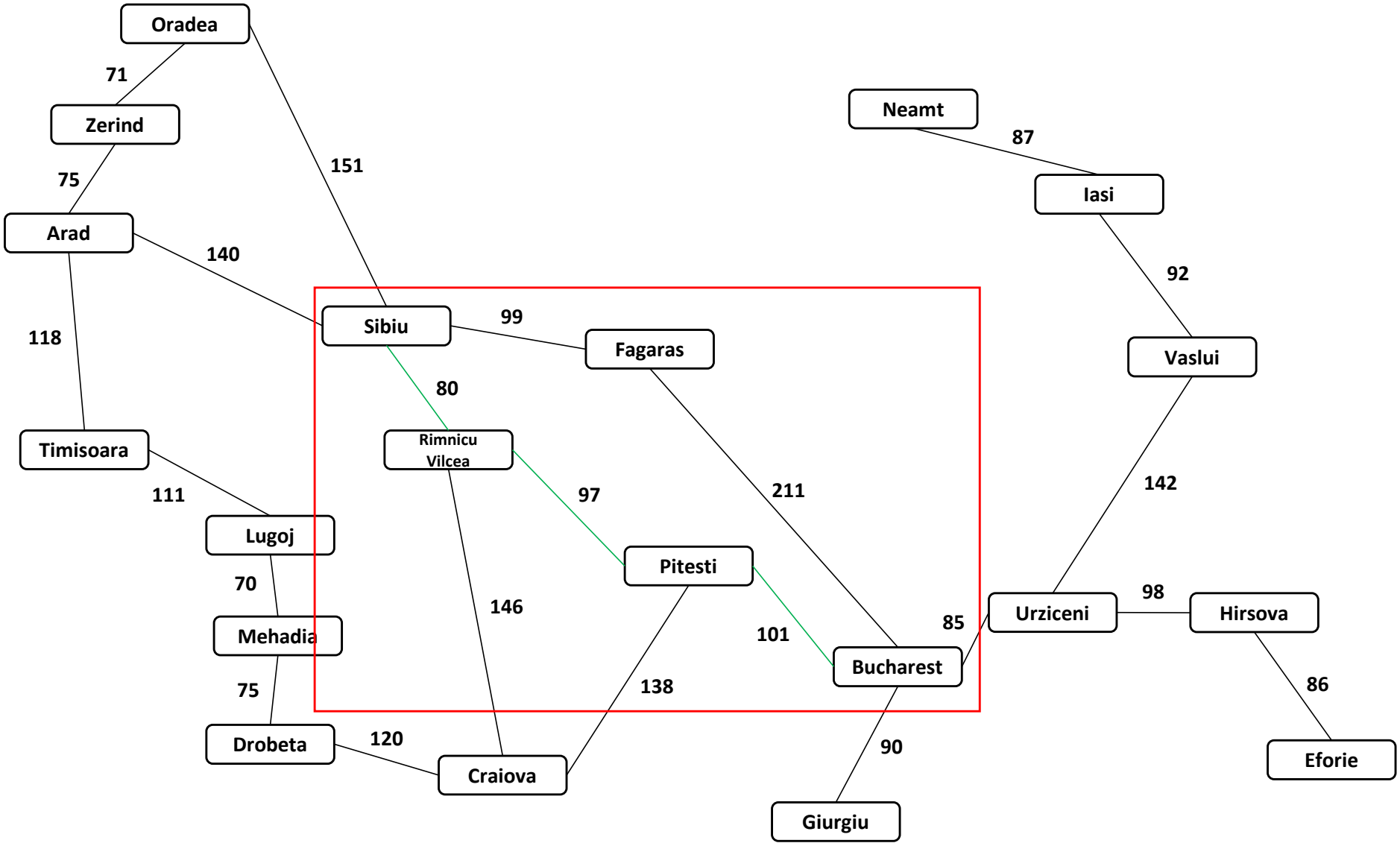


Best-First Search: Pseudocode

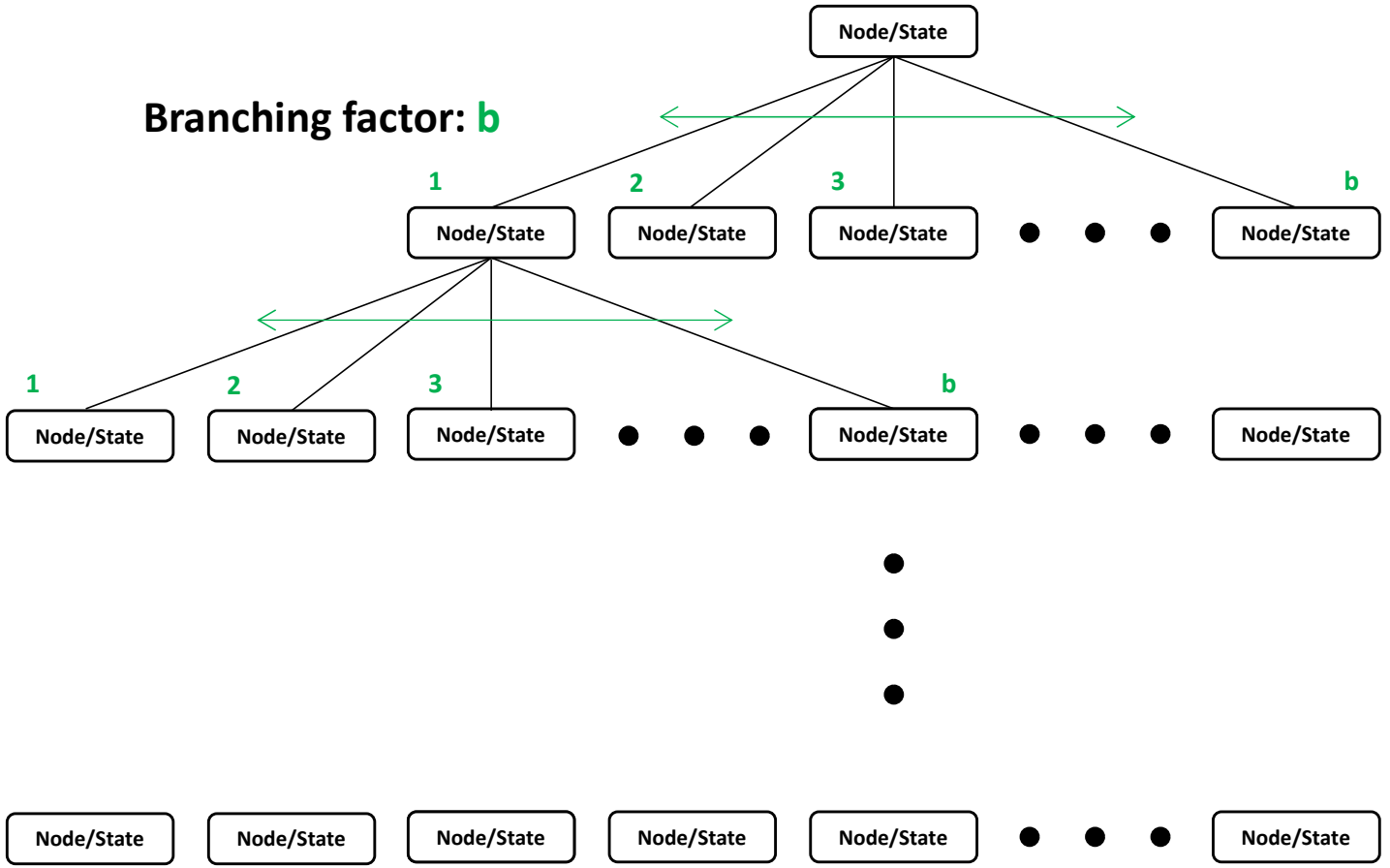
```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    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)
```

Best First Search: Issue



Let's Go Back to Depth First Search



Depth: 0 | $N_0 = 1$

Depth: 1 | $N_1 = b$

Depth: 2 | $N_2 = b^2$

Depth: d | $N_d = b^d$

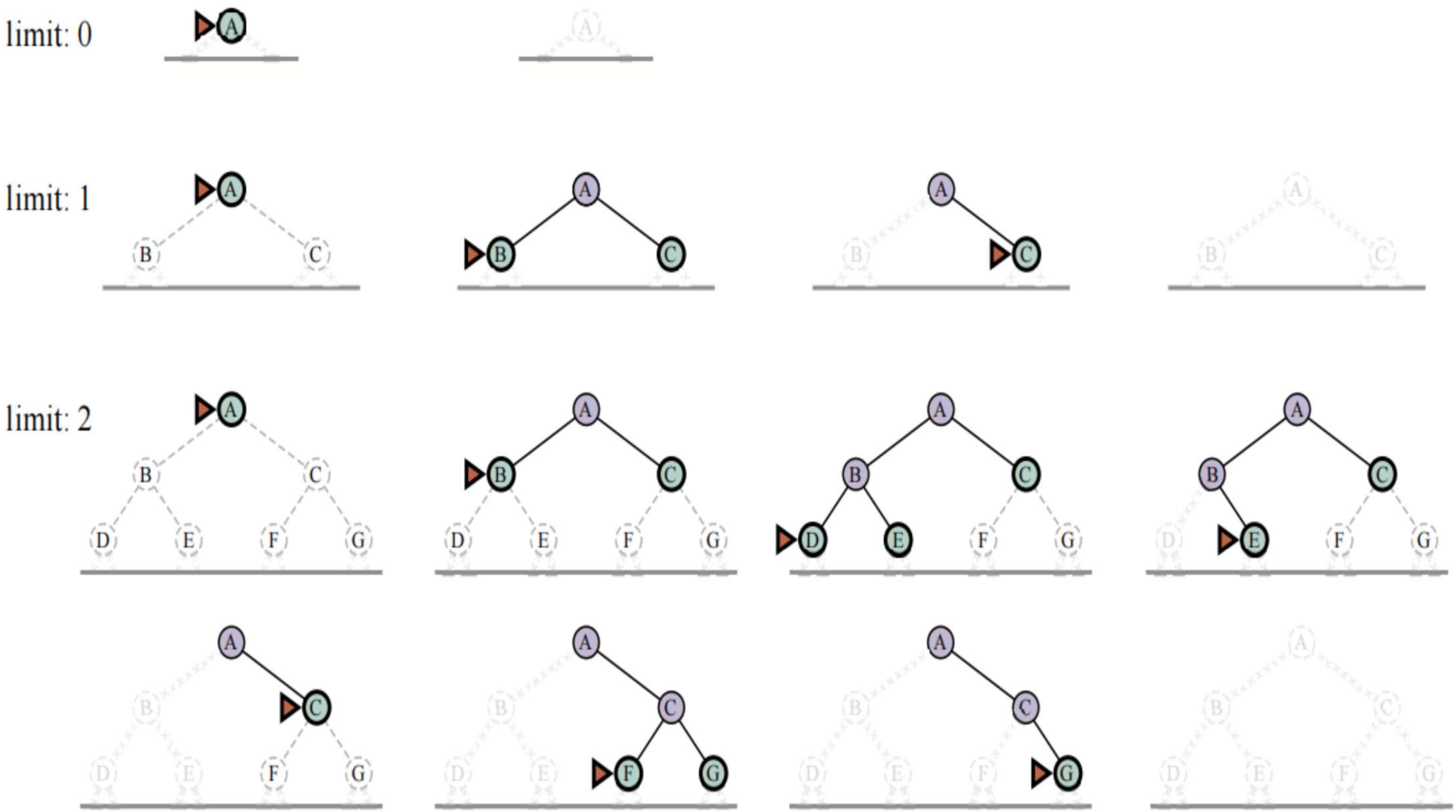
Tree depth is an issue!

"Controlled" DFS: Pseudocode

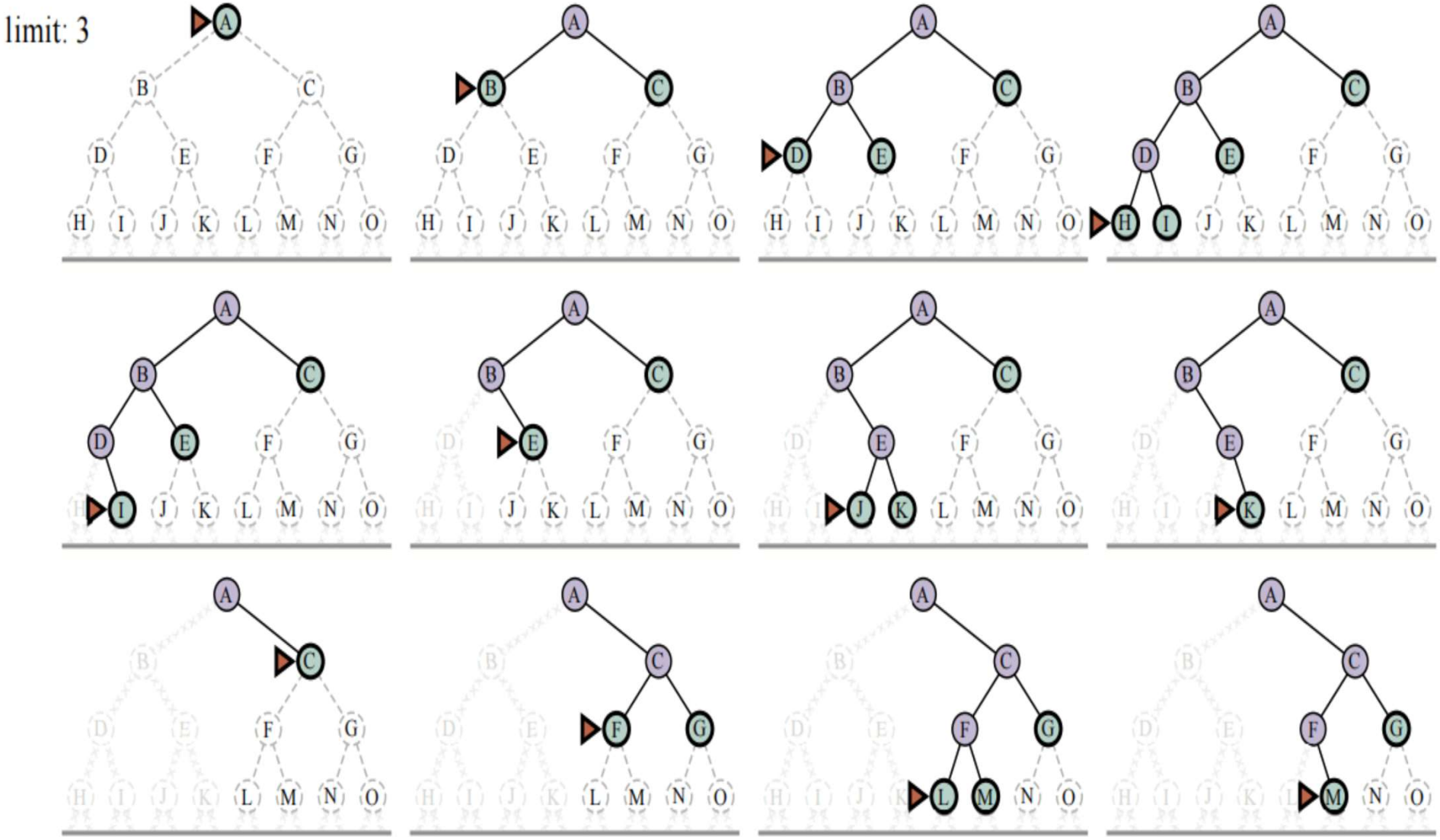
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

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
```

Iterative Deepening DFS: Illustration



Iterative Deepening DFS: Illustration

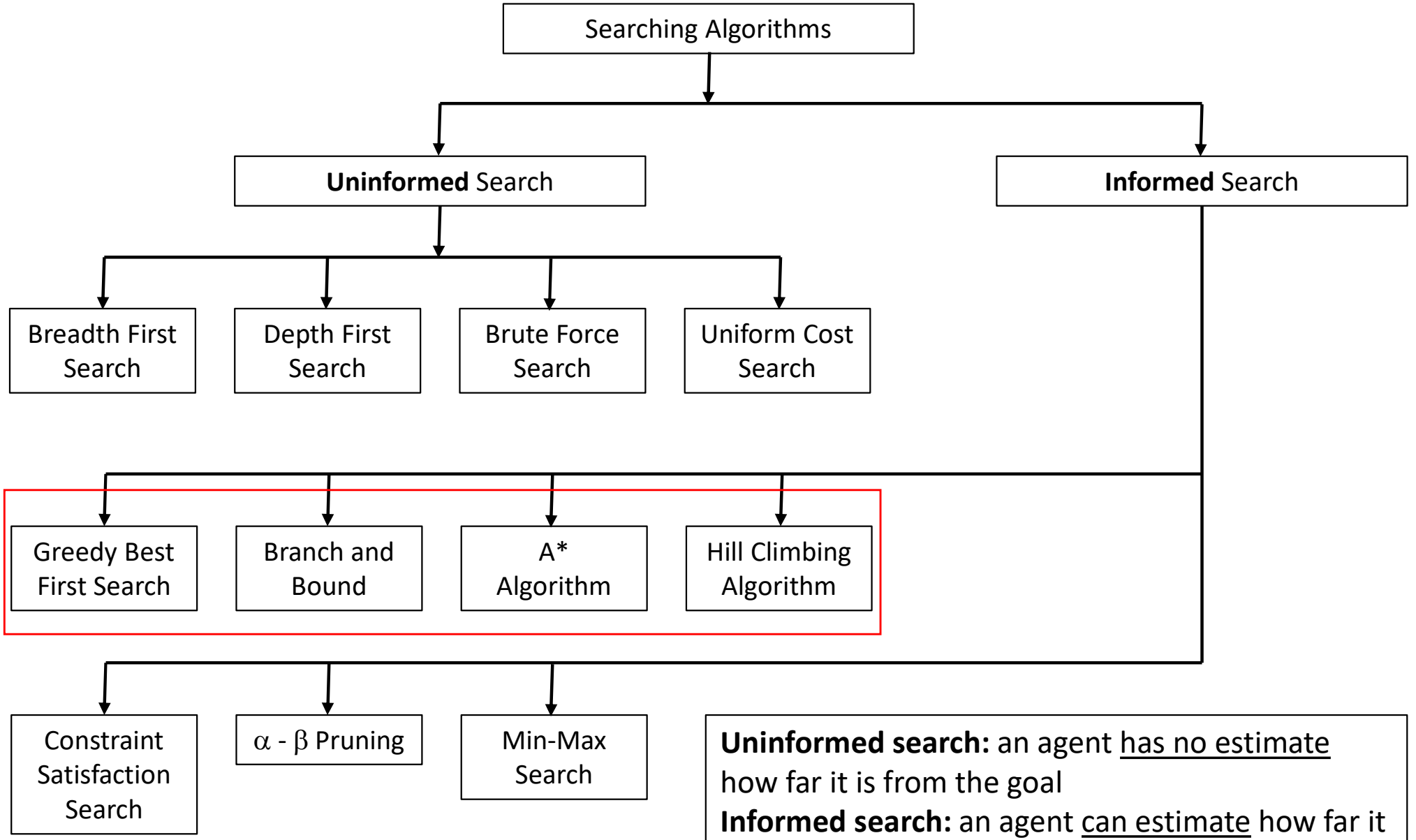


Uninformed Search Algorithms

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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

Selected Searching Algorithms



Uninformed search: an agent has no estimate how far it is from the goal
Informed search: an agent can estimate how far it is from the goal

Informed Search and Heuristics

Informed search relies on domain-specific knowledge / hints that help locate the goal state.

h(n) = h(State n)

h(n) = n(relevant information about State n)

h(n) : heuristic function - estimated cost of the cheapest path from State n to the goal state