CS 480

Introduction to Artificial Intelligence

January 18, 2024

Announcements / Reminders

- **Contribute to the discussion on Blackboard, please**
- **Please follow the Week 02 To Do List instructions (if you haven't already):**

- **Next week I ***plan*** to start attendance taking (for my personal records of class participation)**

Teaching Assistants

Name	e-mail	Office hours
Nagaraju, Ashish	anagaraju@hawk.iit.edu	Mondays 10:30 AM – 12:30 PM CST in SB 108
Vishwanath, Tejass	tvishwanath@hawk.iit.edu	Fridays 02:00 PM - 03:00 PM CST in SB 108

TAs will:

- assist you with your assignments,
- hold office hours to answer your questions,
- grade your lab work (**a specific TA will be assigned to you**).

Take advantage of their time and knowledge!

DO NOT email them with questions unrelated to lab grading.

Make time to meet them during their office hours.

Add a **[CS480 Spring 2024]** prefix to your email subject when contacting TAs, please.

Plan for Today

- **Intelligent Agents**
- **Solving problems by Searching**

Designing the Agent for the Task

Analyze the Problem / Task (PEAS)	Select Agent Architecture	Select Internal Representations	Apply Corresponding Algorithms

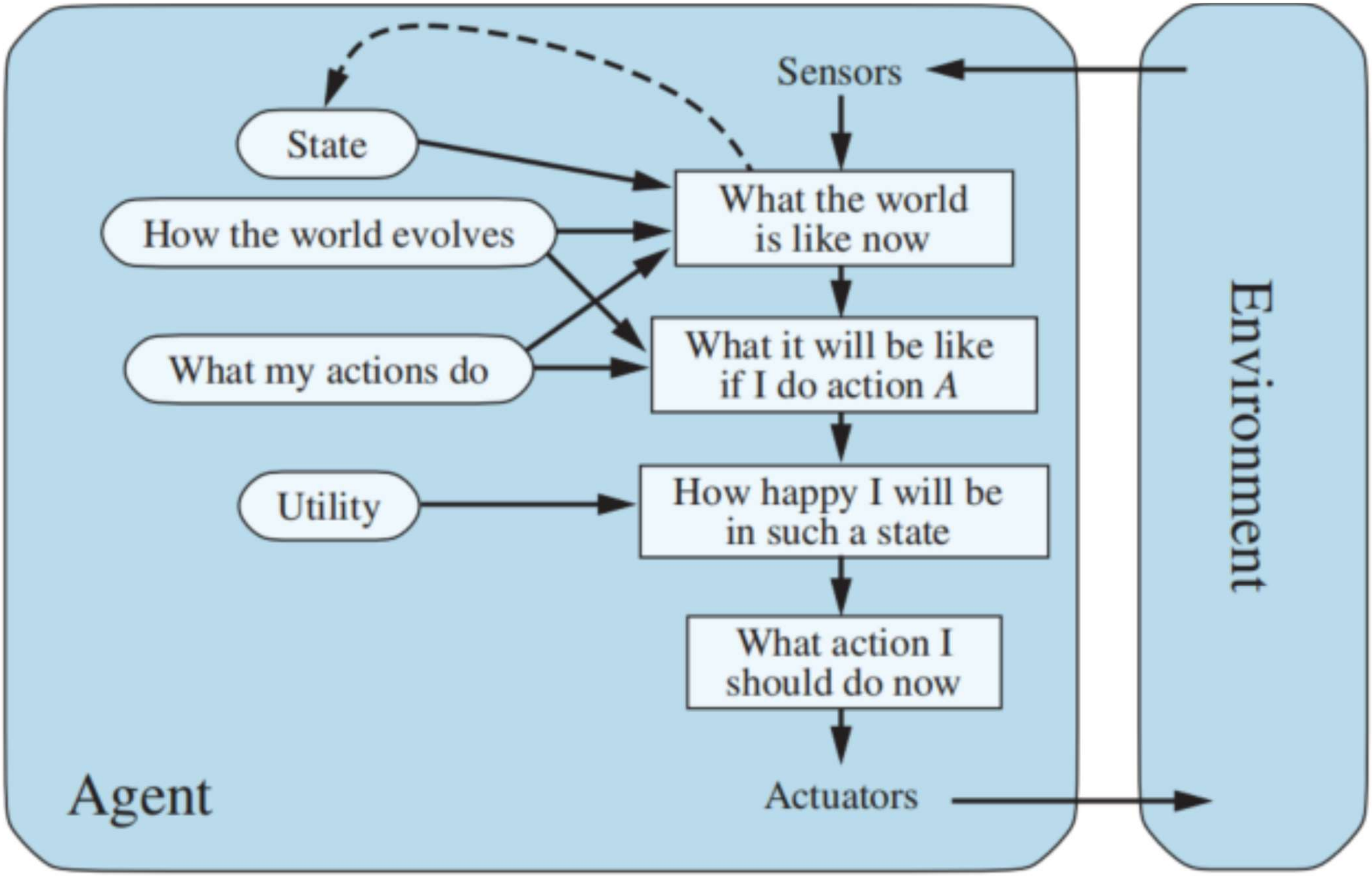
Agent Structure / Architecture

Agent = Architecture + Program

Typical Agent Architectures

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

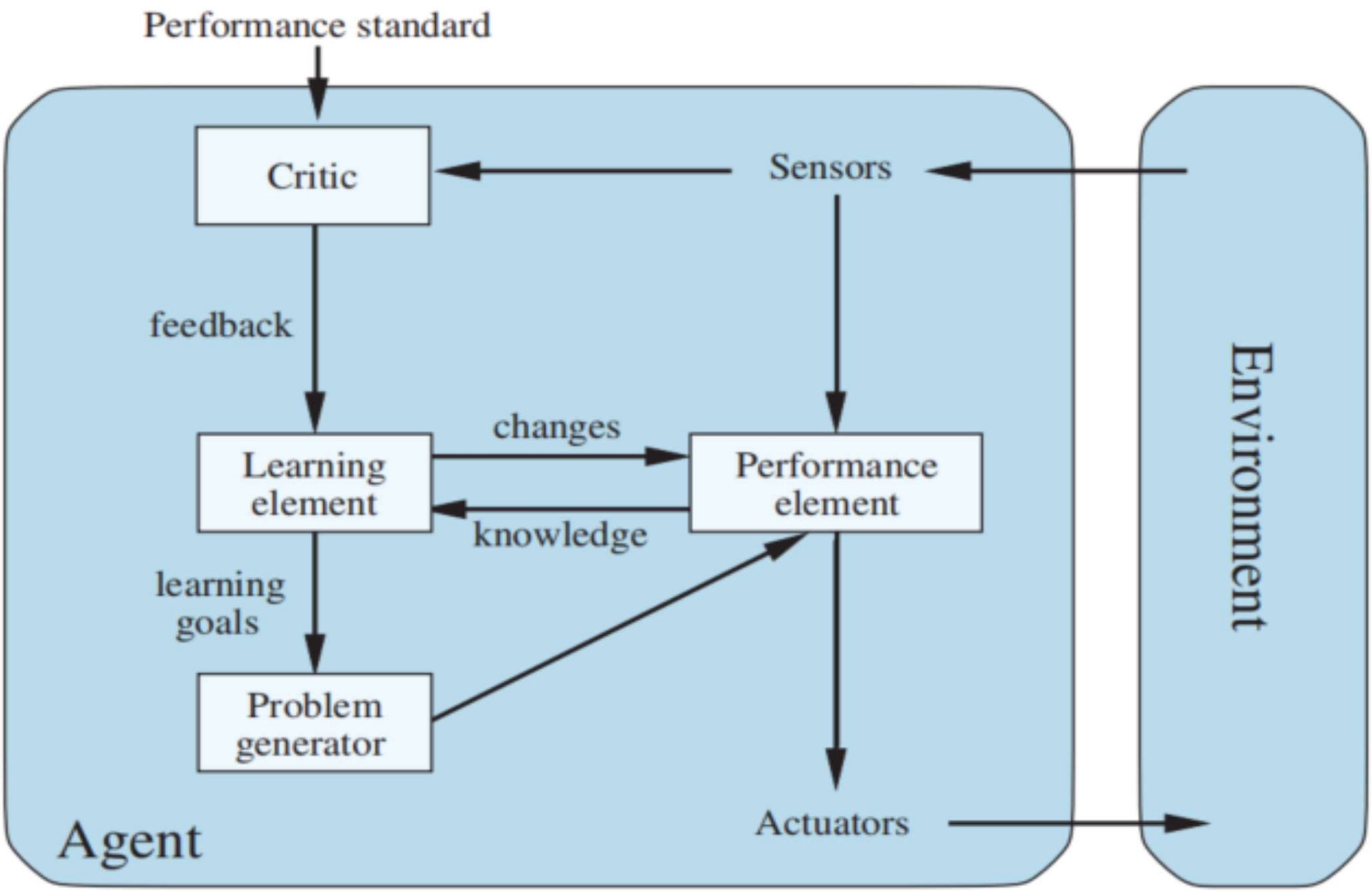
Model-based Agents: Challenges?



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))**

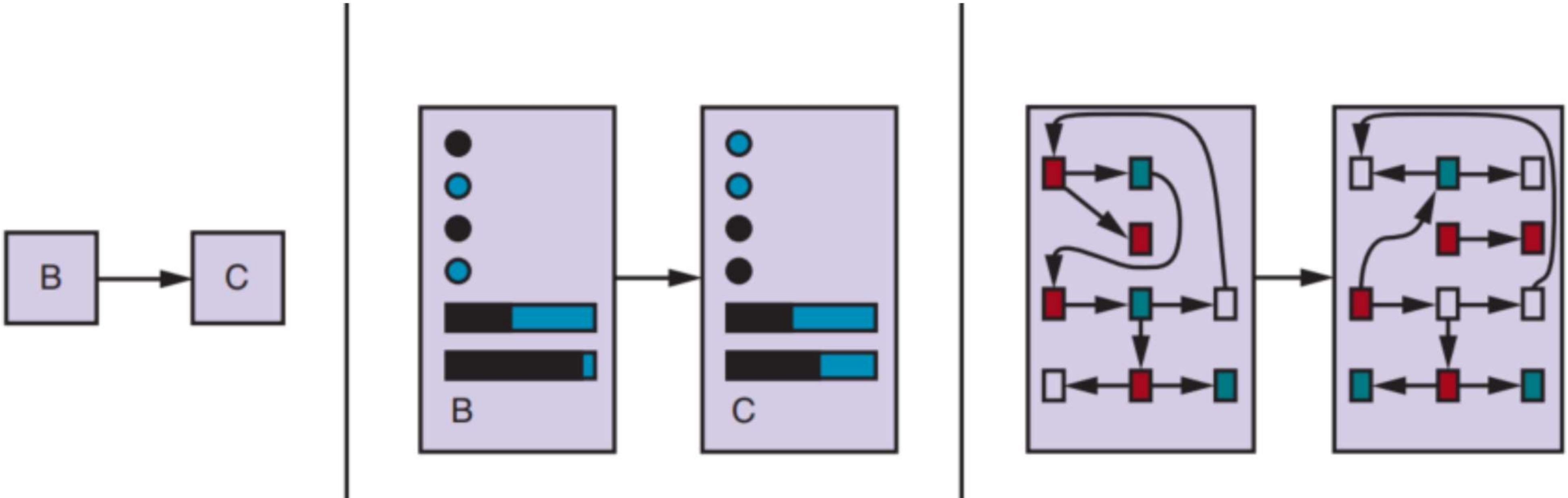
Learning Agent



Designing the Agent for the Task

Analyze the Problem / Task (PEAS)	Select Agent Architecture	**Select Internal Representations**	Apply Corresponding Algorithms

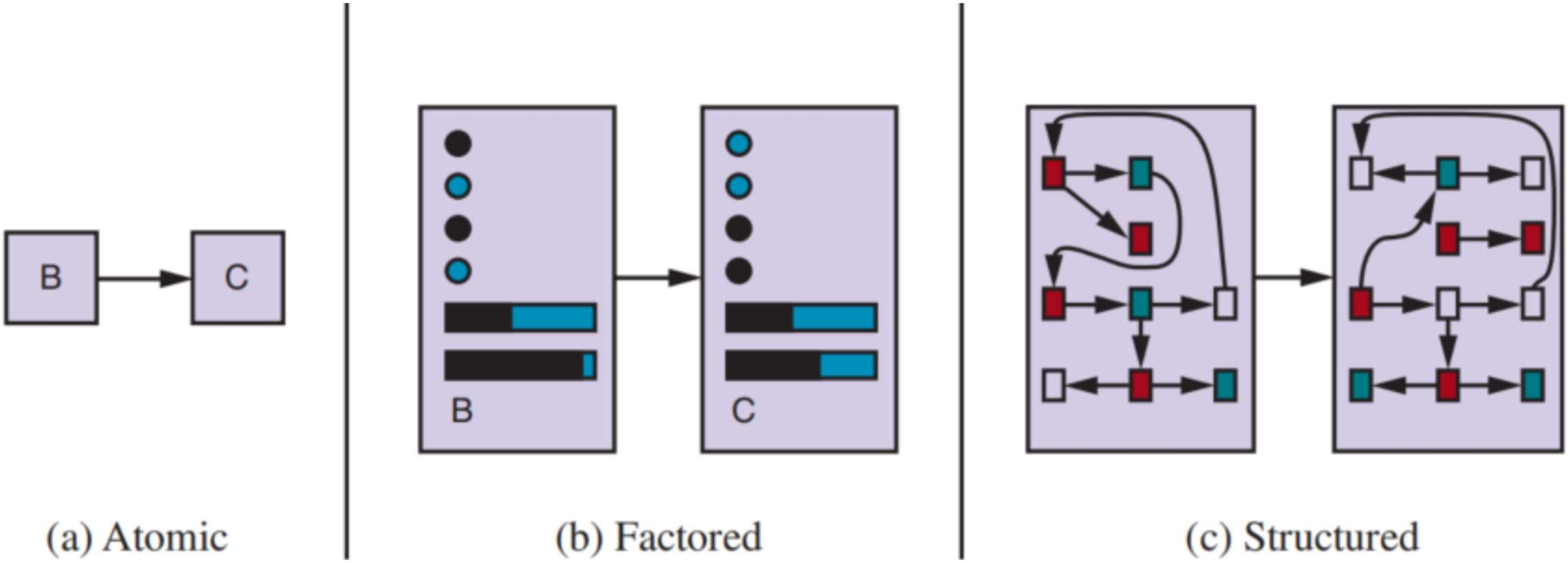
State and Transition Representations



(a) Atomic (b) Factored (c) Structured

- **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



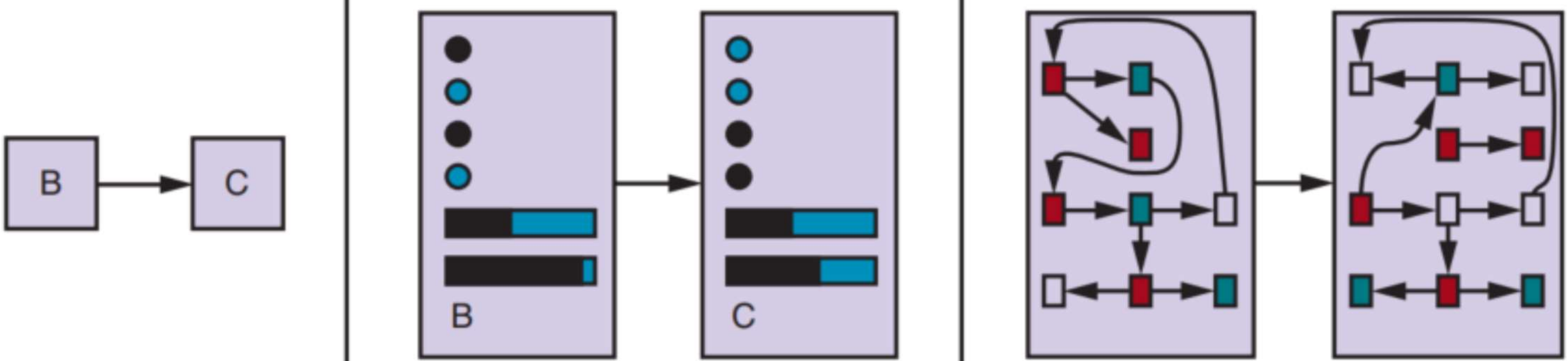
(a) Atomic

(b) Factored

(c) Structured

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

Representations and Algorithms



(a) Atomic (b) Factored (c) Structured

- 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

Analyze the Problem / Task (PEAS)	Select Agent Architecture	Select Internal Representations	Apply Corresponding Algorithms

Finite State Machine: A Turnstile

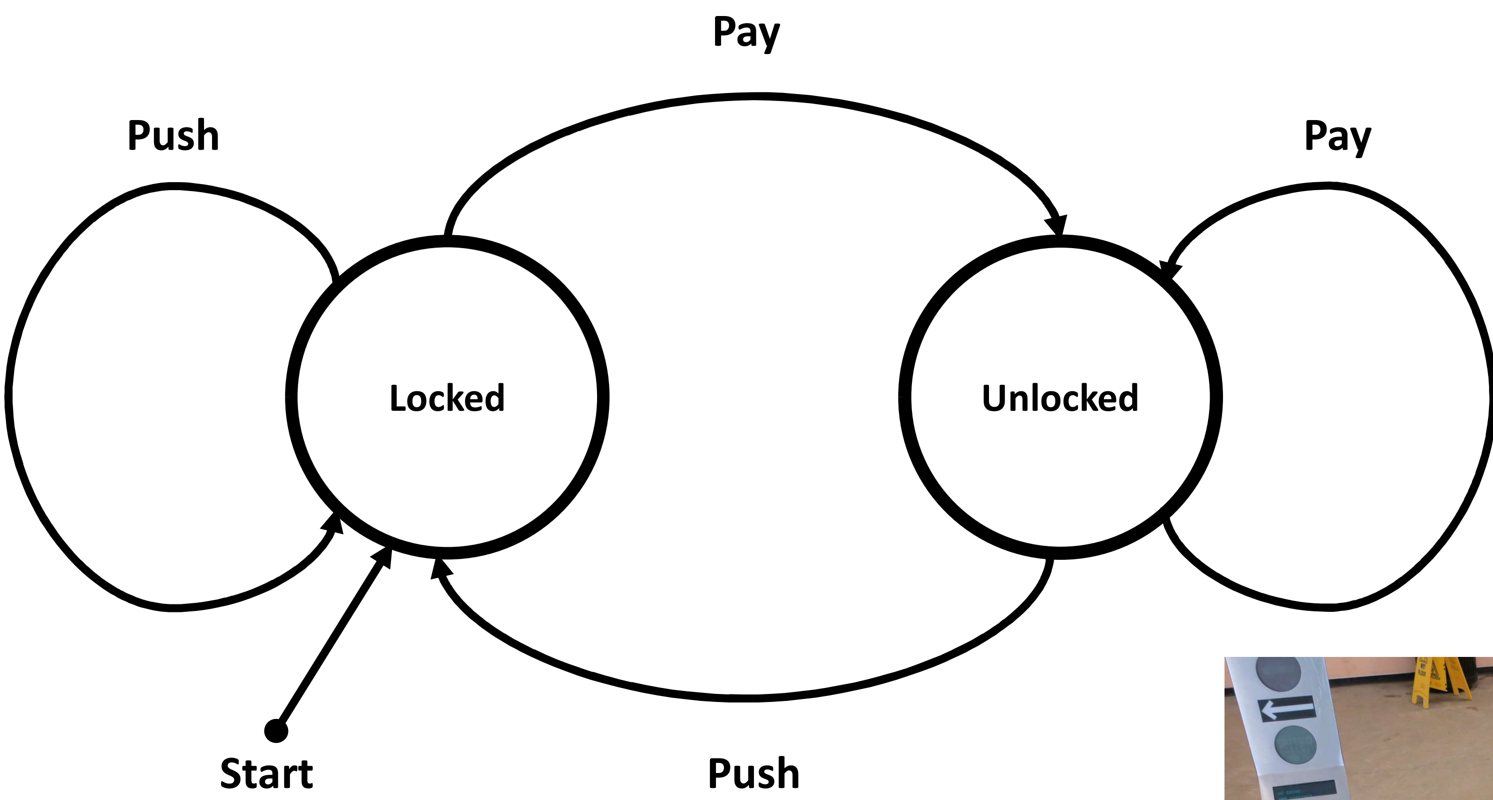
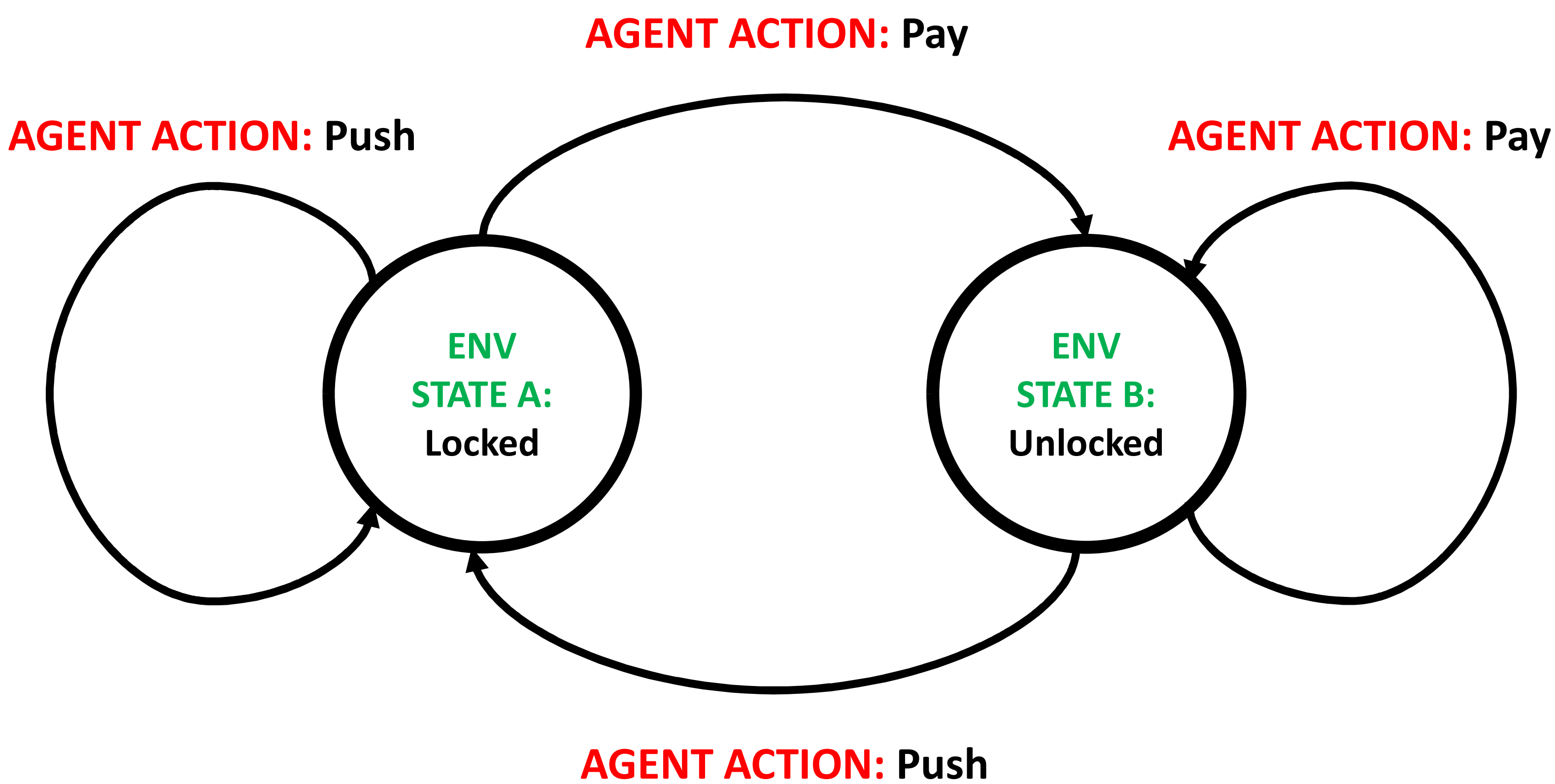
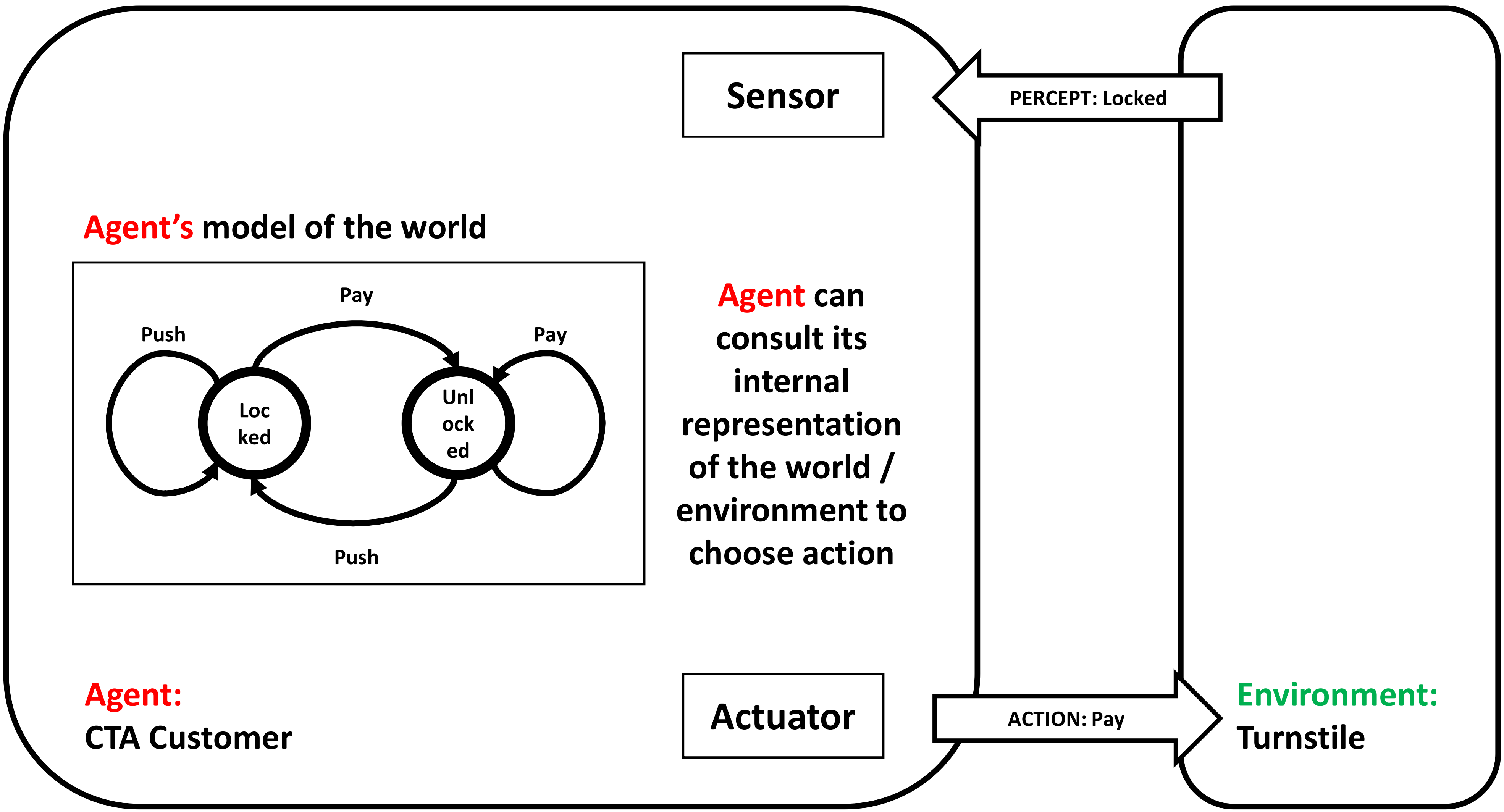


Image source: Wikipedia

Finite State Machine: A Turnstile

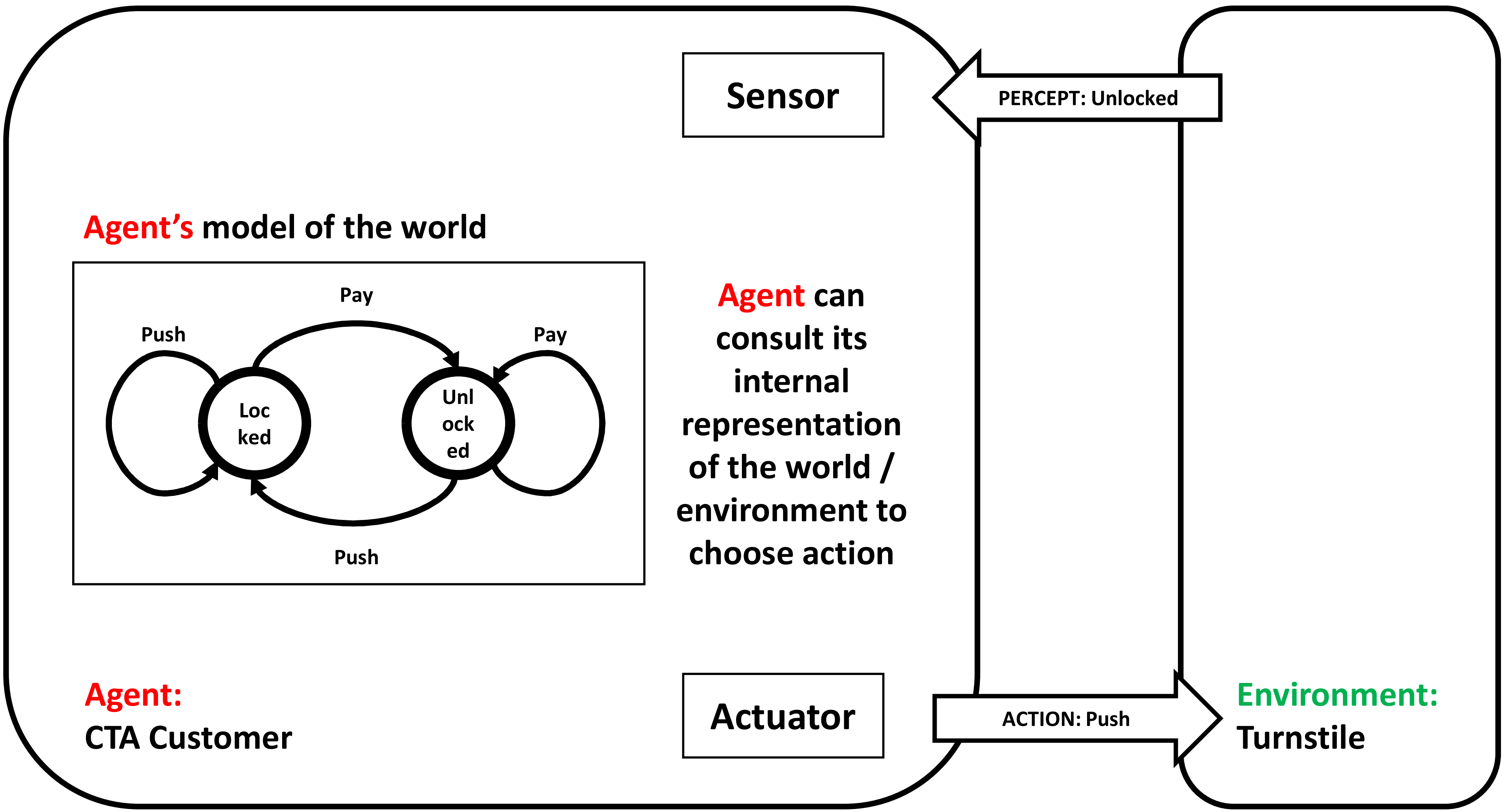


Model-based Reflex Agent Example



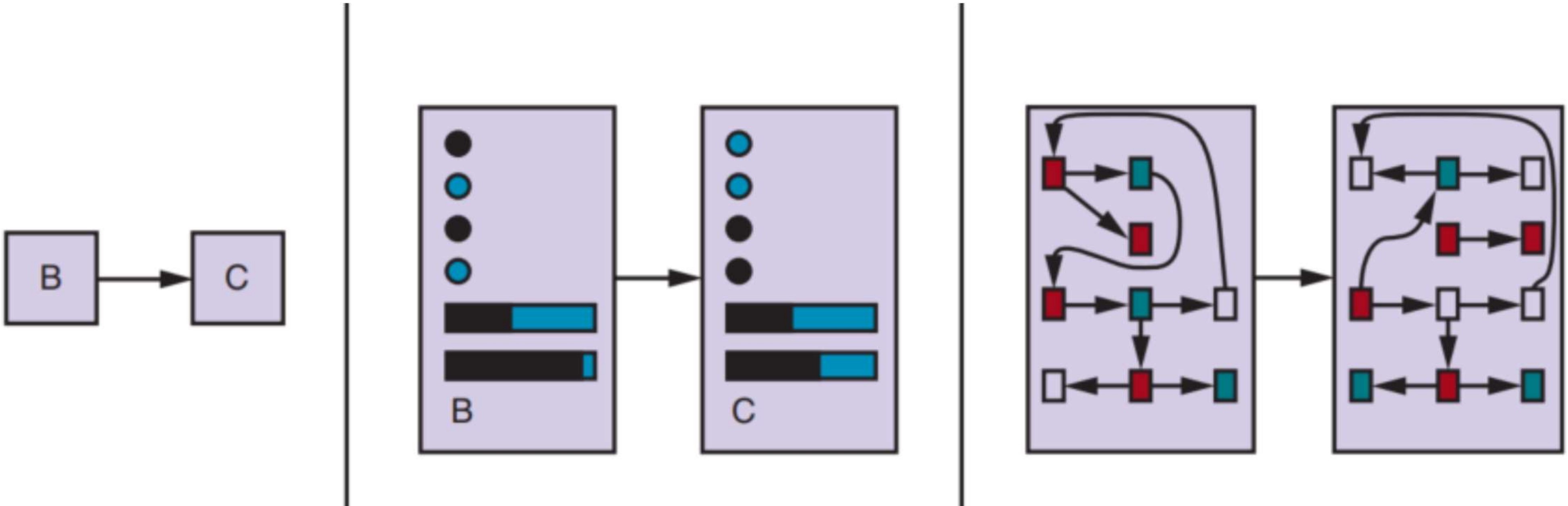
Note: This problem could be easily solved with a simple (without internal model) reflex agent.

Model-based Reflex Agent Example

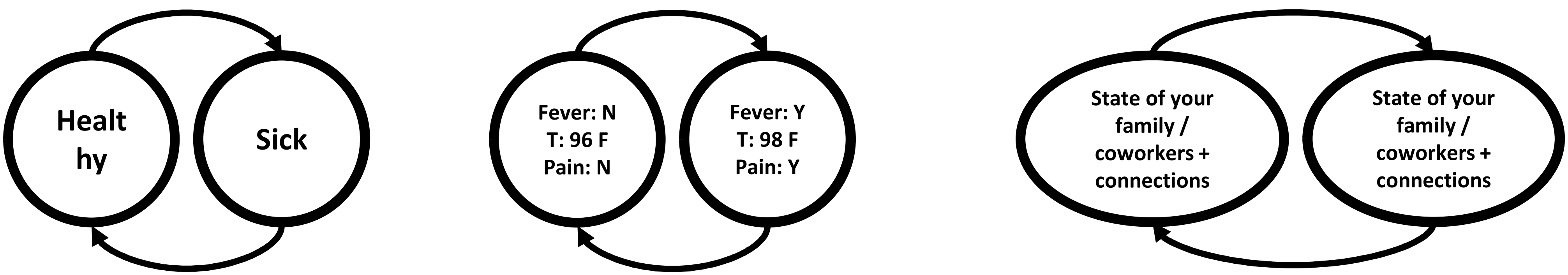


Note: This problem could be easily solved with a simple (without internal model) reflex agent.

Representations: Examples



(a) Atomic (b) Factored (c) Structured



Designing the Agent for the Task

Analyze the Problem / Task (PEAS)	Select Agent Architecture	Select Internal Representations	Apply Corresponding Algorithms

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 <u>search</u> 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)

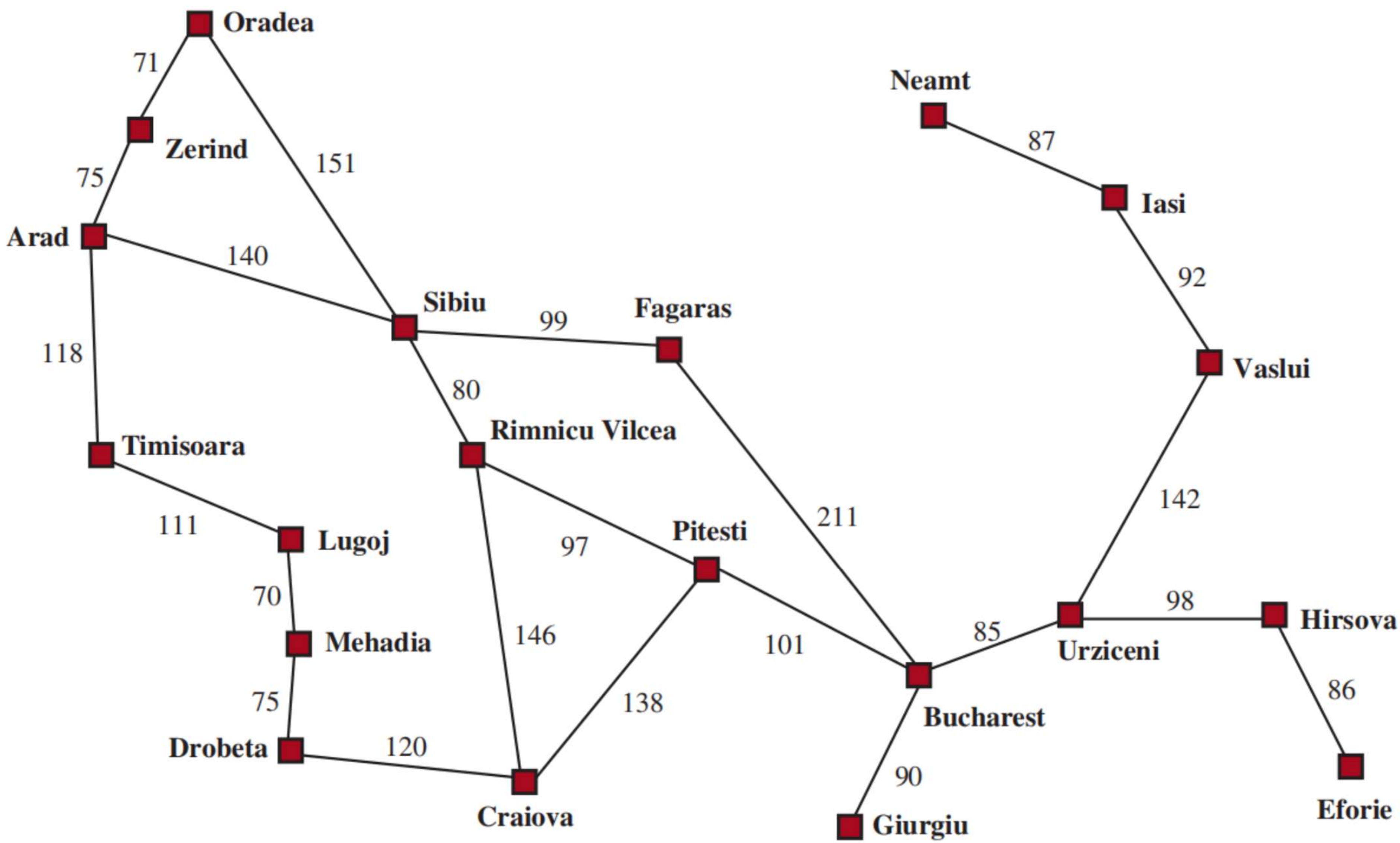
Designing the Searching Problem

Analyze and define the Problem / Task	Model and build the State Space	Select searching algorithm	Search

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

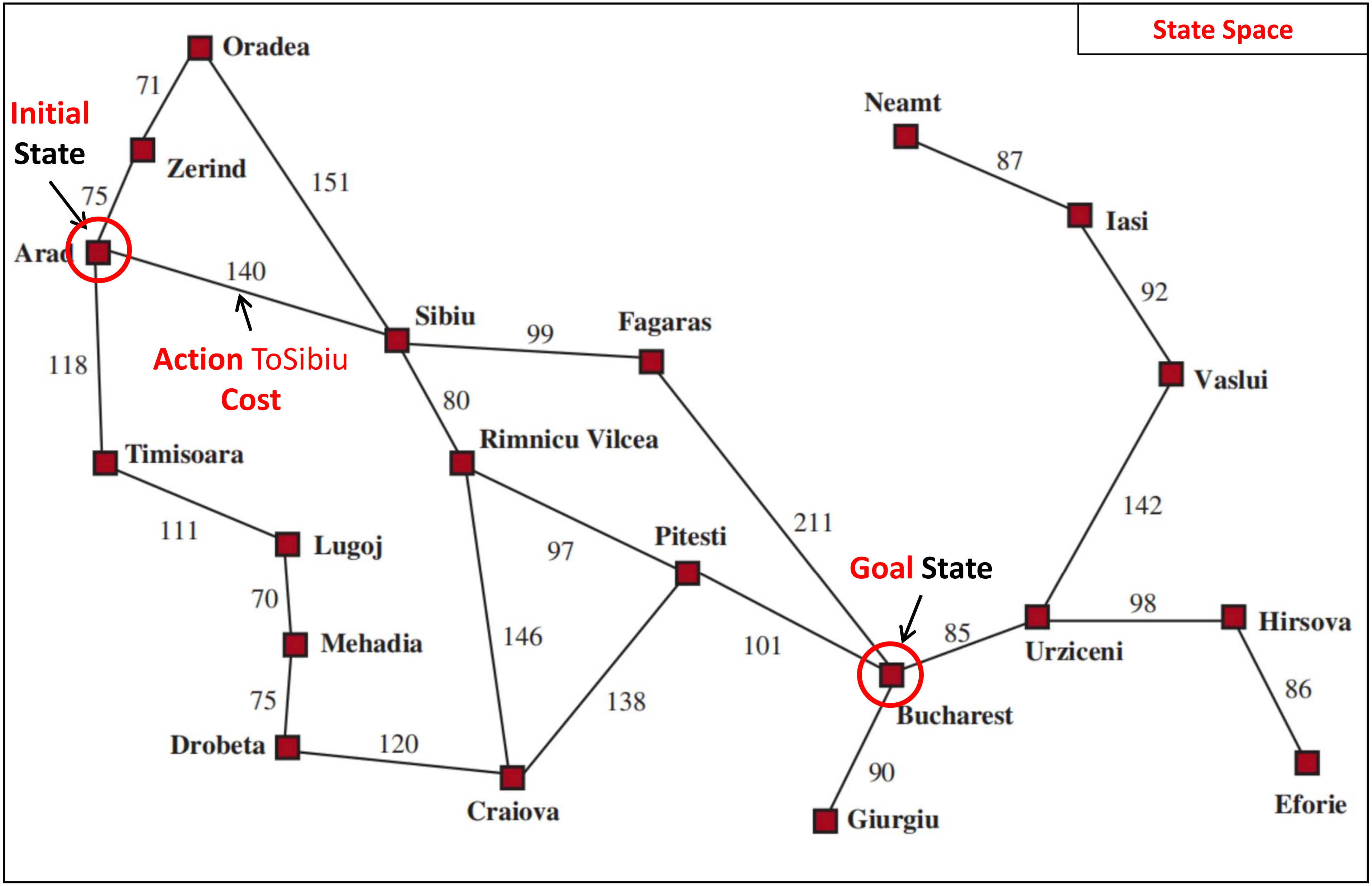


Problem: Get from Arad to Bucharest efficiently (for example: quickly or cheaply).

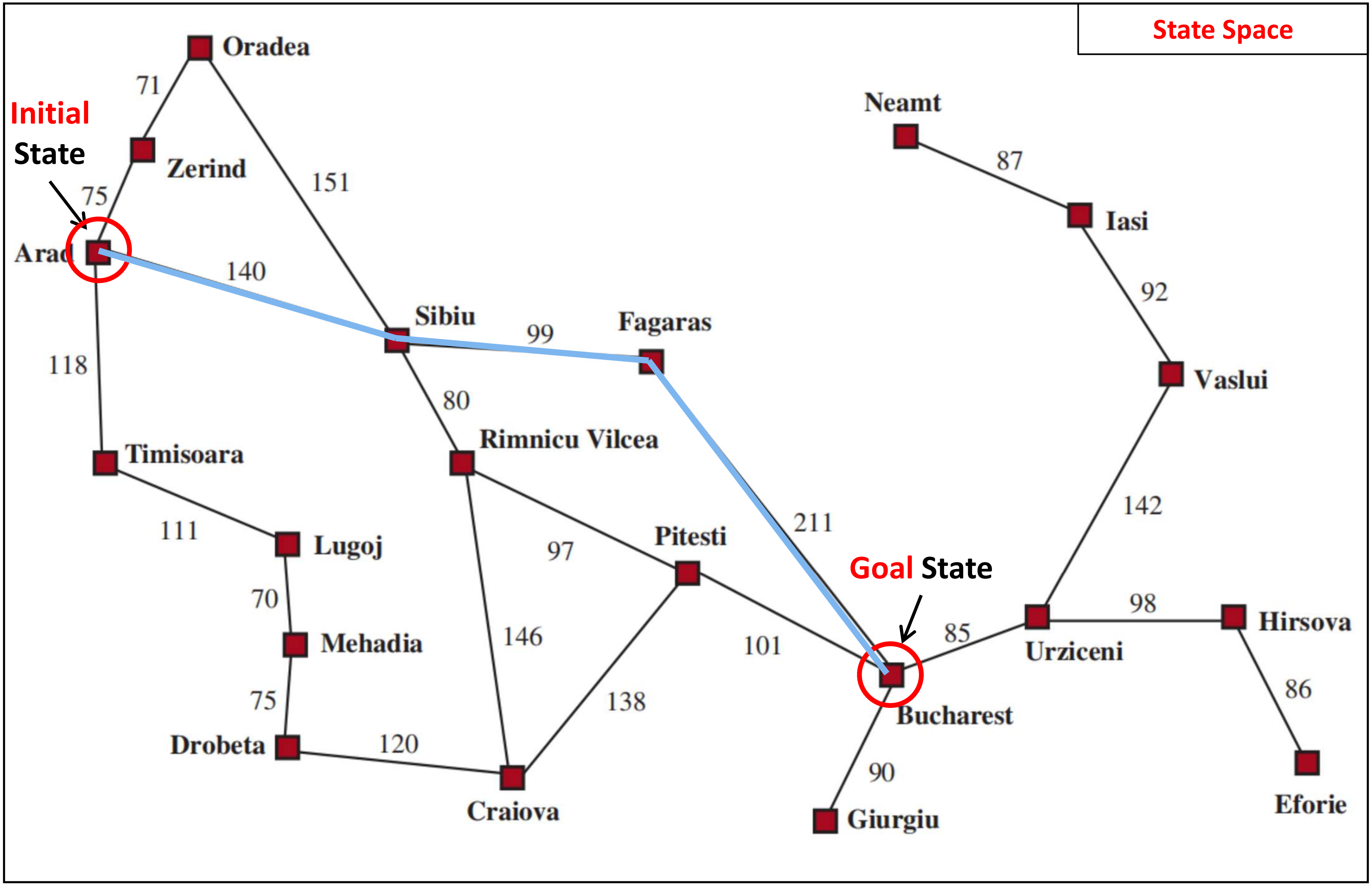
Search Problem: Romanian Roadtrip

- **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**

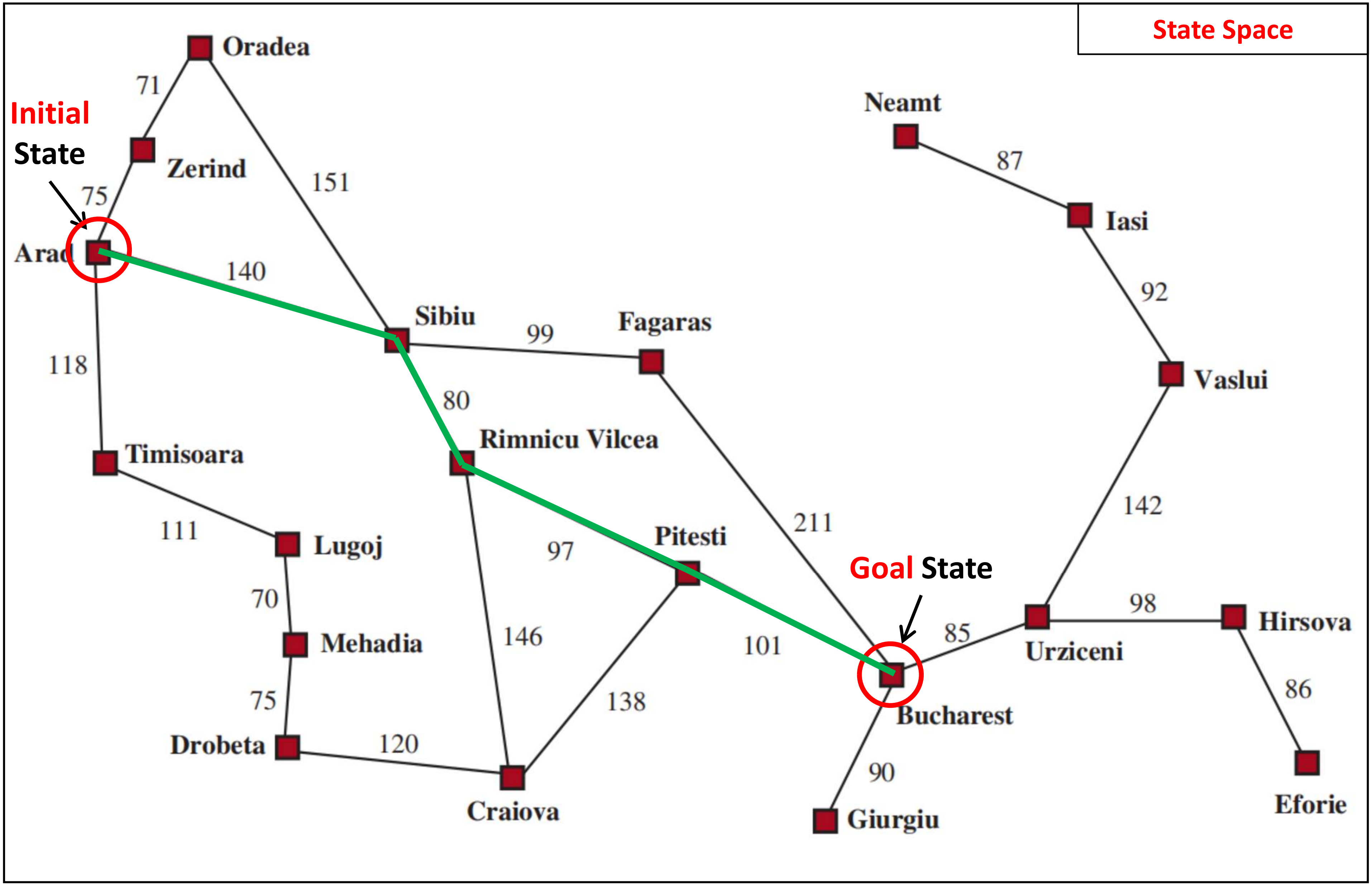
Sample Problem: Romanian Roadtrip



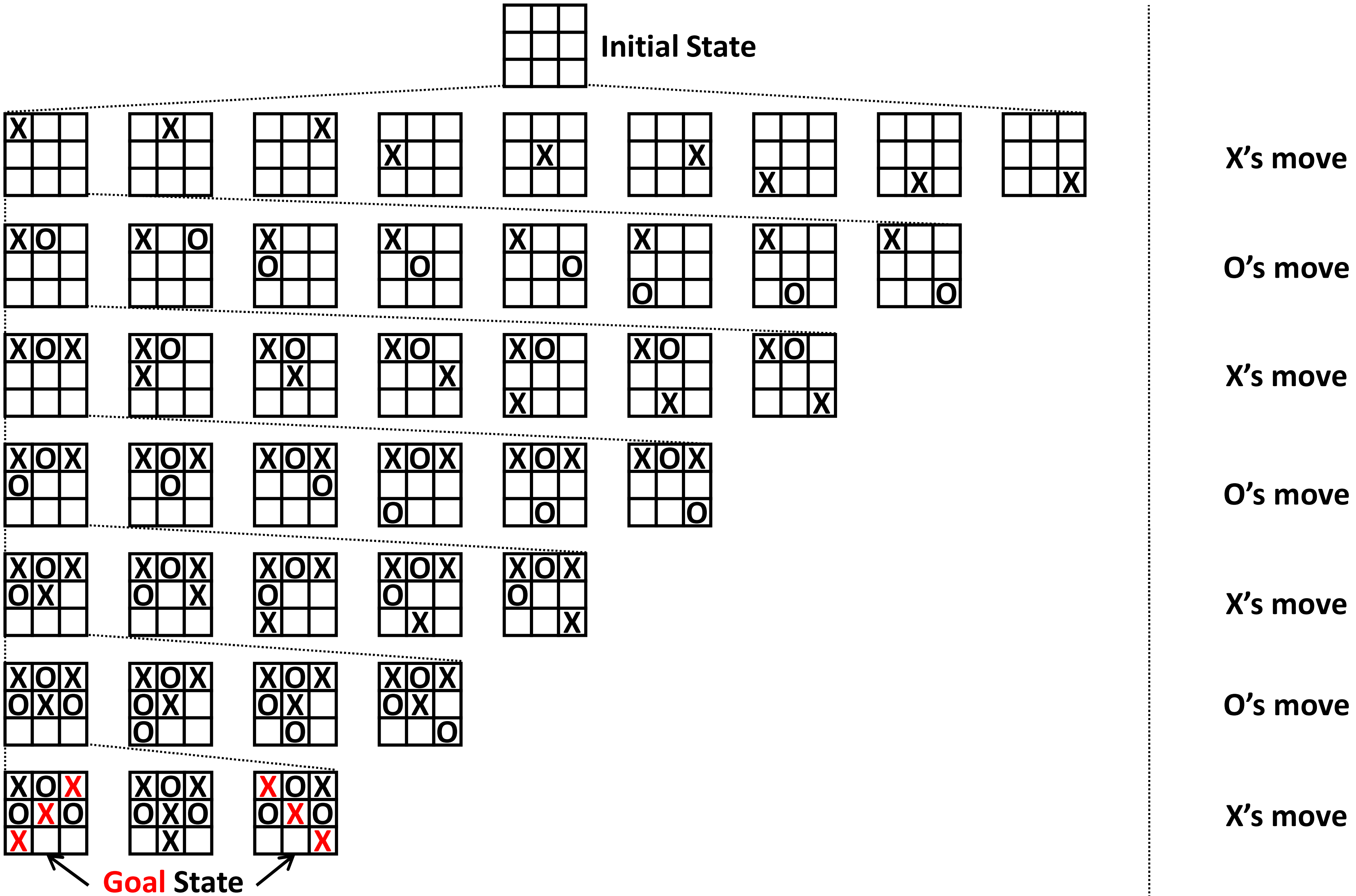
Romanian Roadtrip: Potential Solution



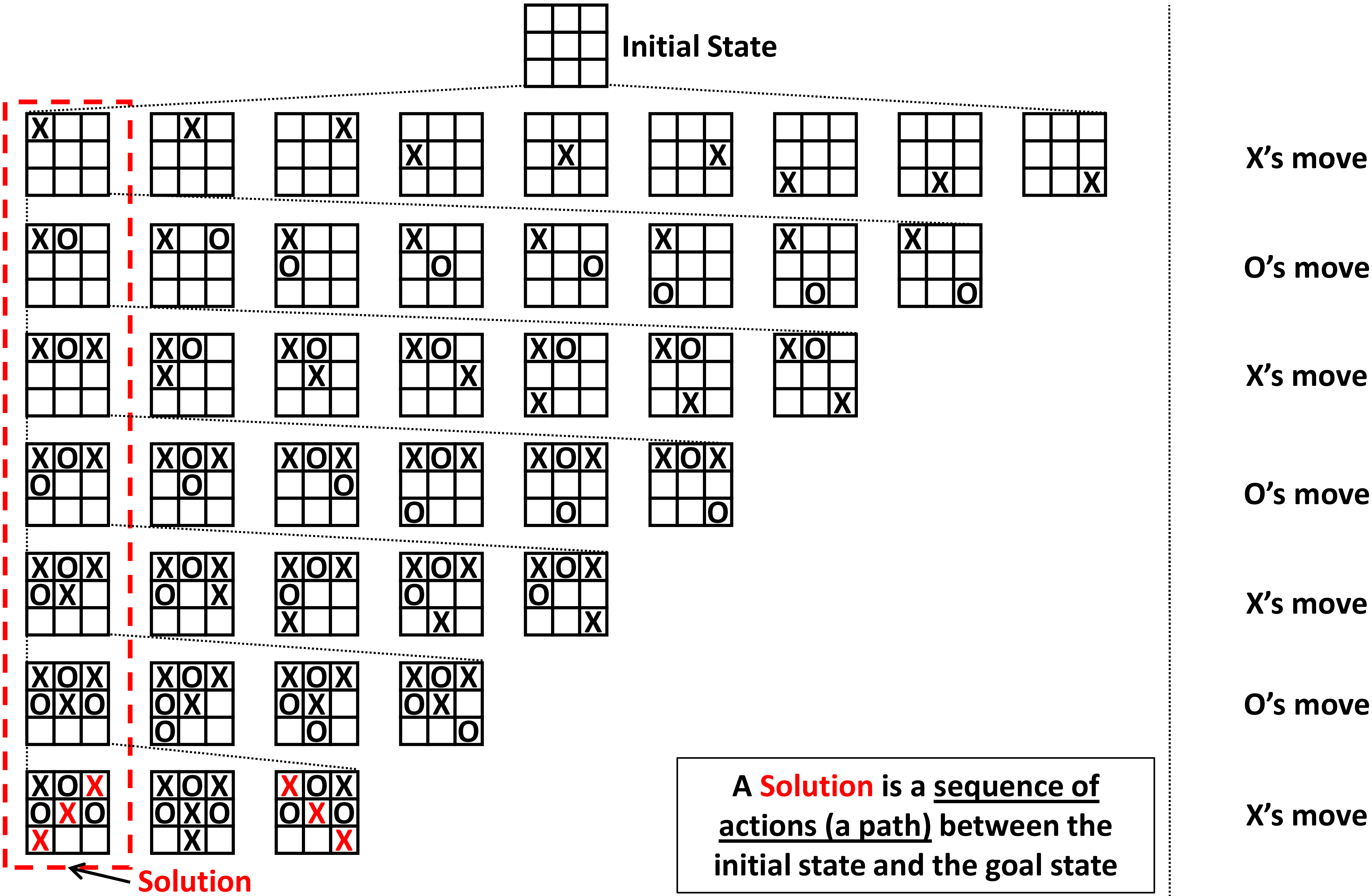
Romanian Roadtrip: Potential Solution



Tic Tac Toe: (Partial) State Space



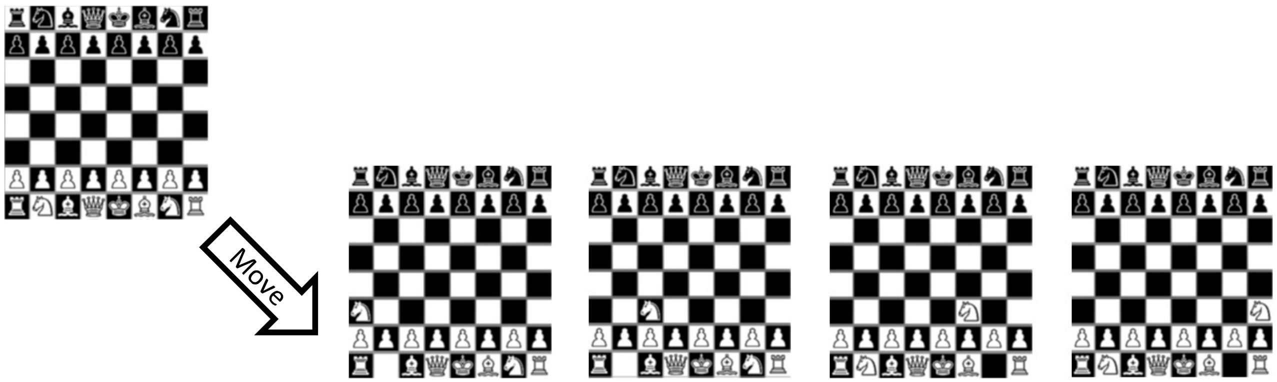
Tic Tac Toe: Solution



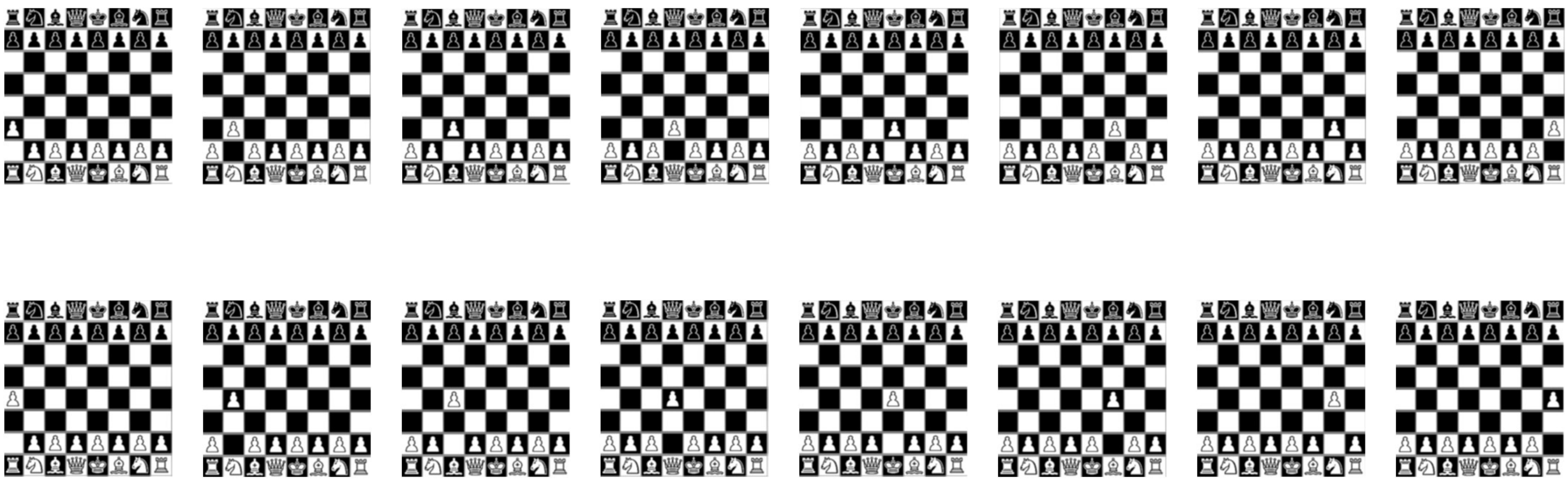
A **Solution** is a sequence of actions (a path) between the initial state and the goal state

Chess: (First Move) State Space

Initial State



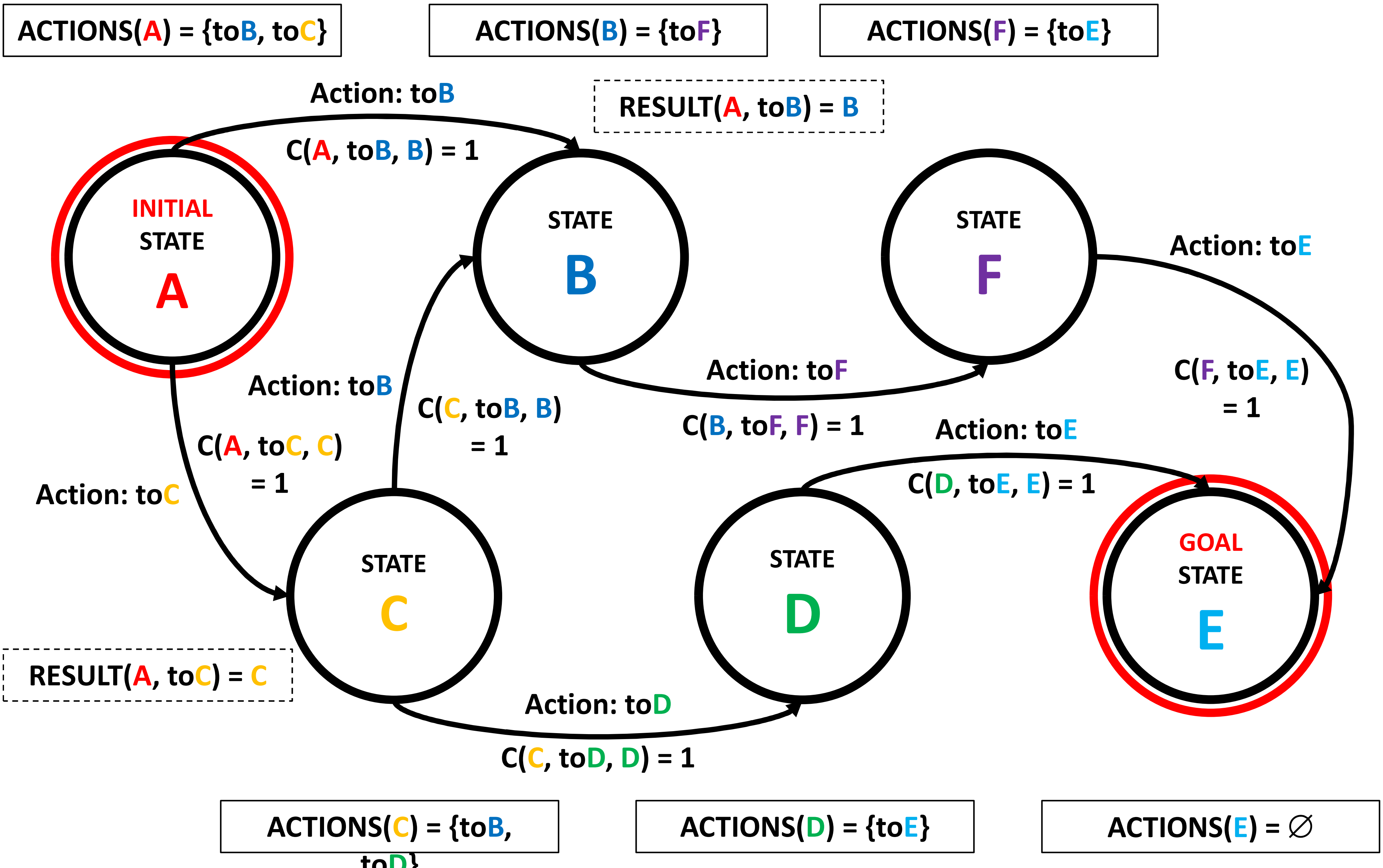
20 Possible legal <u>first</u> moves:
16 pawn moves
4 knight moves



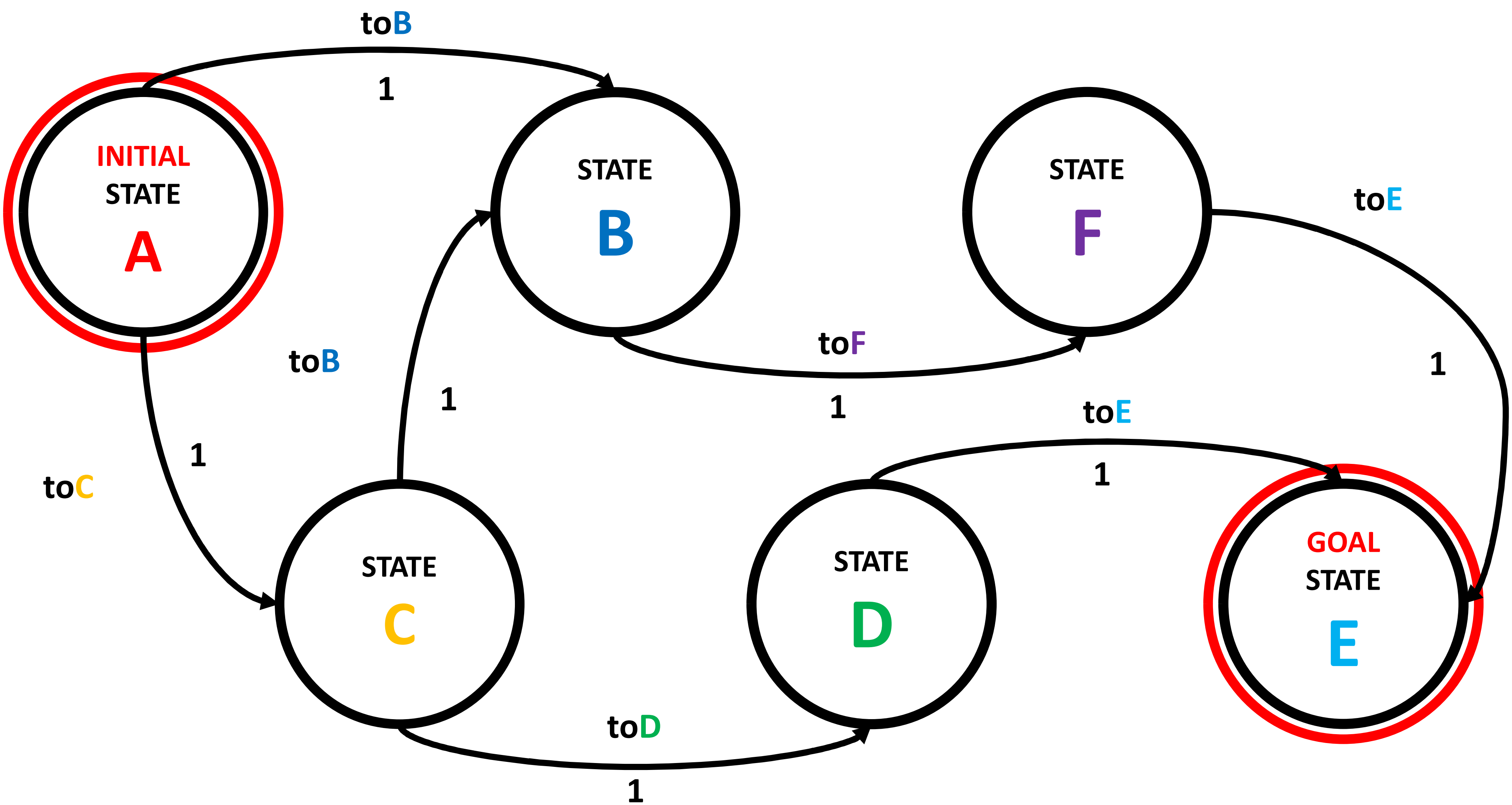
Designing the Searching Problem

Analyze and define the Problem / Task	Model and buid the State Space	Select searching algorithm	Search

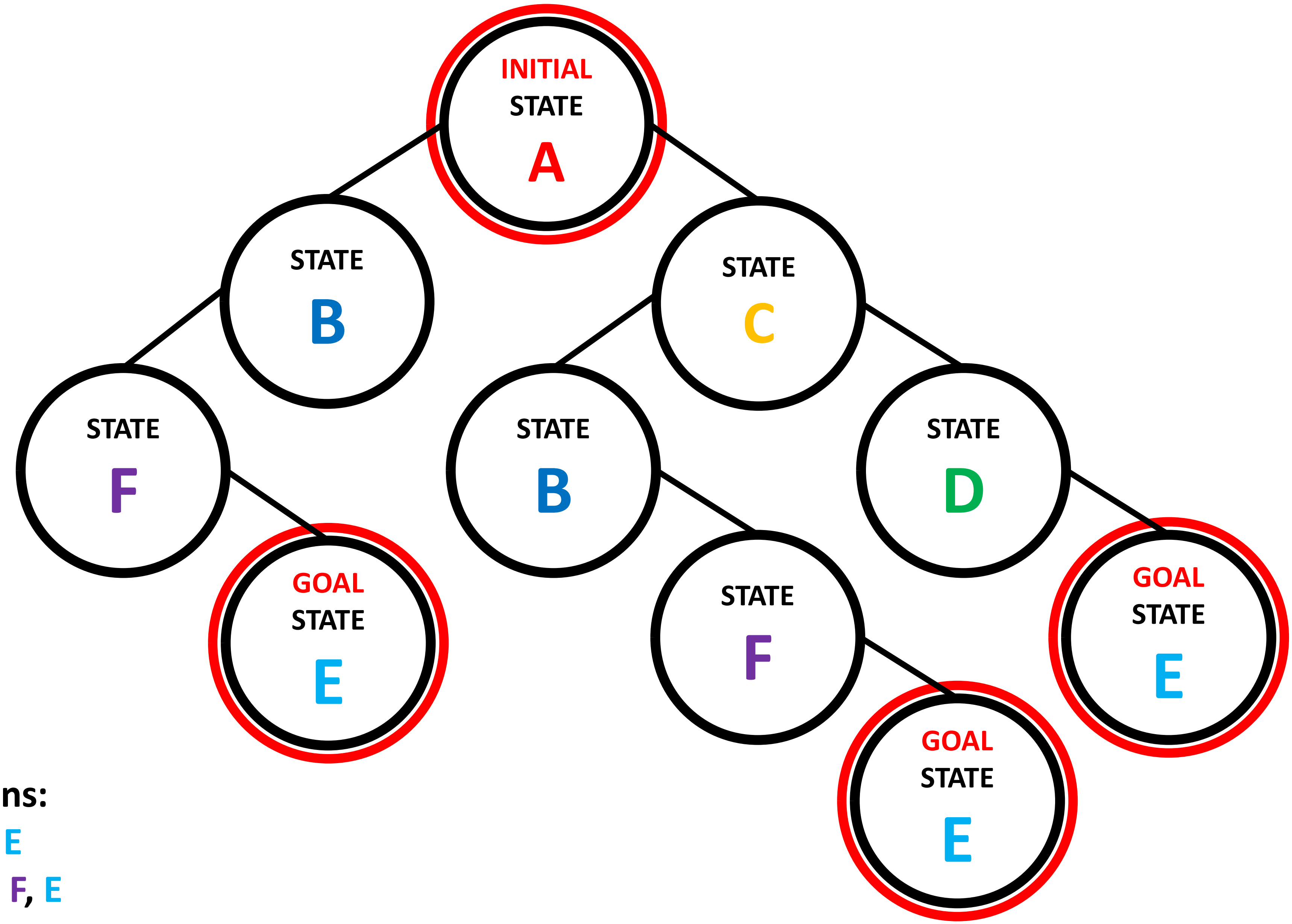
State Space Model: A Graph



State Space Model: A Graph



Searching State Space: Search Tree



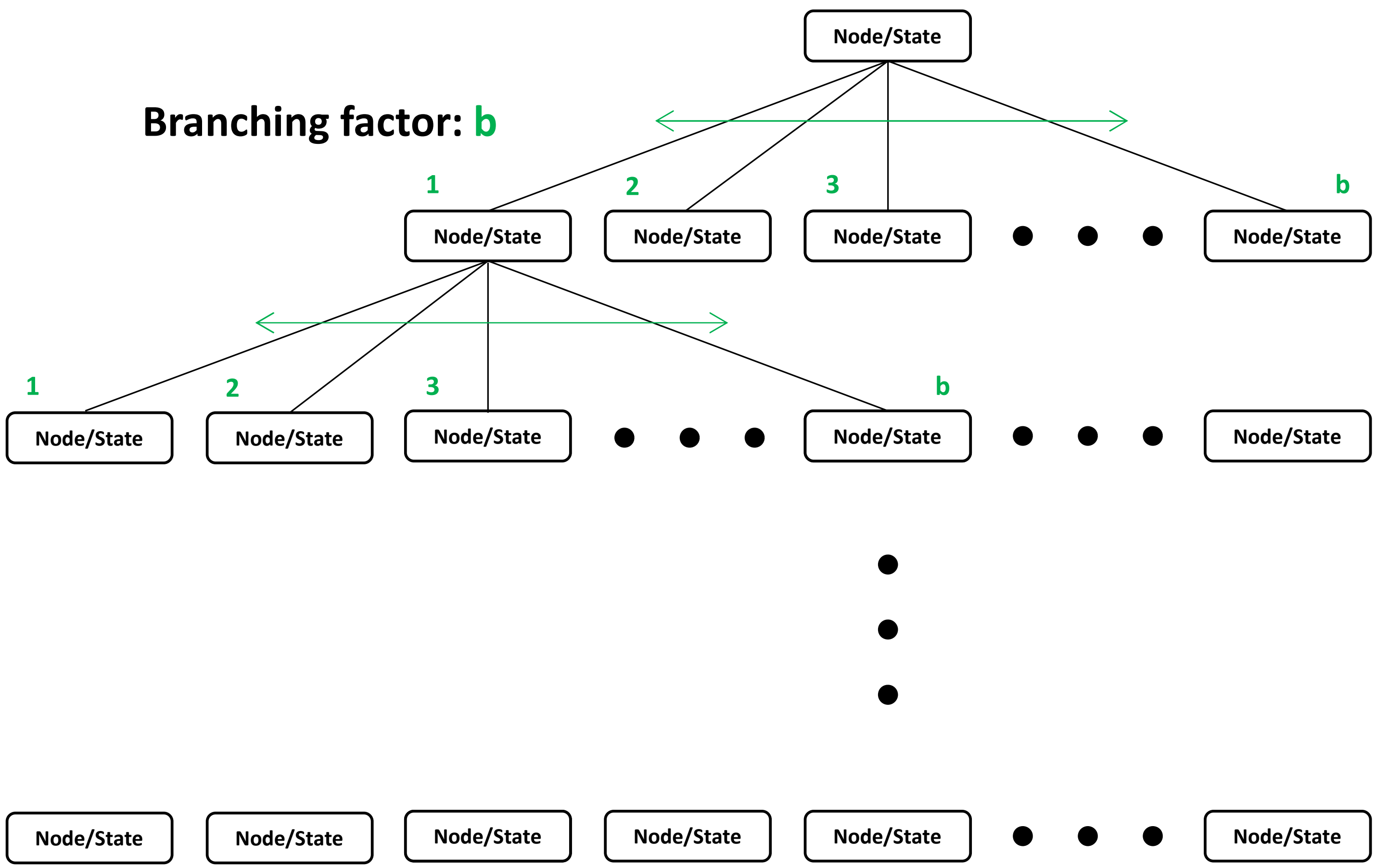
Solutions:

A, B, F, E

A, C, B, F, E

A, C, D, E

Search Tree Challenges: Size



Depth: 0 | $N_0 = 1$

Depth: 1 | $N_1 = b$

Depth: 2 | $N_2 = b^2$

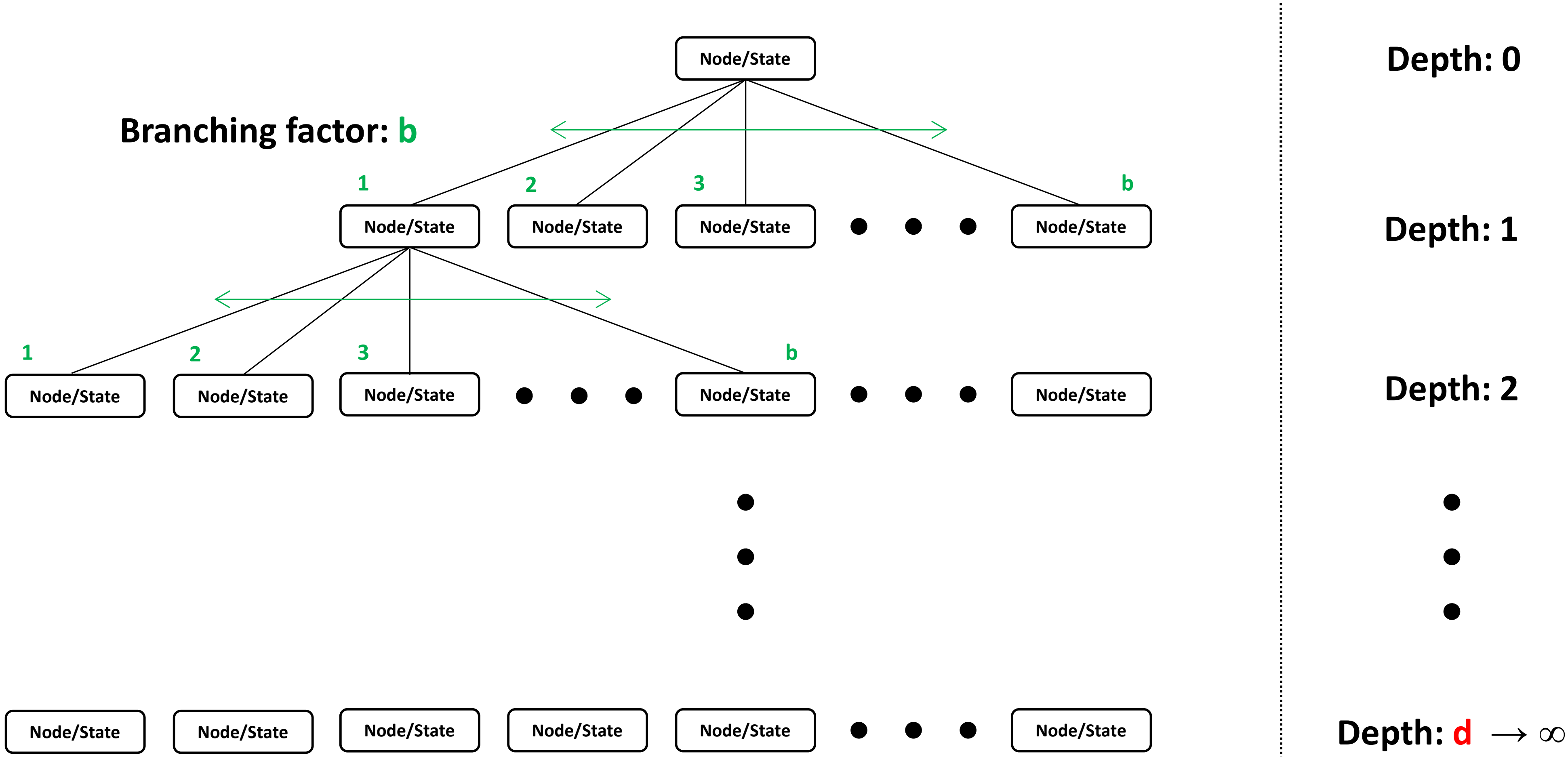
…

Depth: d | $N_d = b^d$

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!

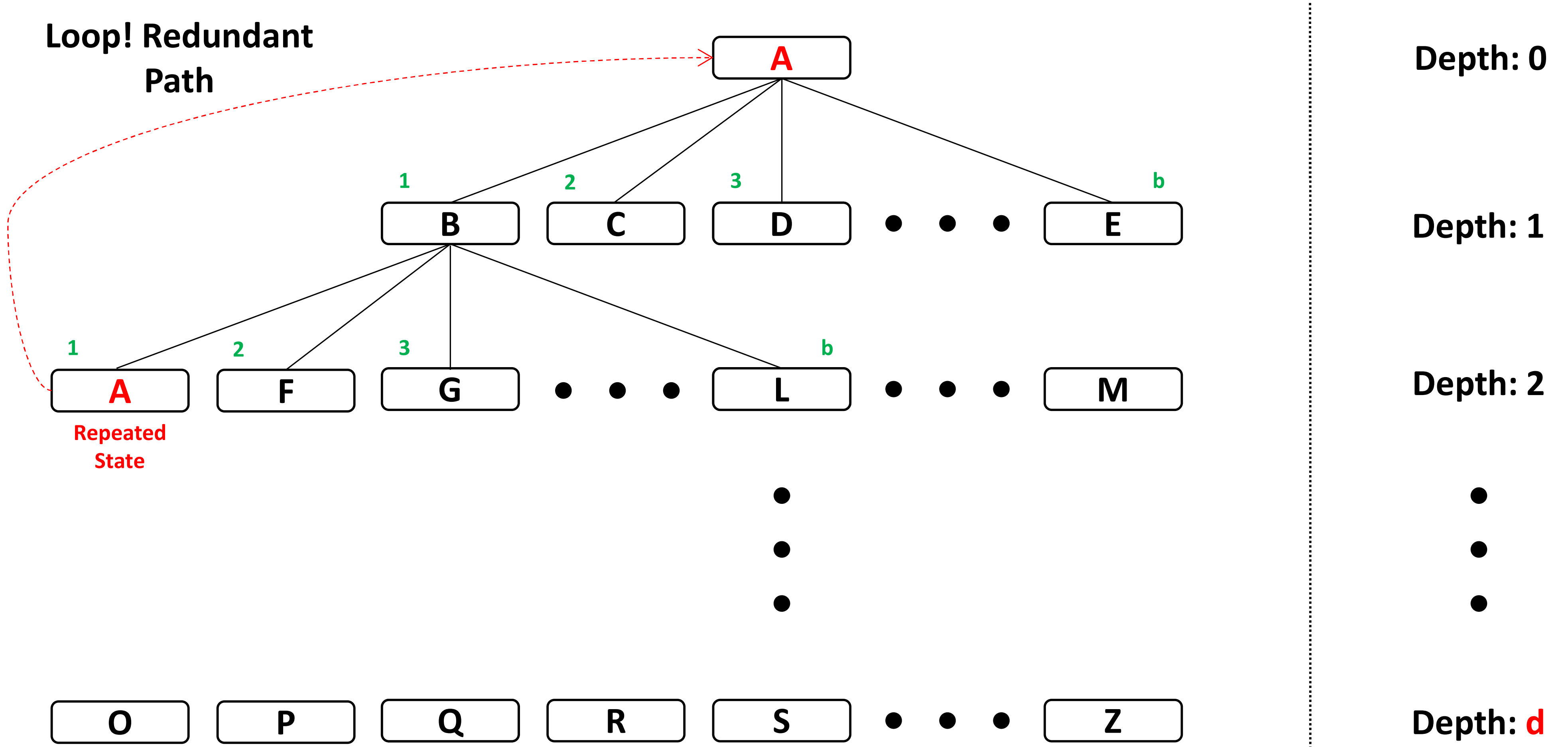
Search Tree Challenges: Infiniteness



Unmanageable and impossible to search with brute force!

Memory and time use grows quickly!

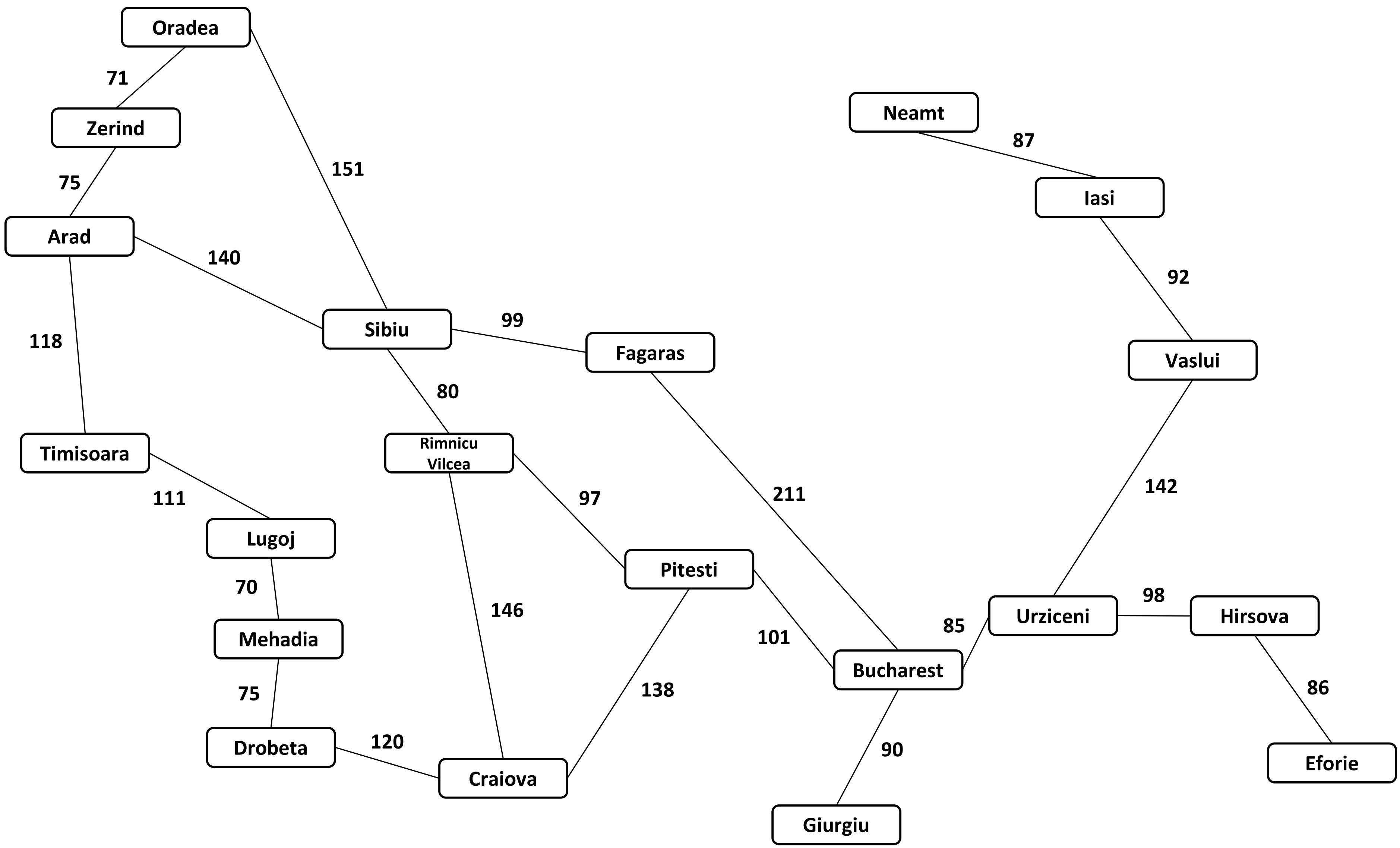
Search Tree Challenges: Loops



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

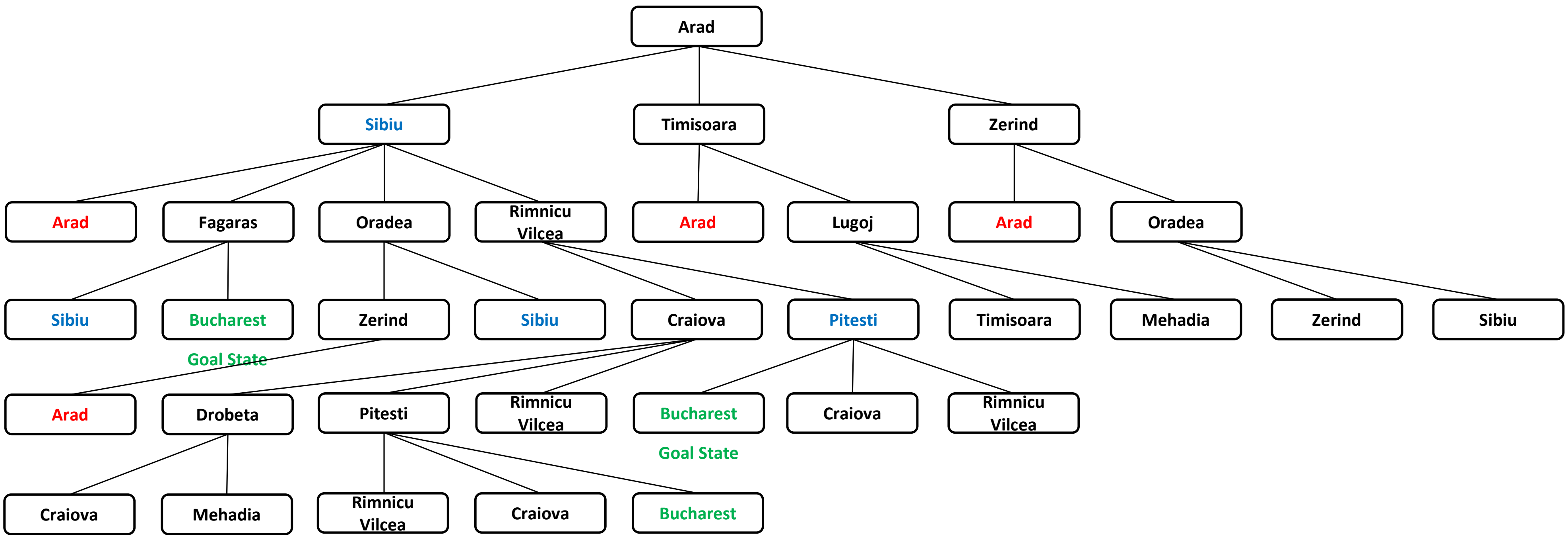
Memory and time use grows quickly!

Sample Problem: Dracula's Roadtrip



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

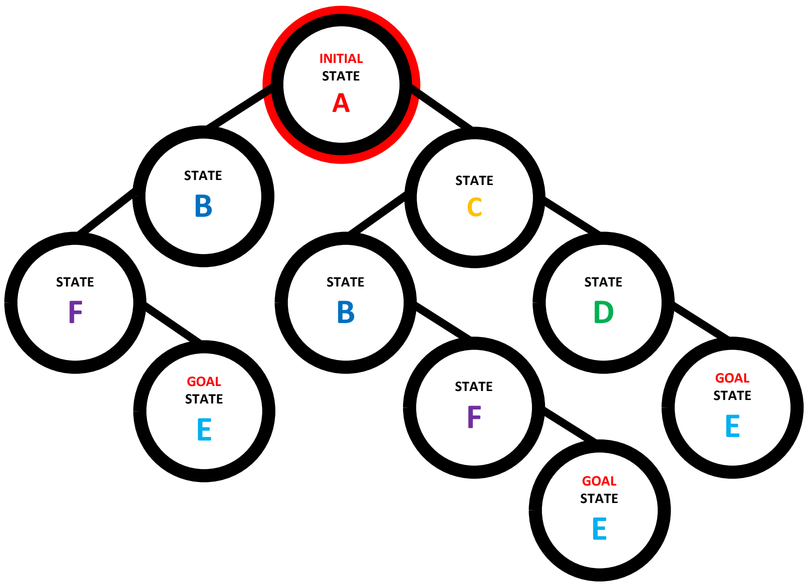
Romanian Roadtrip as a Tree



INCOMPLETE! I need to redraw it in smarter way

Search Tree: Implementations

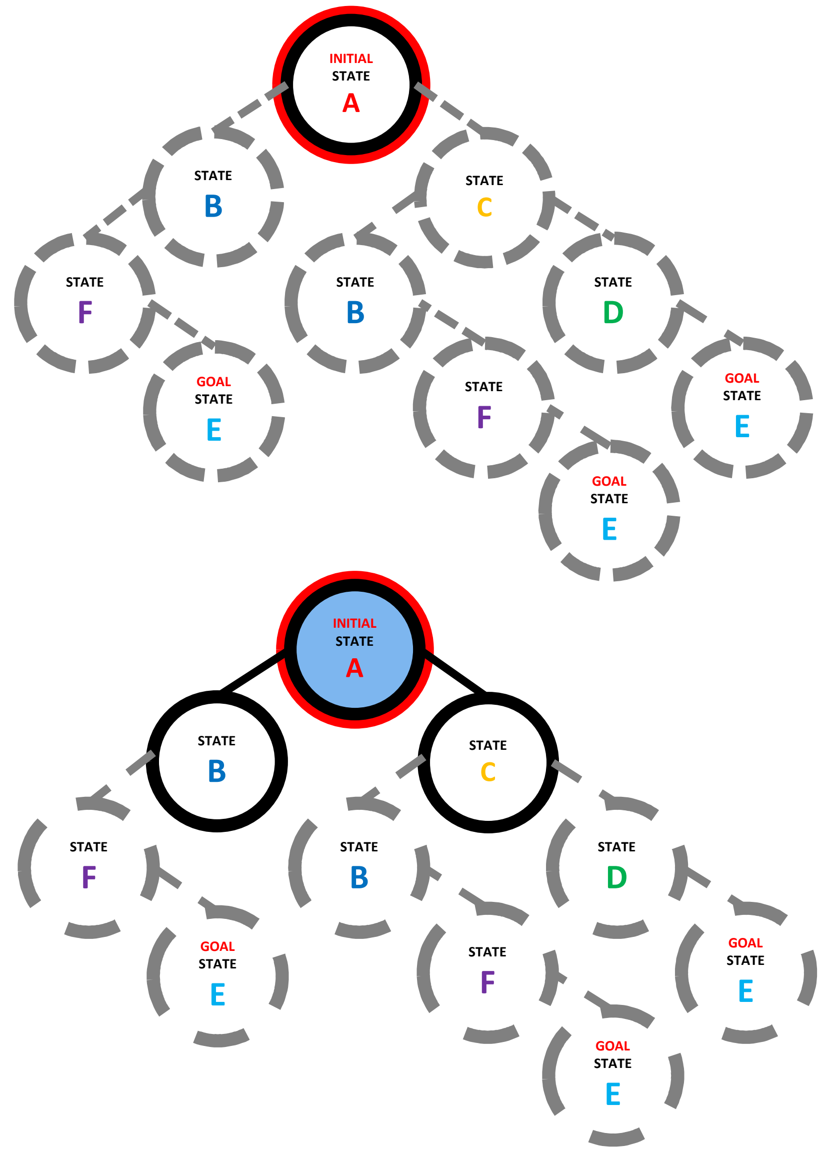
Build entire search tree



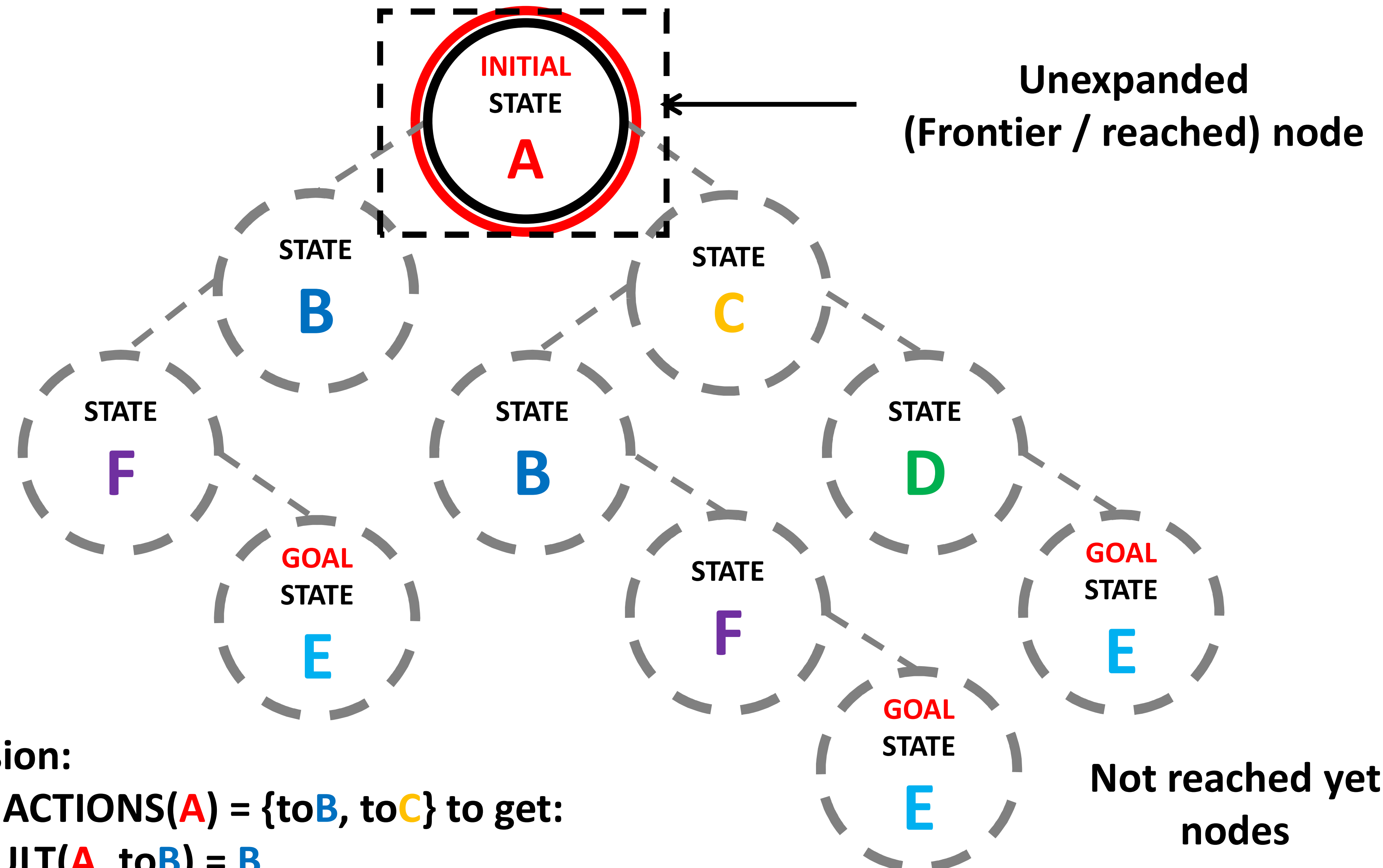
Challenges:

- memory requirements
- impossible for infinite number of states

Expand/generate nodes as you go



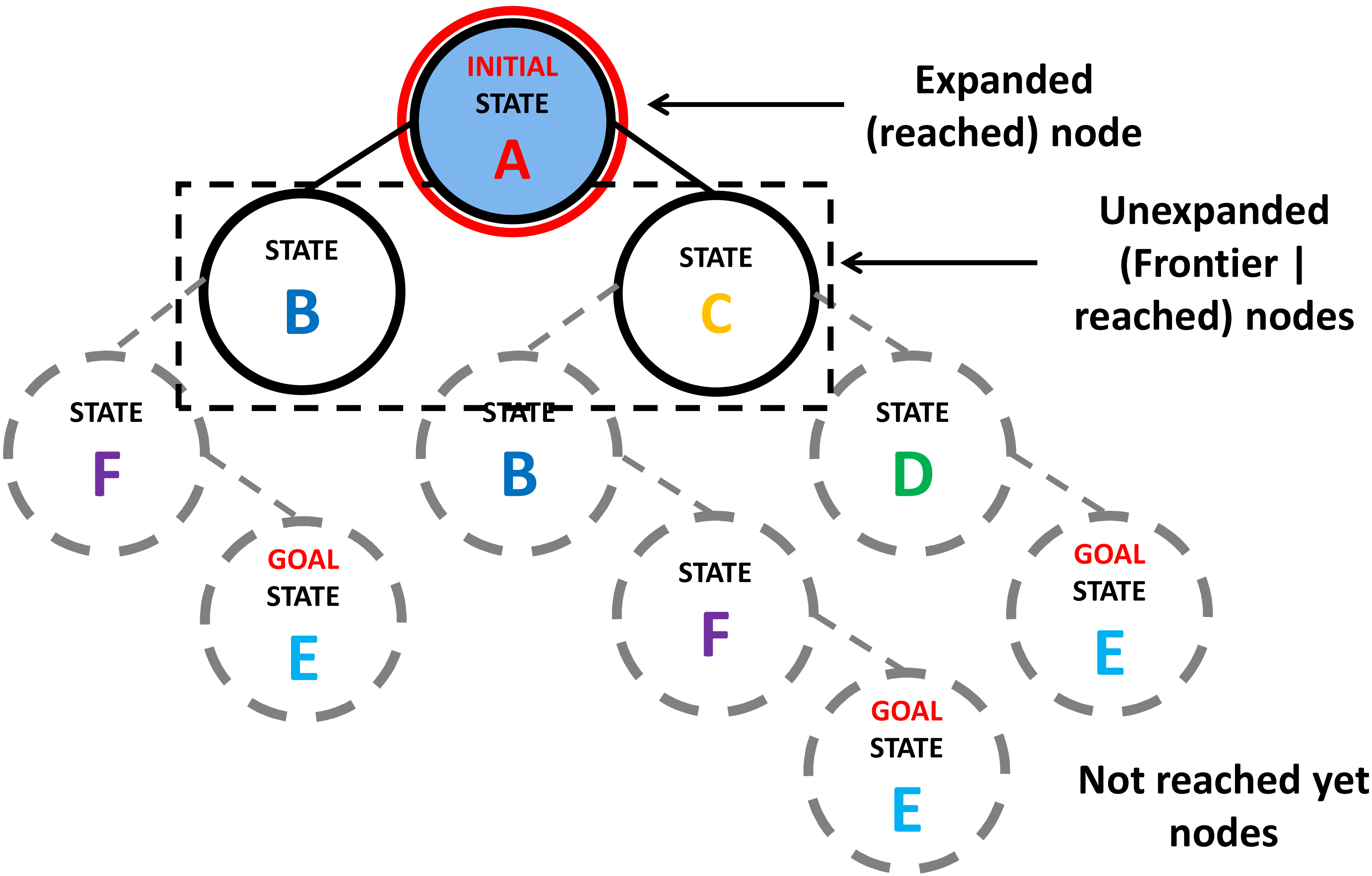
Search Tree: Node Expansion



Expansion:

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

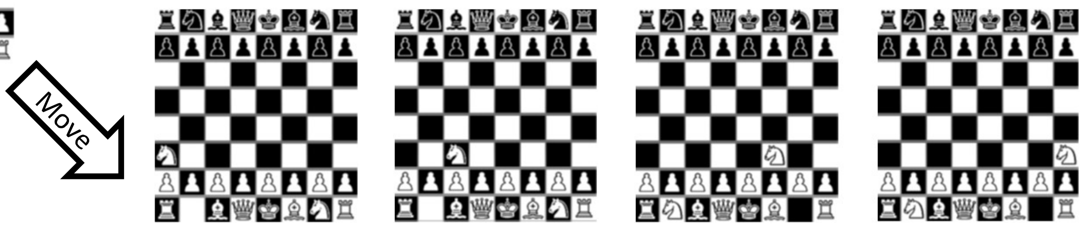
Search Tree: Node Expansion



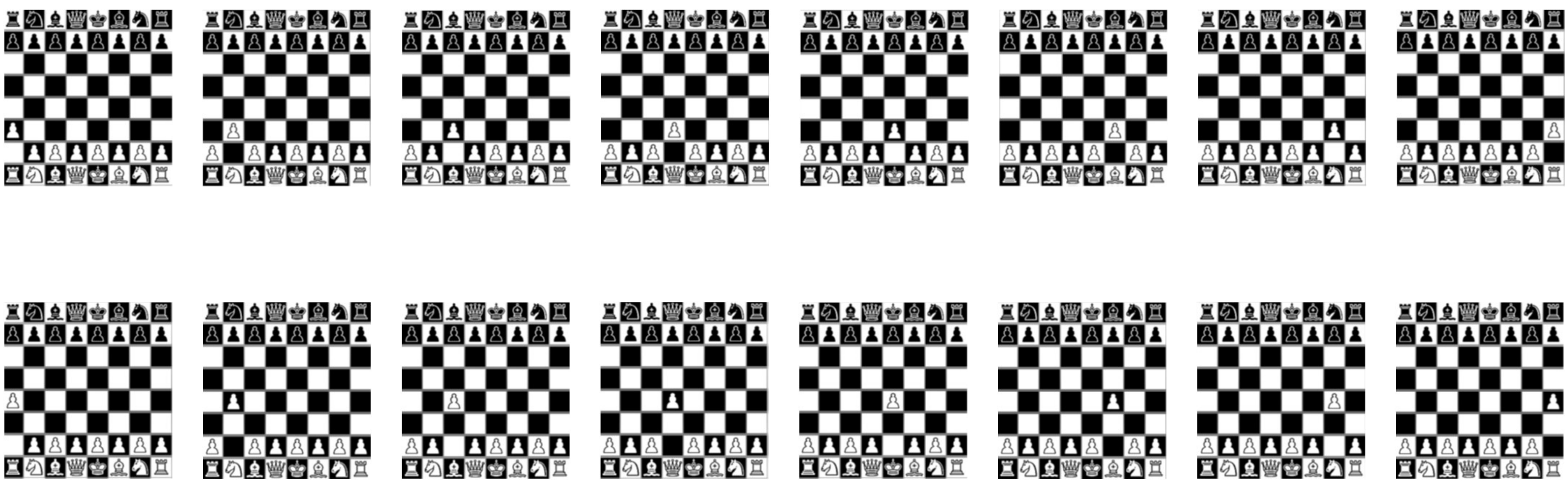
Chess: State Node Expansion

Initial State

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

Analyze and define the Problem / Task

Model and buid the State Space

Select searching algorithm

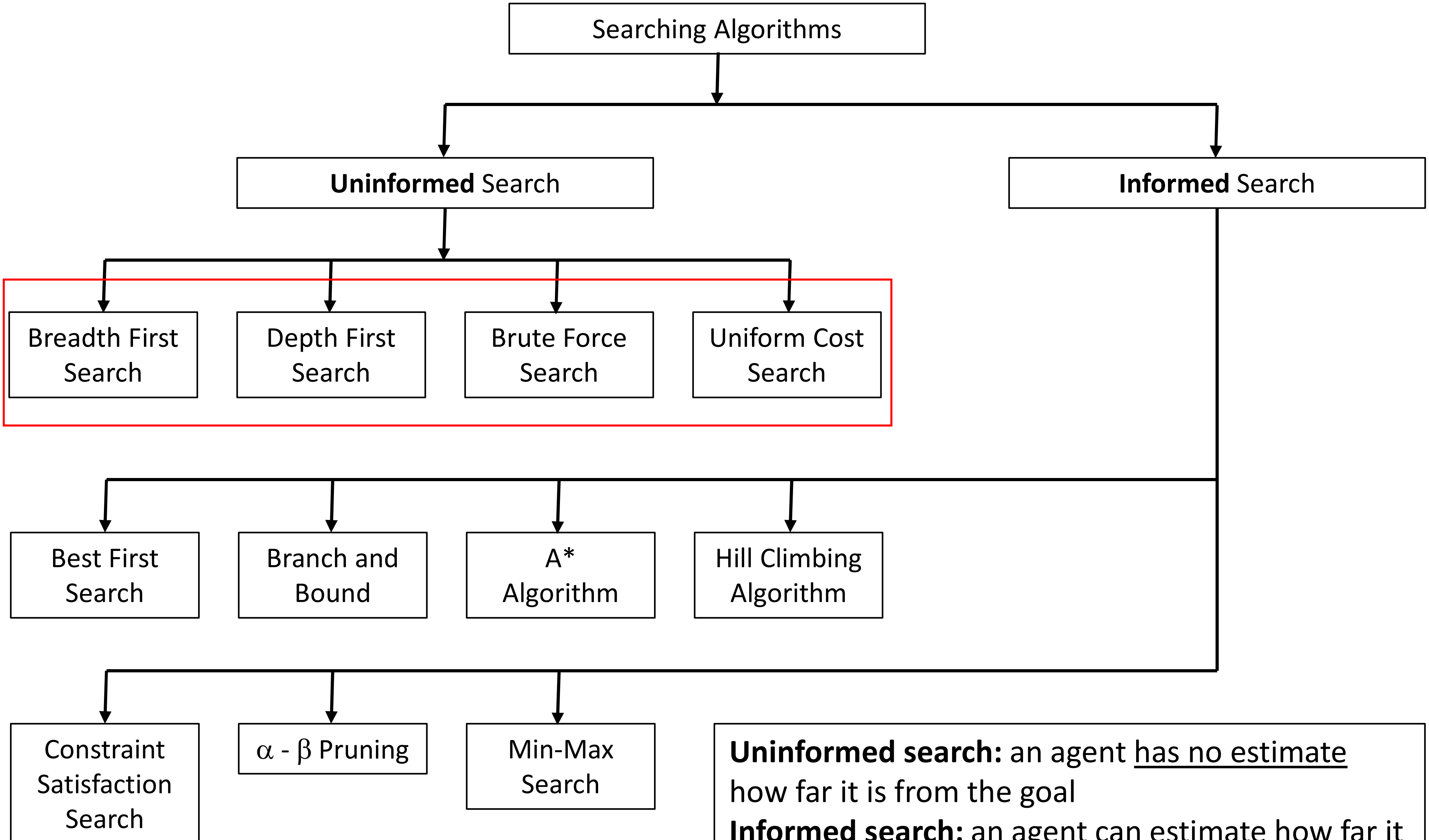
Search

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

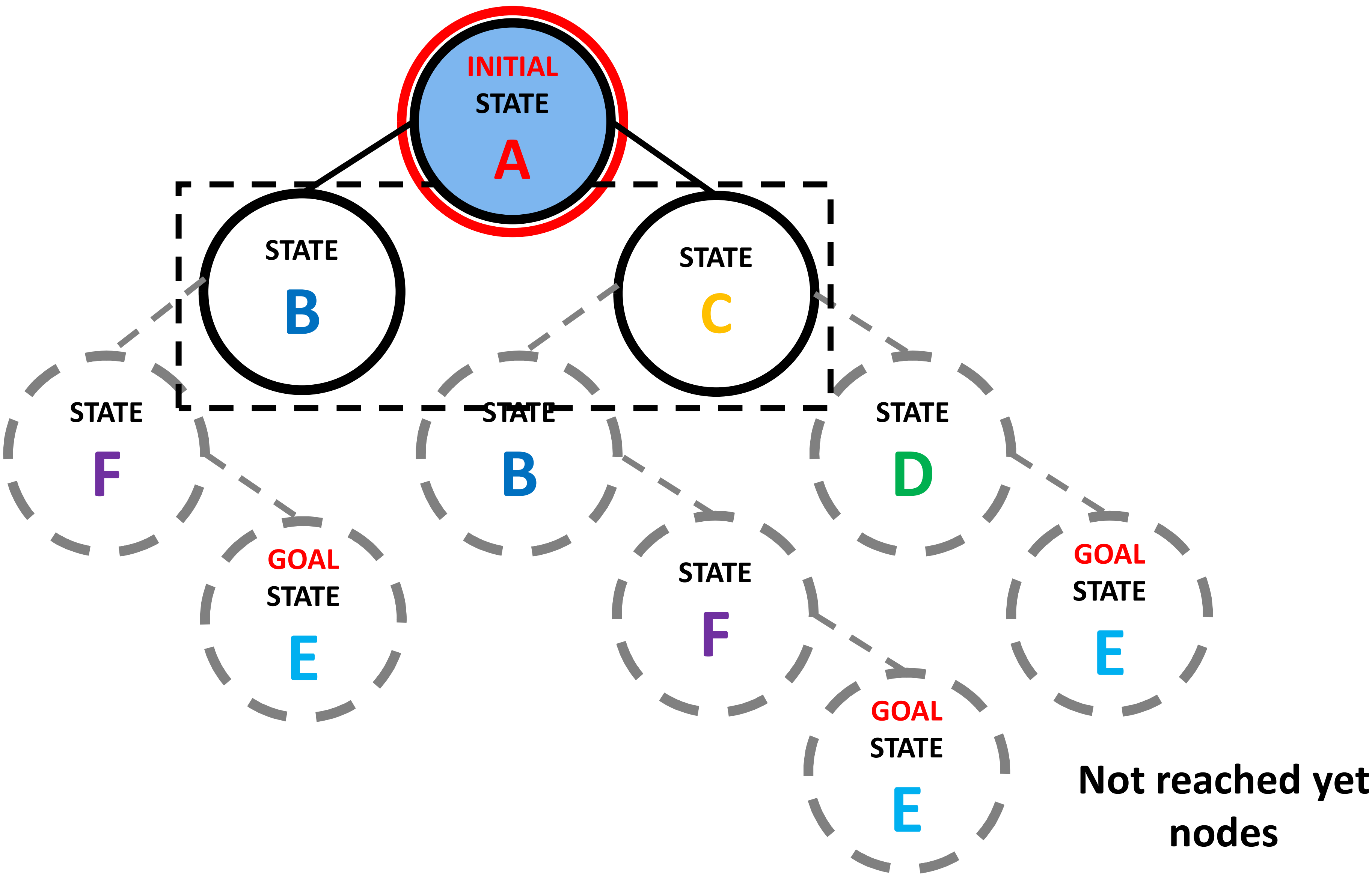


Uninformed search: an agent <u>has no estimate</u> how far it is from the goal
Informed search: an agent <u>can estimate</u> how far it is from the goal

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?



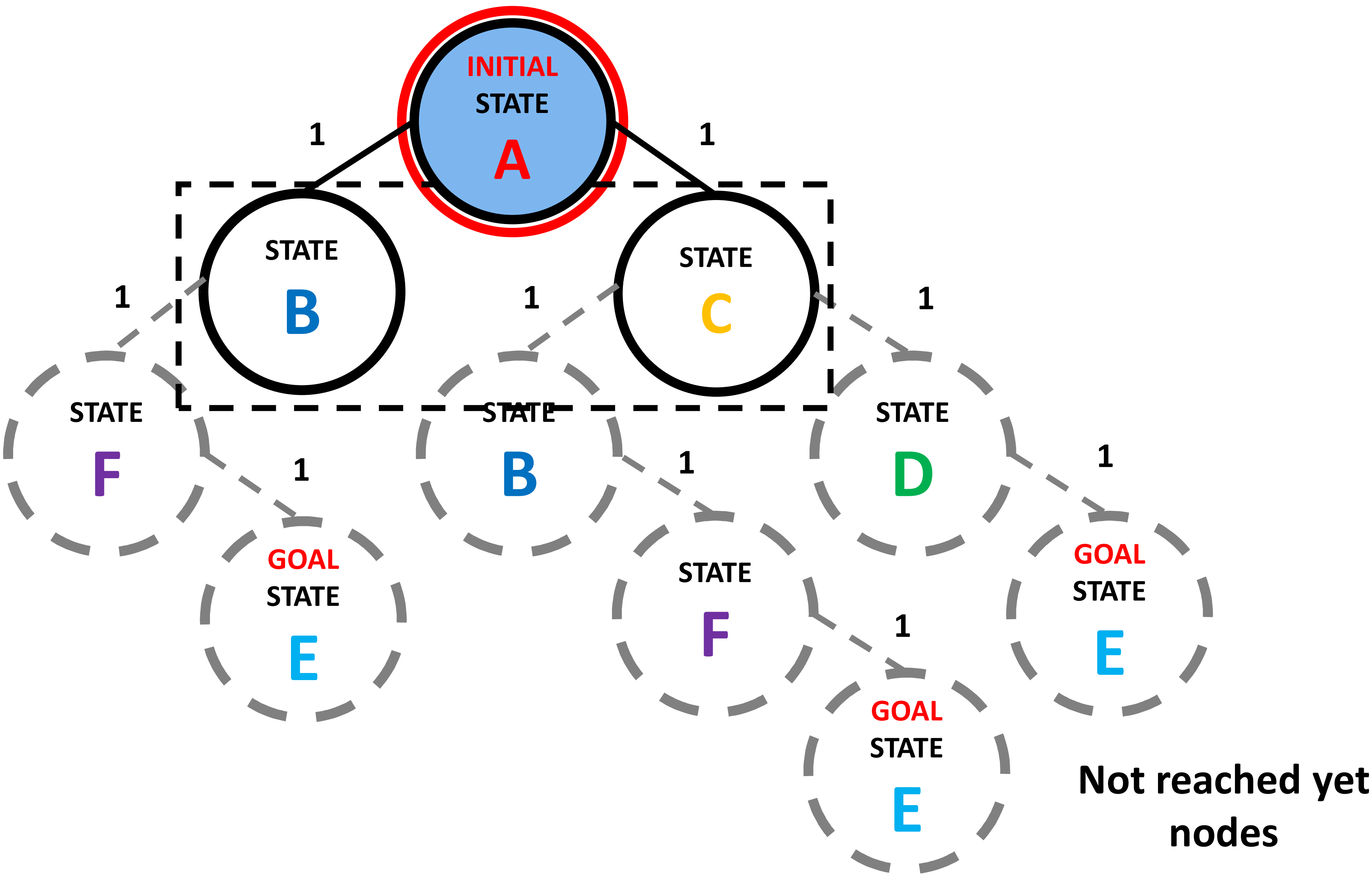
Evaluation function

Calculate / obtain:

$f(n) = f(\text{State } n)$
$f(n) = f(\text{relevant information about State } n)$

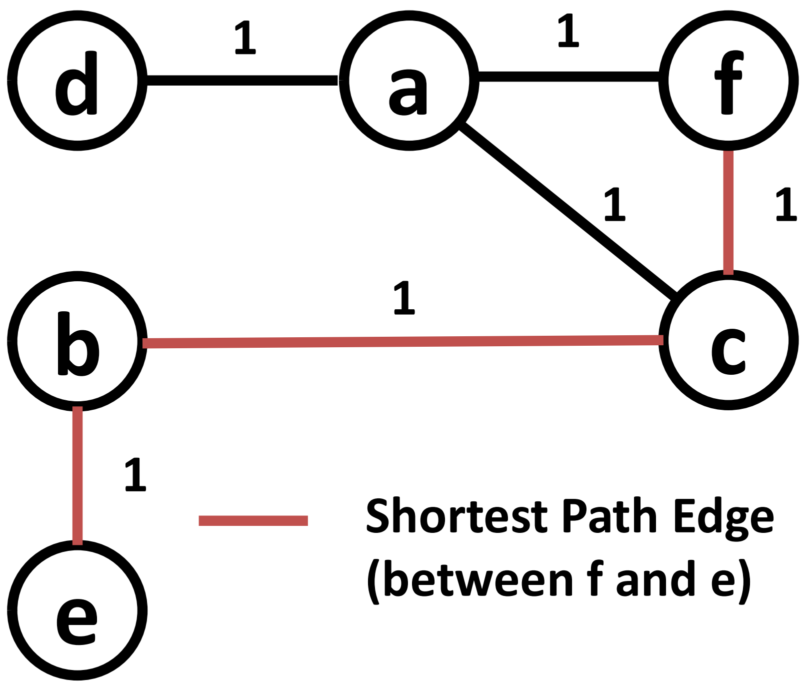
A state n with minimum f(n) should be chosen for expansion
What about ties?

Search Tree: Uniform Action Cost



Uniform Cost Search | Dijkstra's Algo

Weighted Graph G



Popular algorithms:

- Dijkstra's algorithm

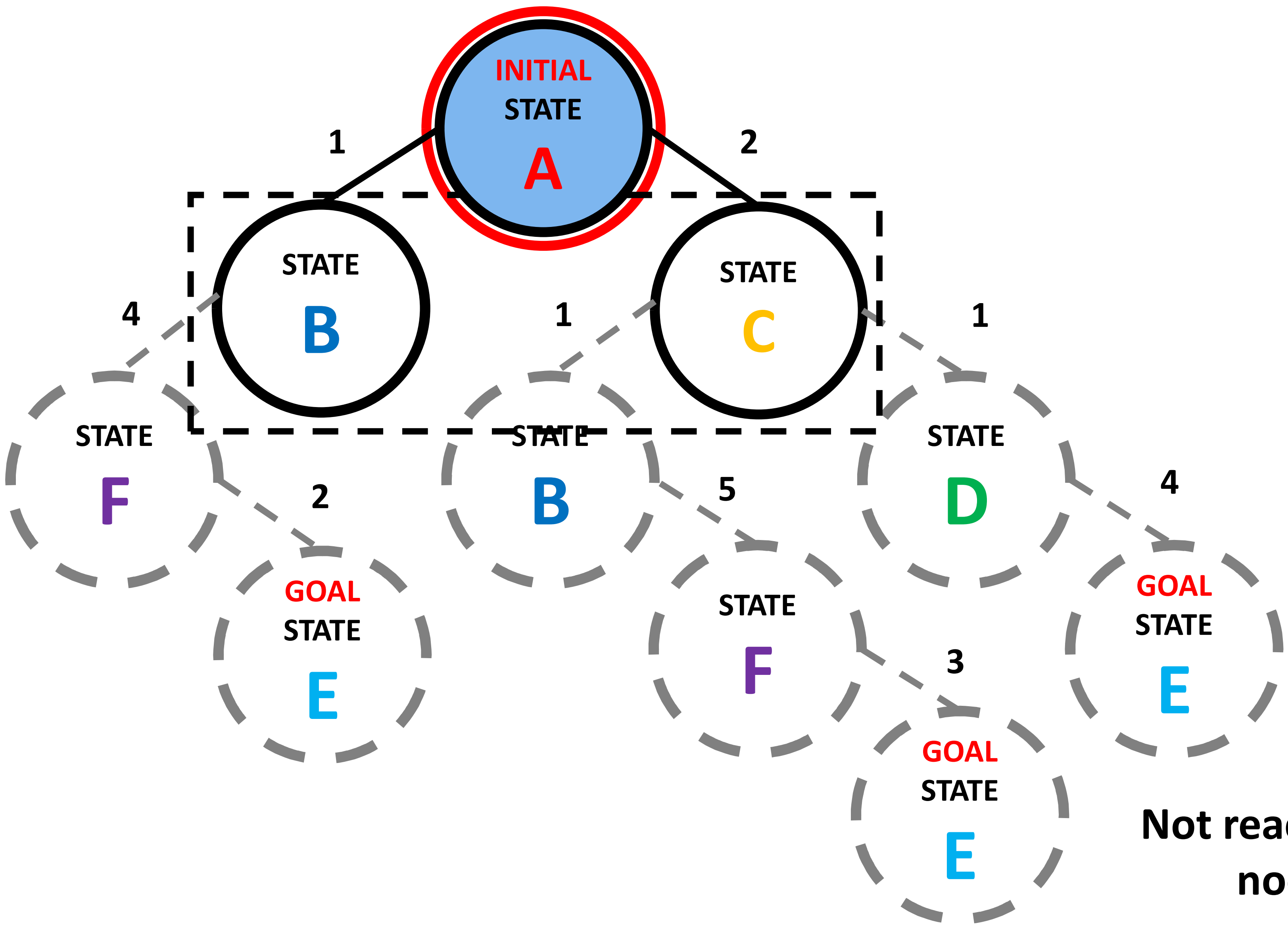
Shortest Path Problem

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

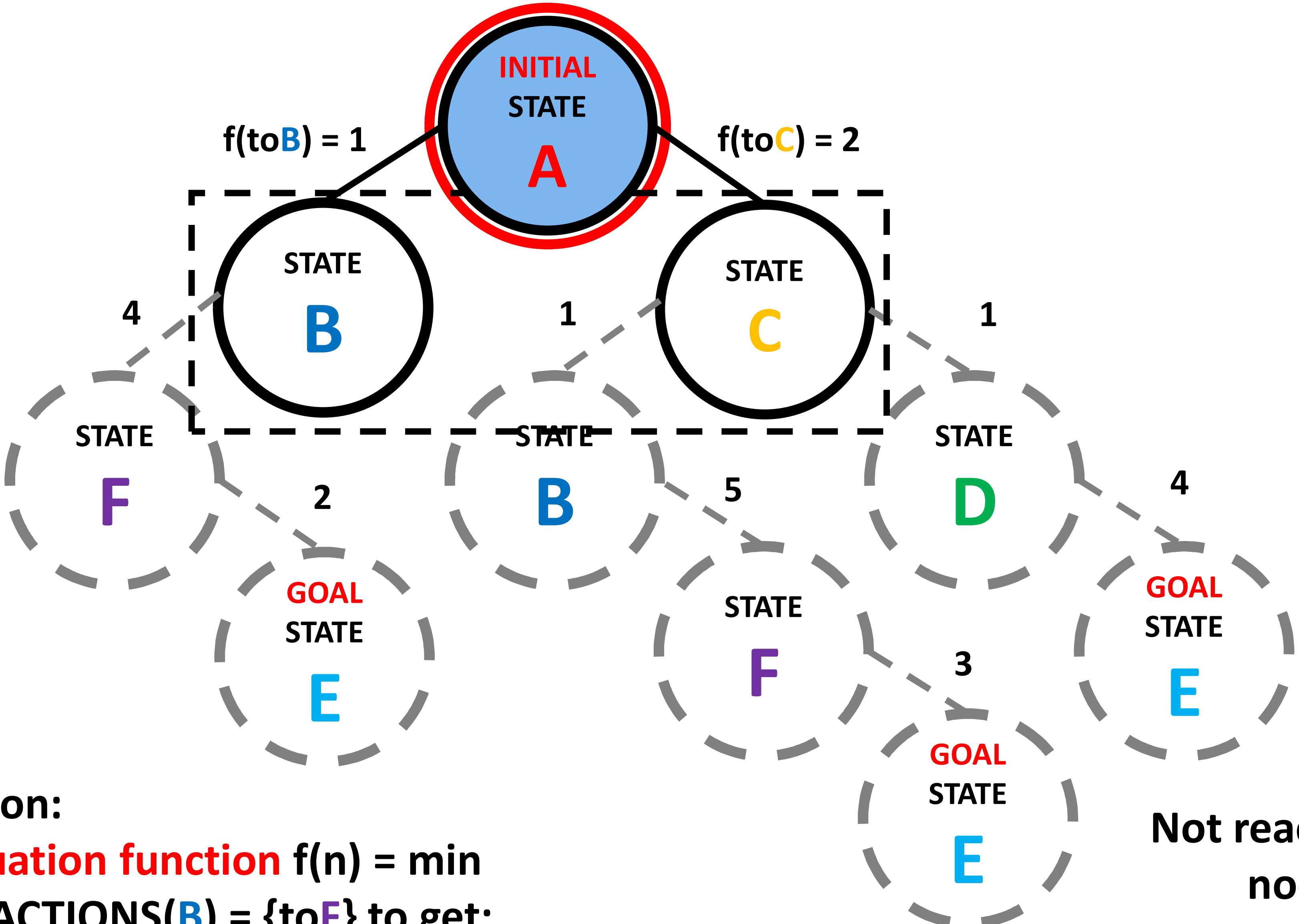
```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
 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



Not reached yet nodes

Search Tree: Variable Action Cost

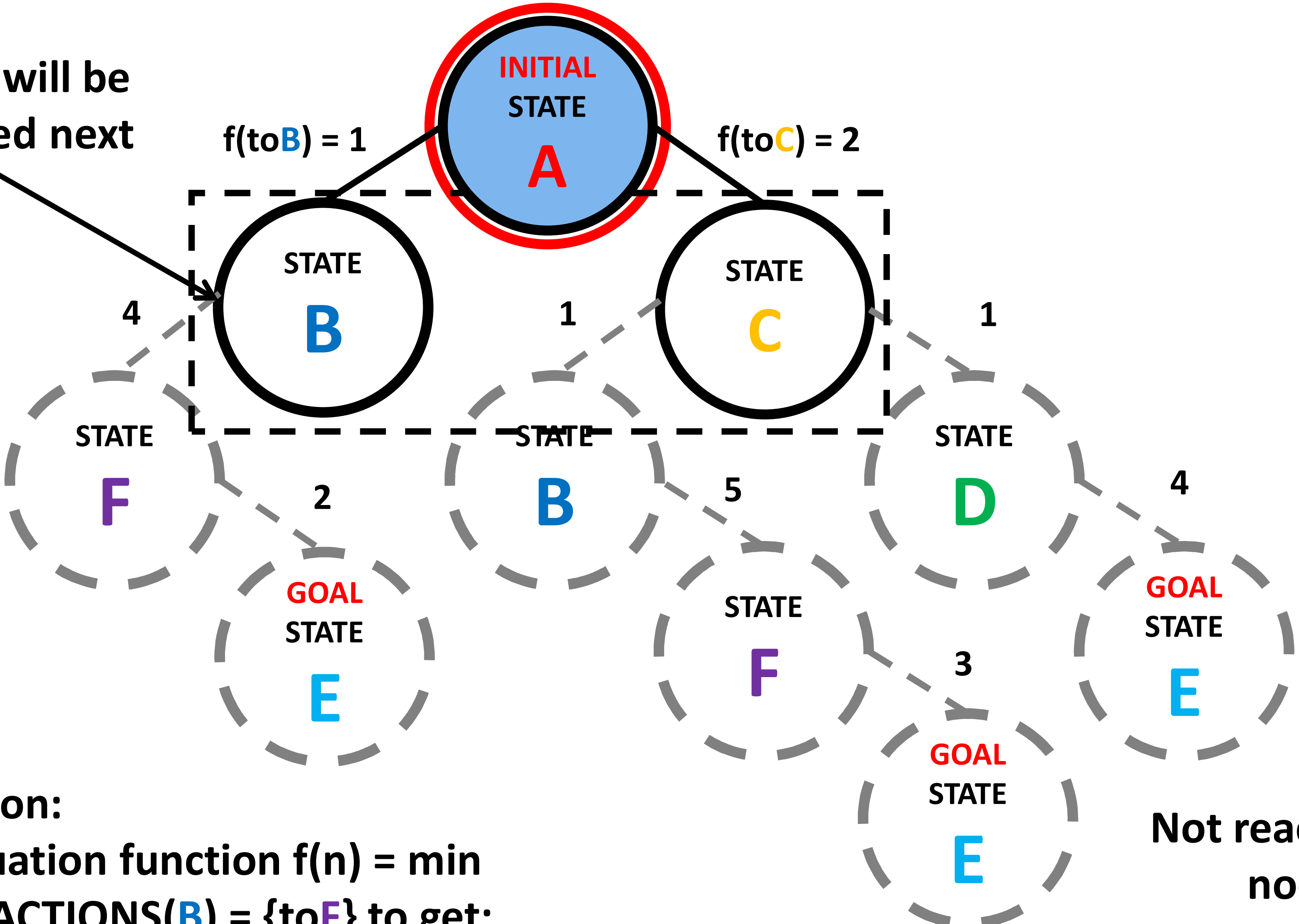


Expansion:

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

Search Tree: Best-First Search

Node B will be expanded next



Not reached yet nodes

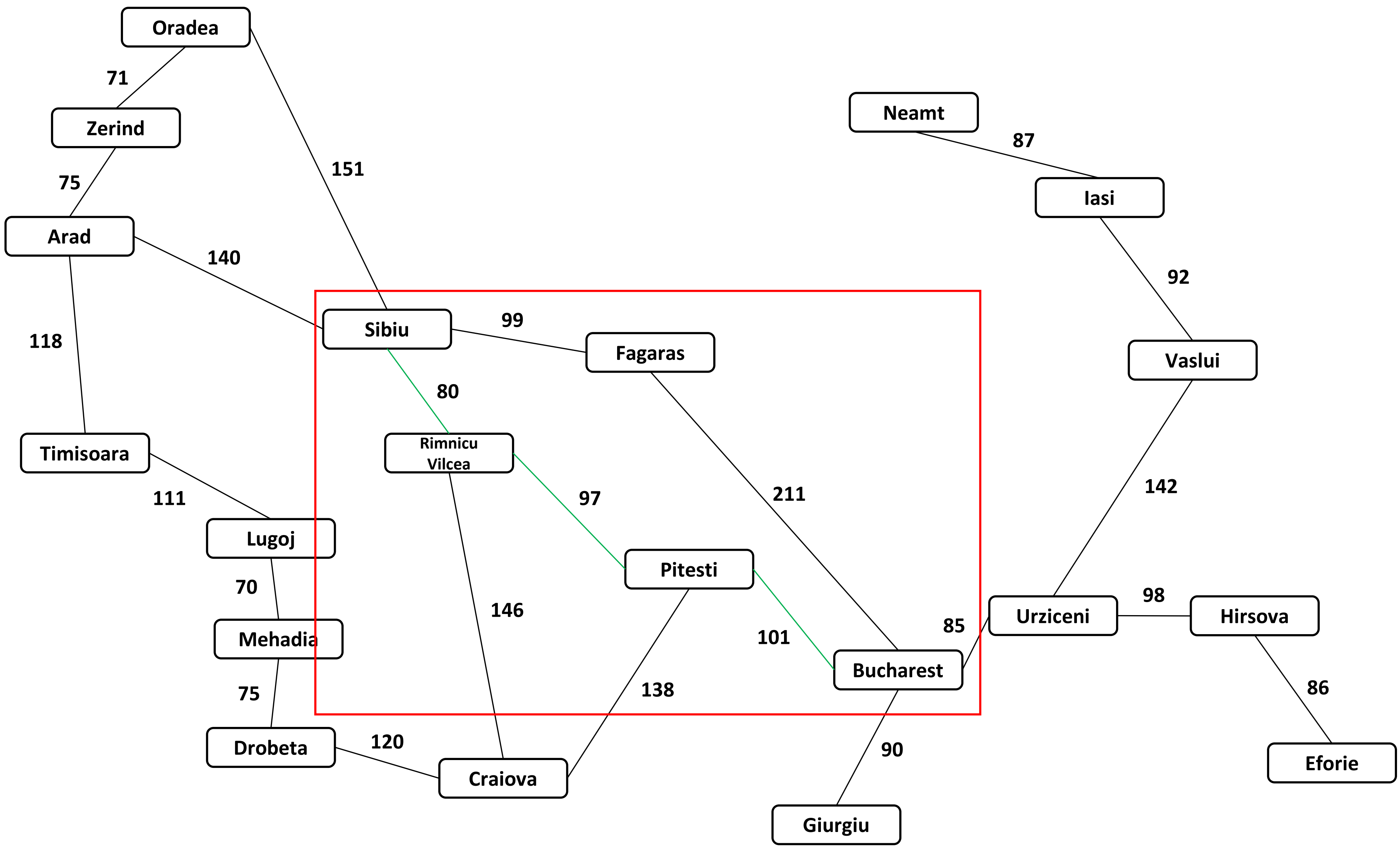
Expansion:

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

Best-First Search: Pseudocode

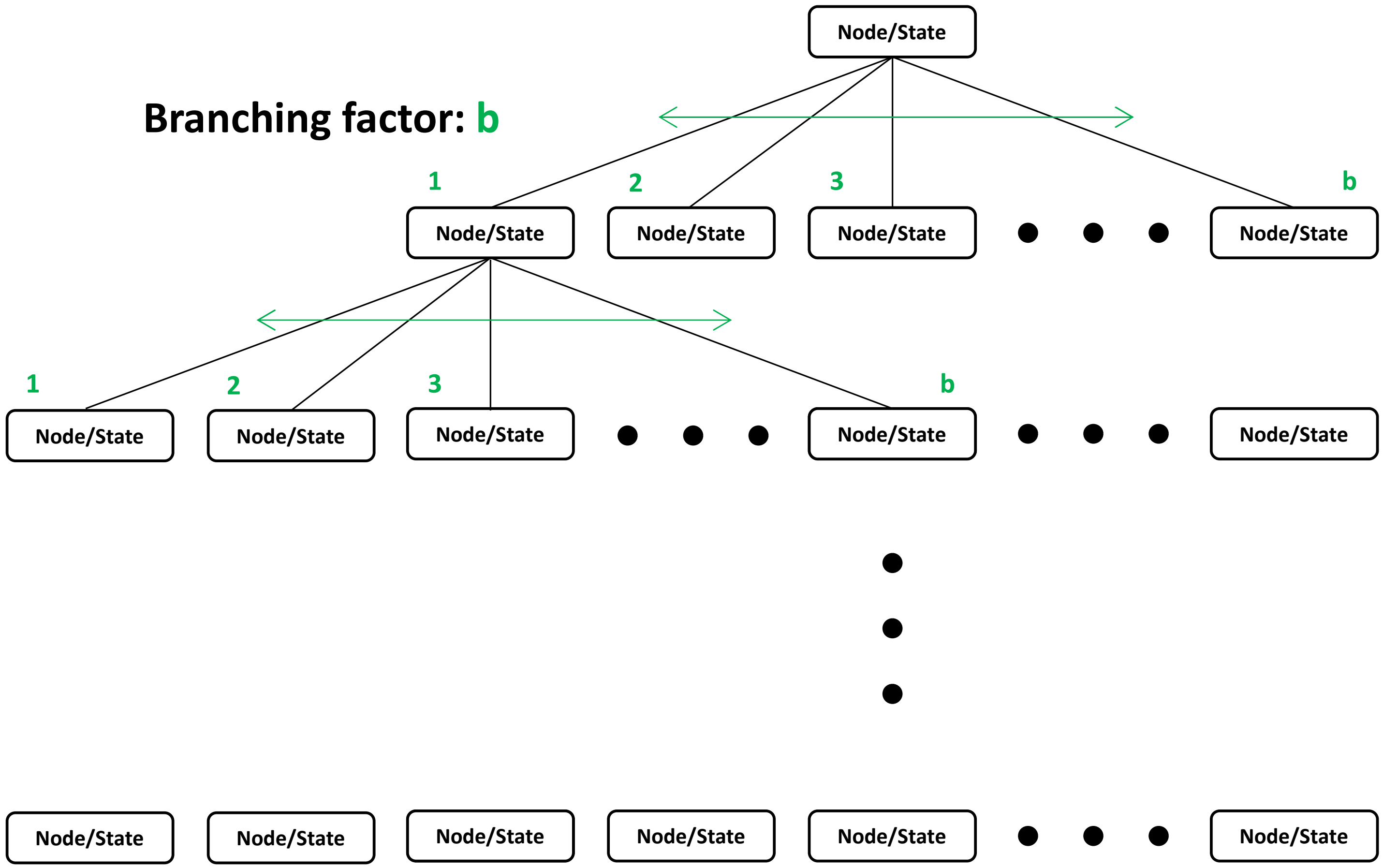
```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
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

Branching factor: b



Depth: 0 | $N_0 = 1$

Depth: 1 | $N_1 = b$

Depth: 2 | $N_2 = b^2$

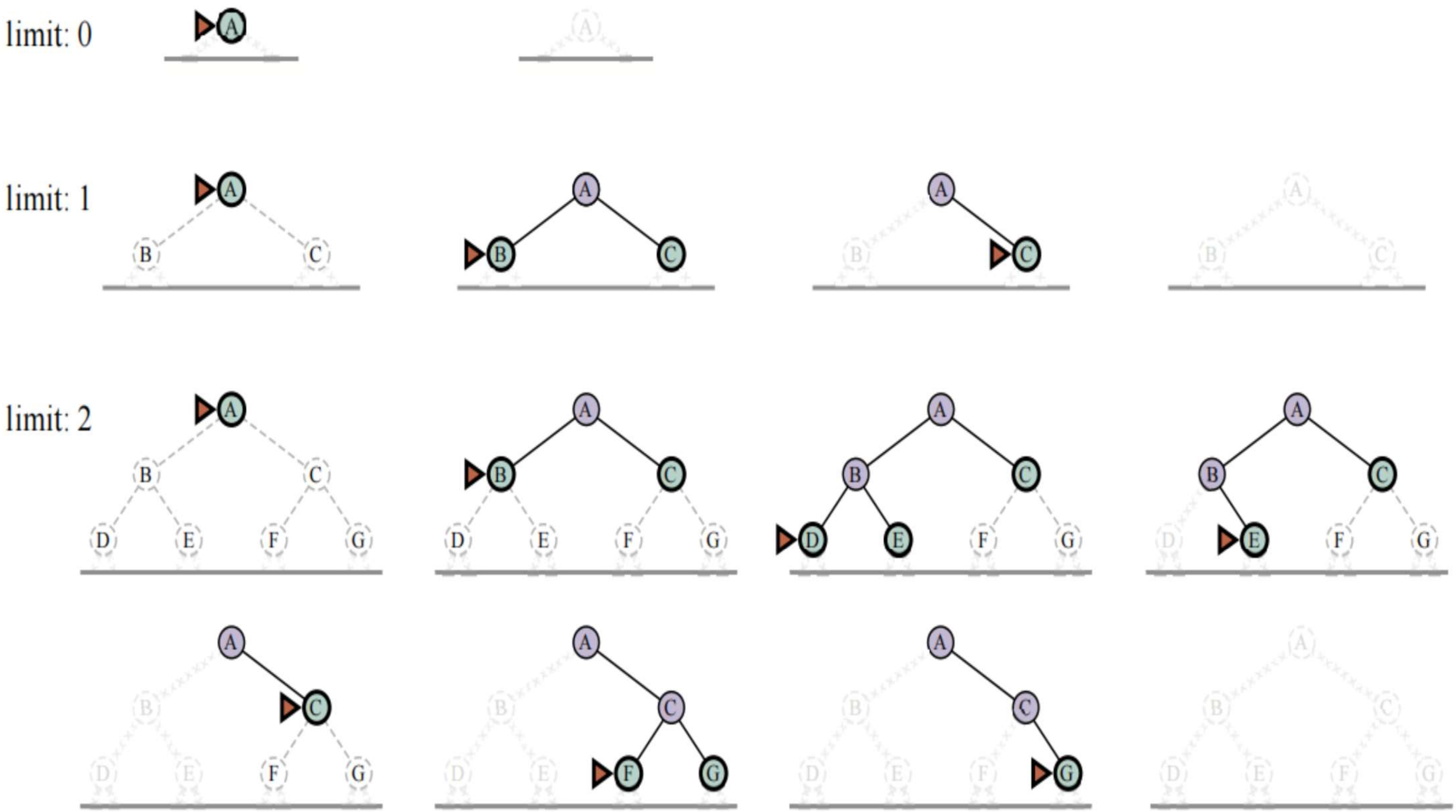
Depth: d | $N_d = b^d$

Tree depth is an issue!

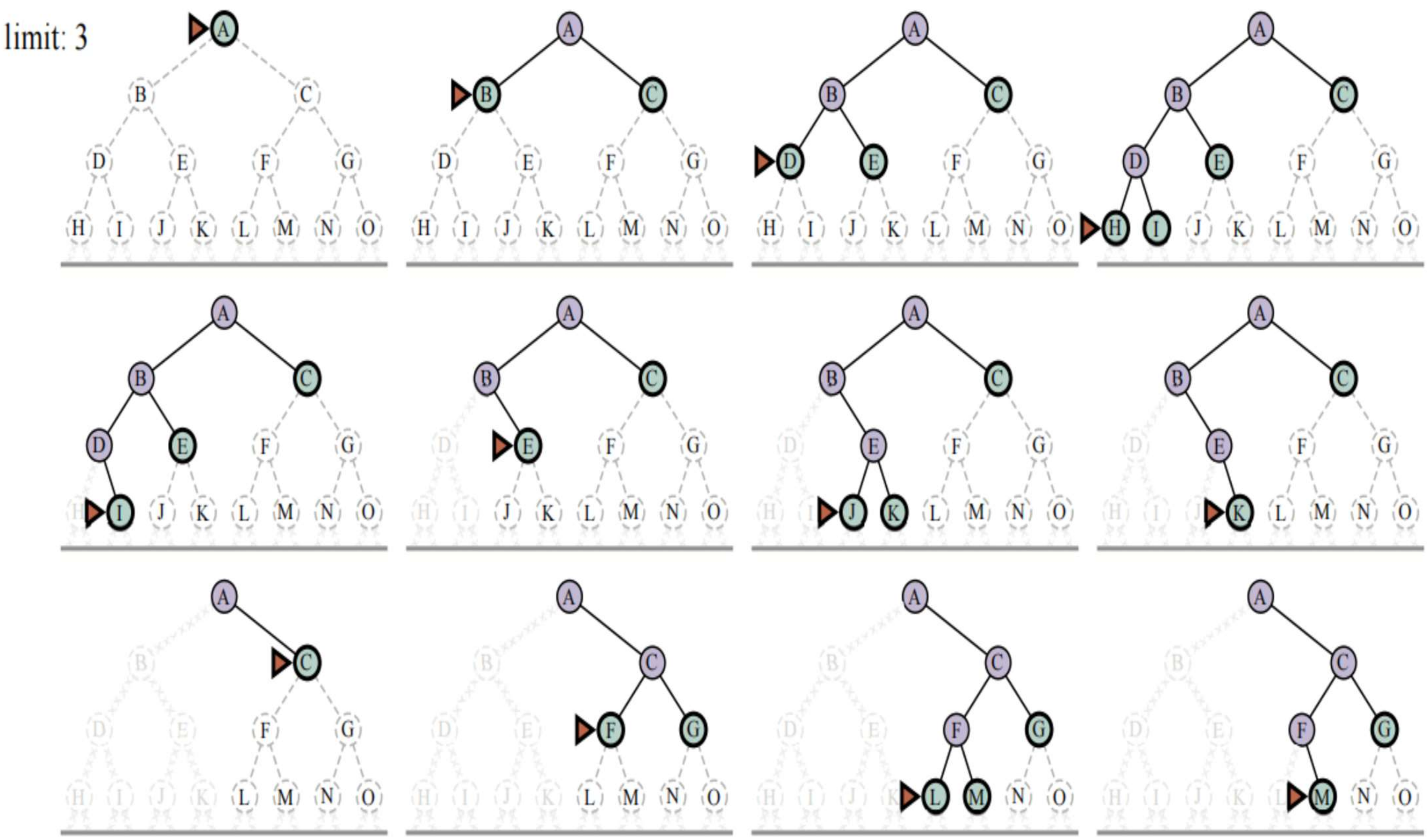
"Controlled" DFS: Pseudocode

```
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

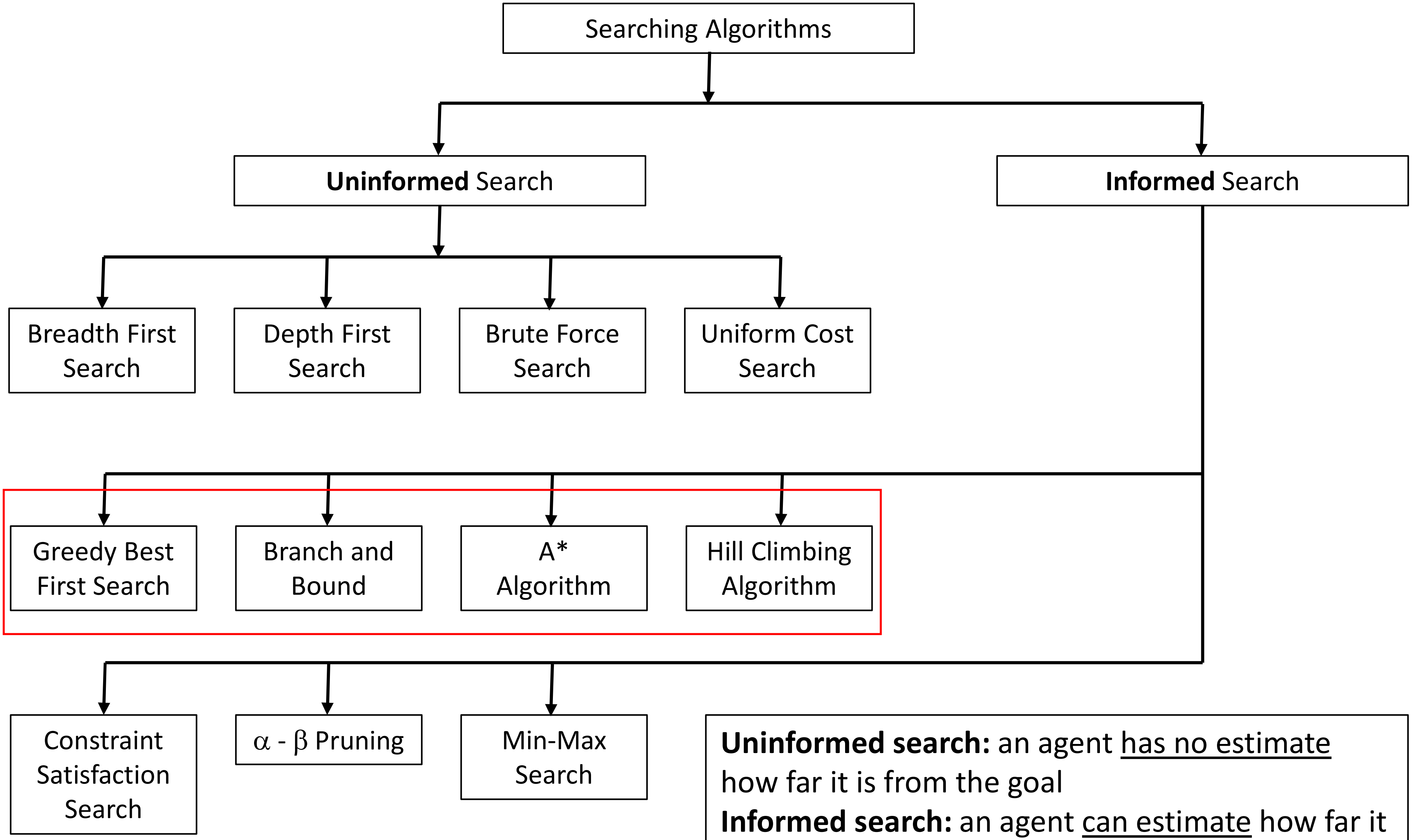


Uninformed Search Algorithms

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