

# CS 480

## *Introduction to Artificial Intelligence*

September 6, 2022

# Announcements / Reminders

- Please follow the Week 02 To Do List instructions
- Quiz #01 will be posted on Friday
  - a separate announcement will be sent to you to let you know that it is available
  - you will have a week or more to complete it
- Written Assignment #01 will be posted this week
- **Midterm** Exam (consider fixed):
  - October 13th, 2022 during (Thursday) lecture time

# Plan for Today

- **Problem Solving: Searching**

# 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?

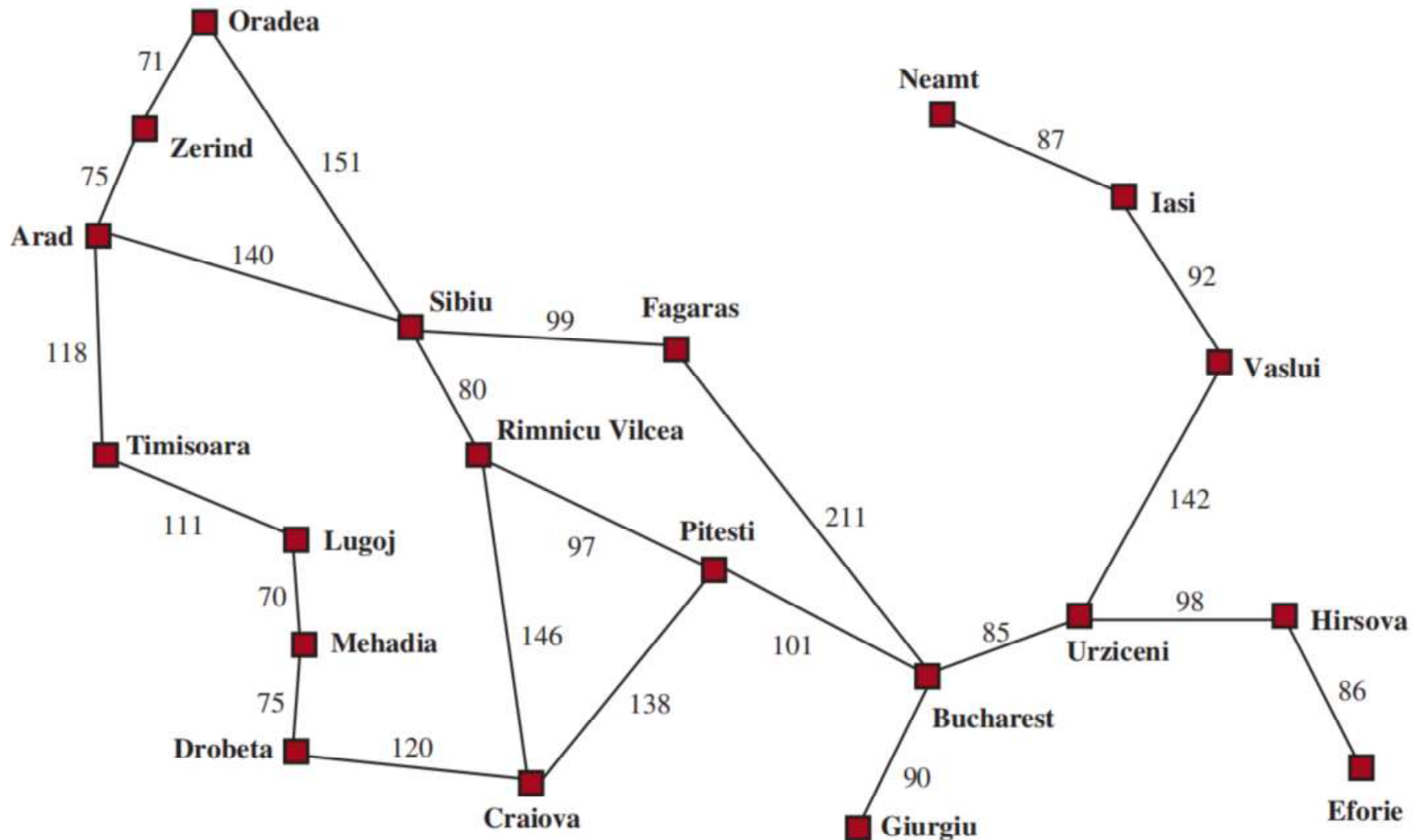
You can execute the “plan” without worrying about incoming percepts (open-loop control)

# 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: Dracula's Roadtrip

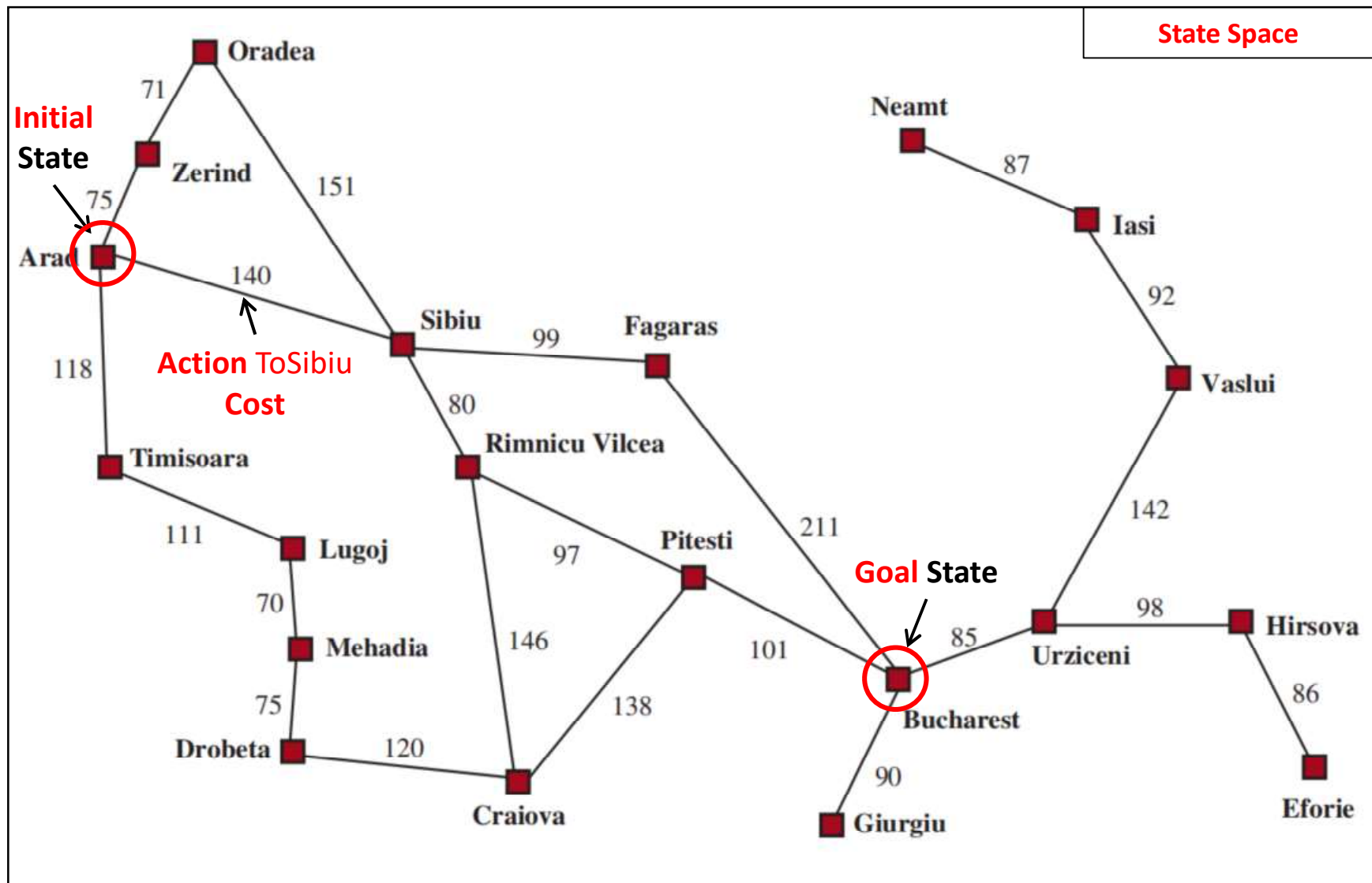


**Problem:** Get from Arad to Bucharest efficiently (for example: quickly or cheaply).

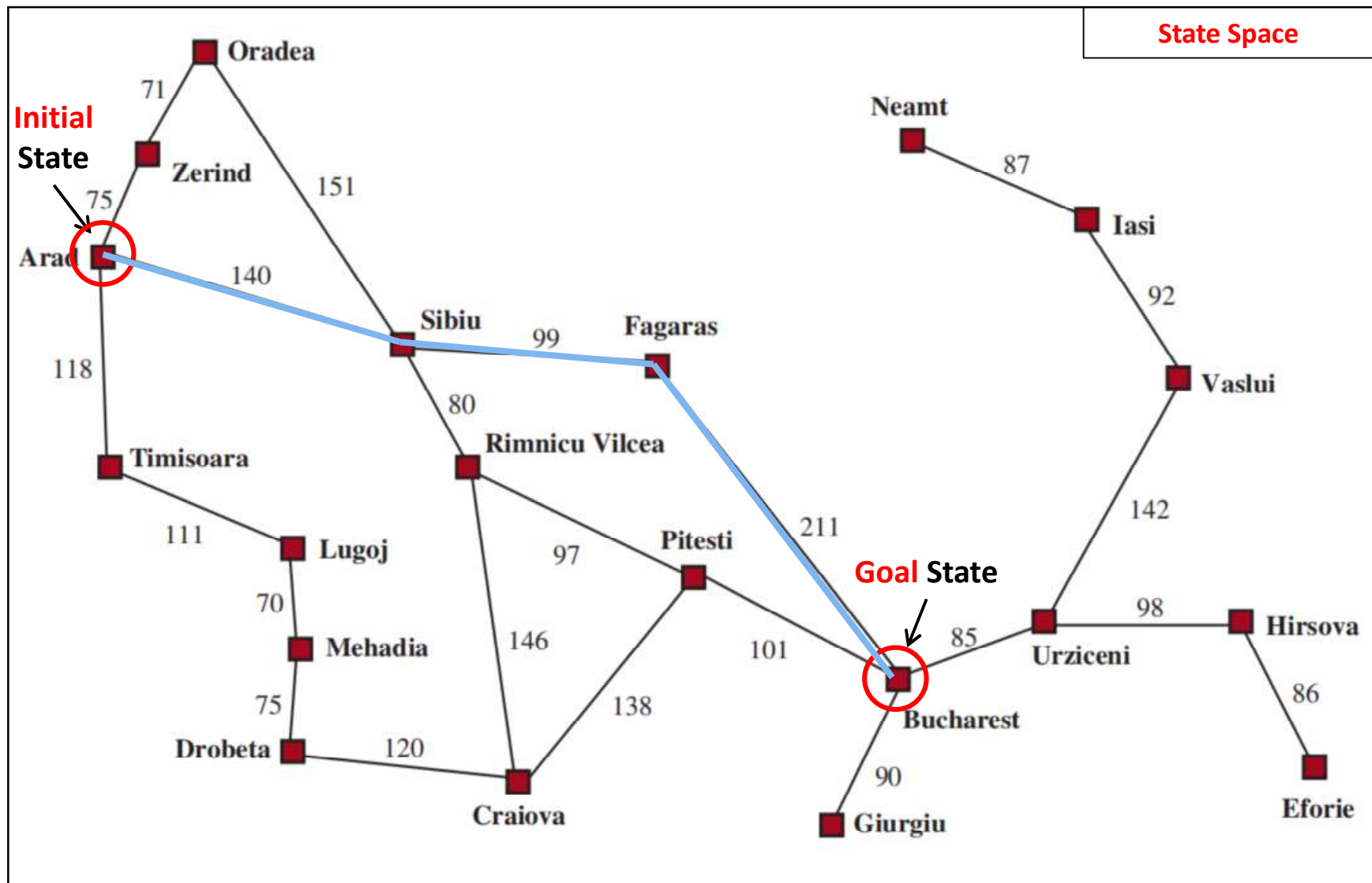
# Search Problem: Dracula's Roadtrip

- 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:  **$ActionCost(Arad, ToSibiu, Sibiu) = 140$**

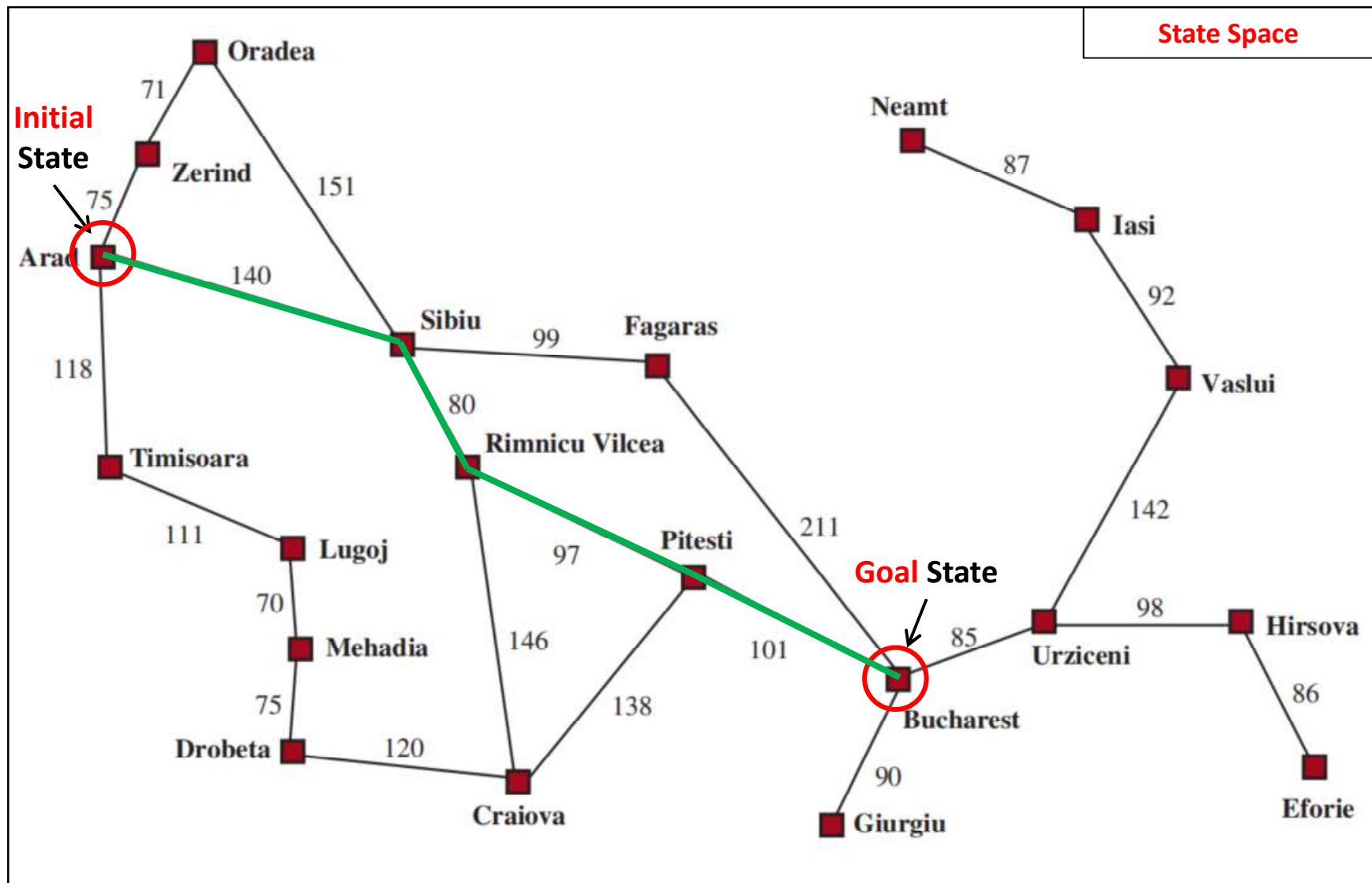
# Sample Problem: Dracula's Roadtrip



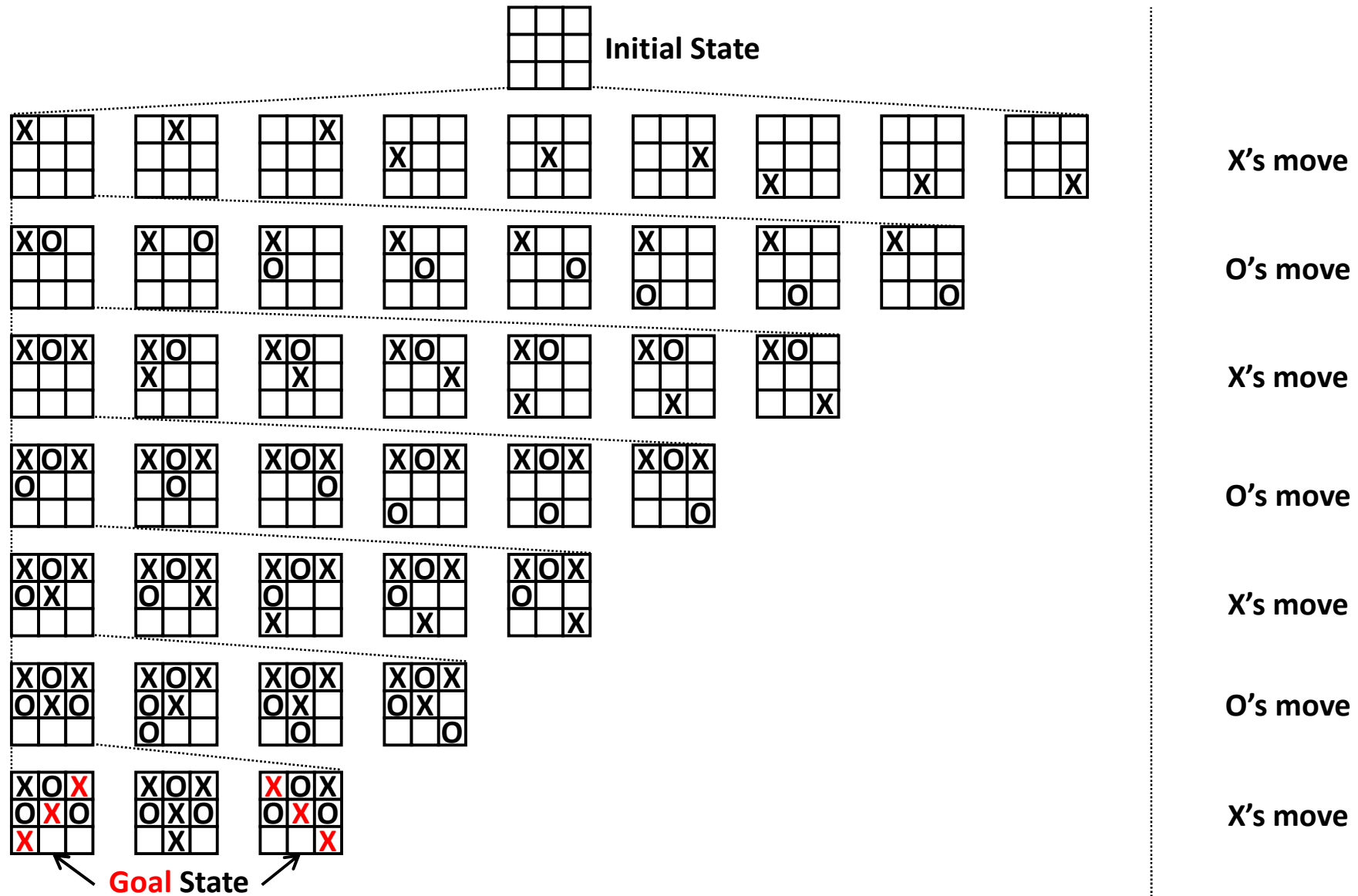
# Dracula's Roadtrip: Potential Solution



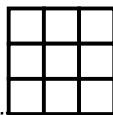
# Dracula's Roadtrip: Potential Solution



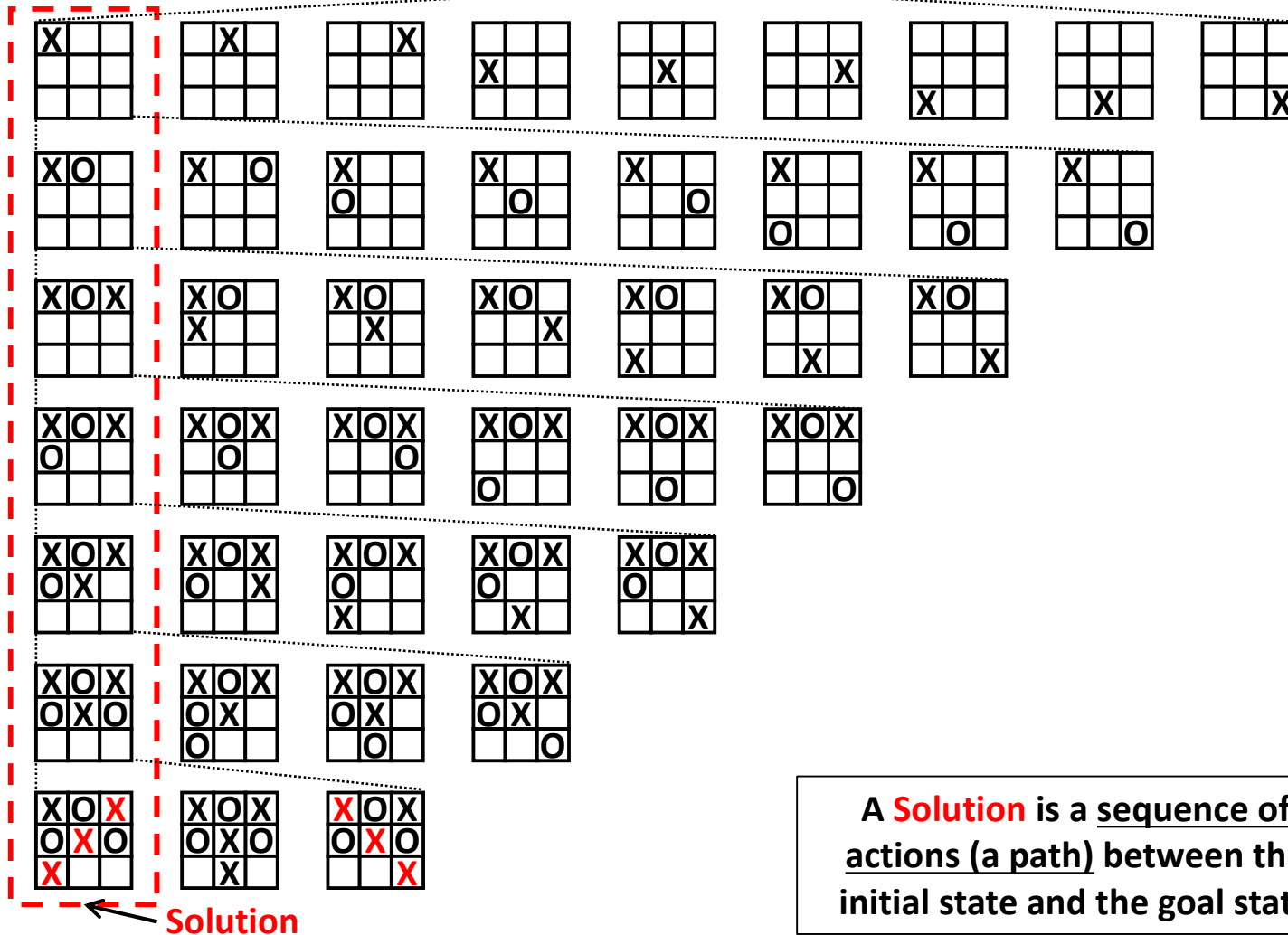
# Tic Tac Toe: (Partial) State Space



# Tic Tac Toe: Solution



Initial State



X's move

O's move

X's move

O's move

X's move

O's move

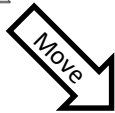
X's move

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

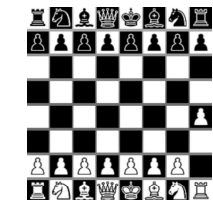
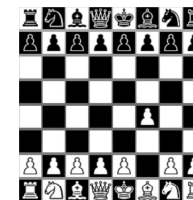
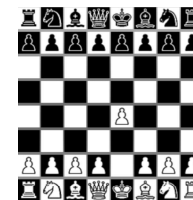
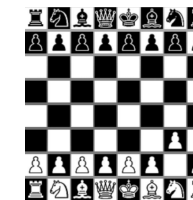
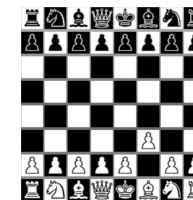
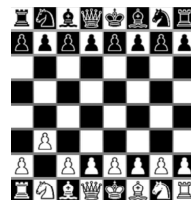


# Chess: (First Move) State Space

Initial  
State

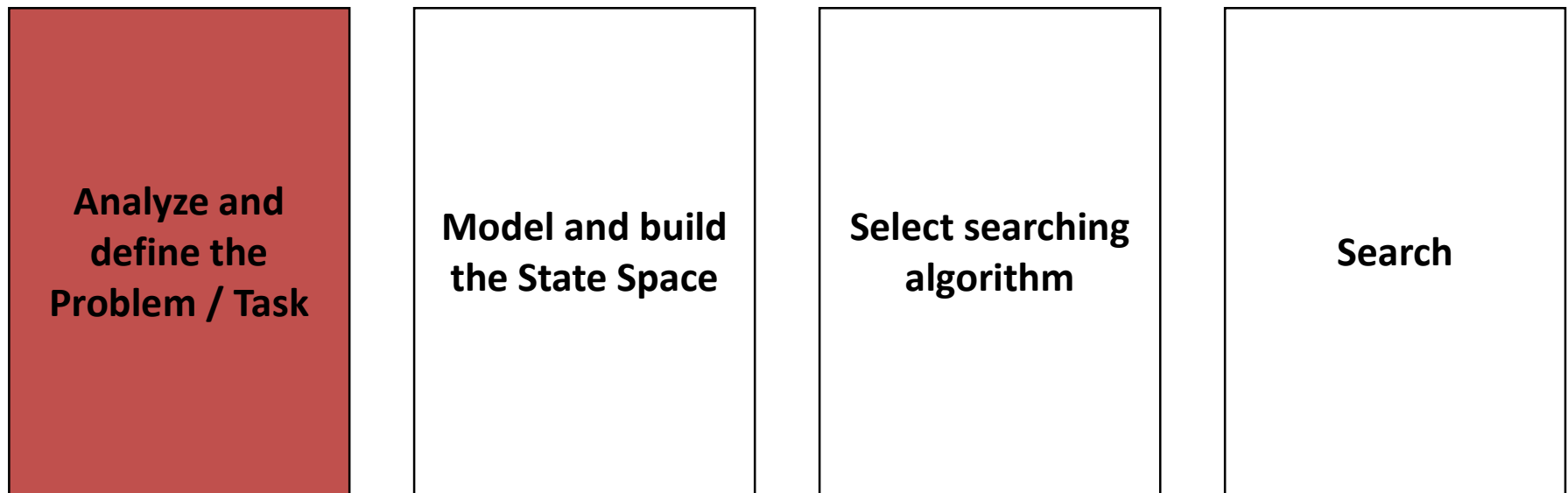


20 Possible **legal** first moves:  
16 pawn moves  
4 knight moves





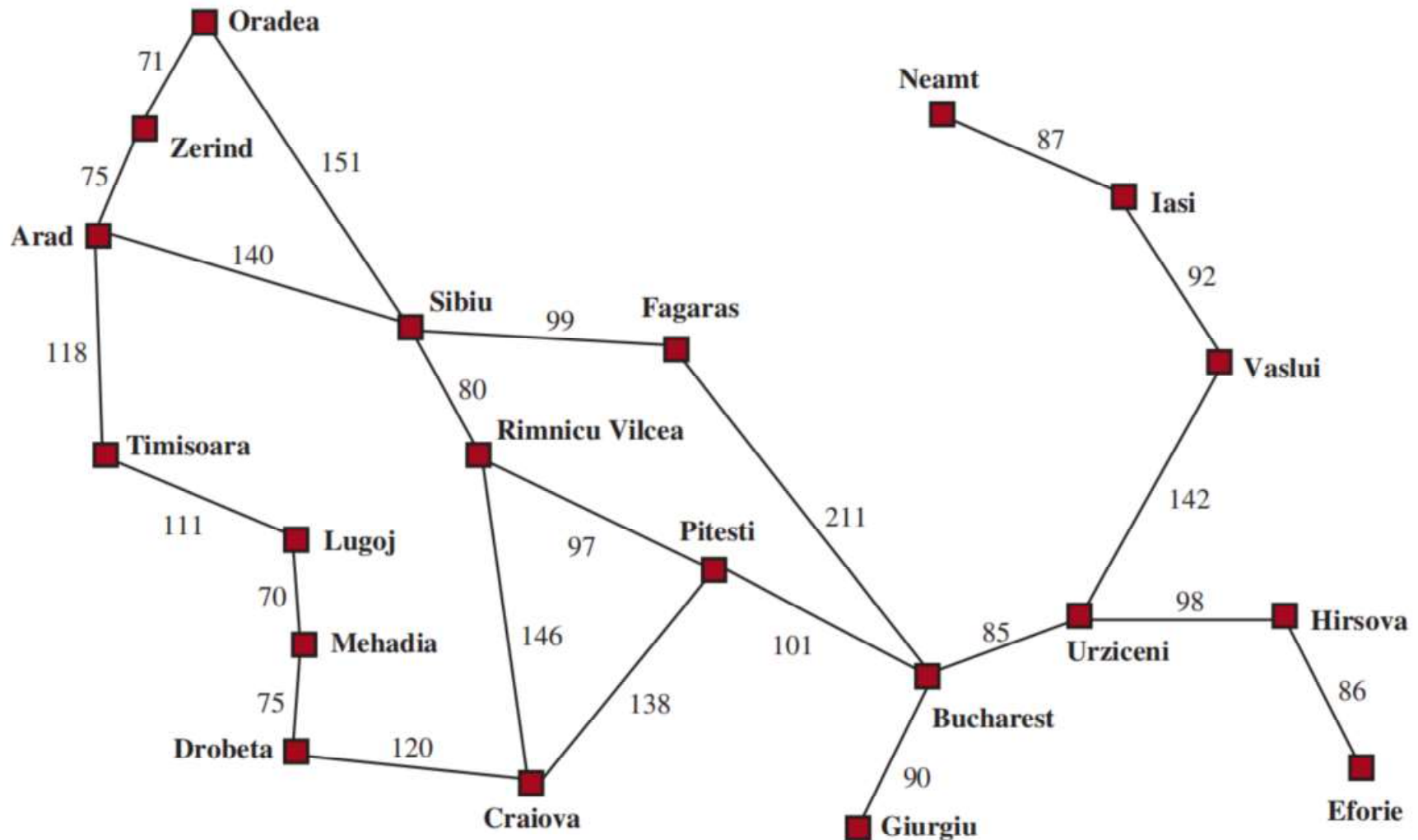
# Designing the Searching Problem



# 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: Dracula's Roadtrip

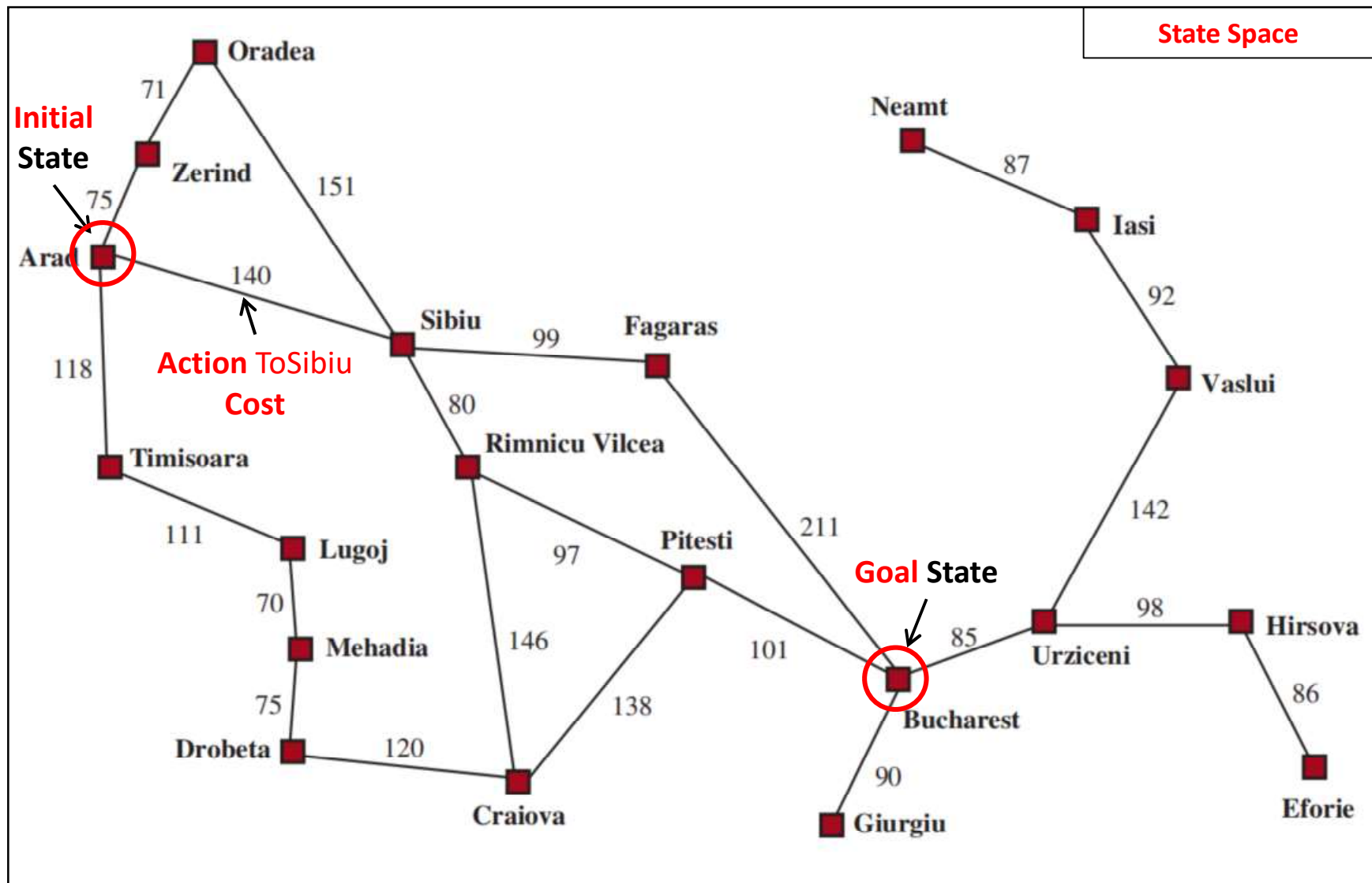


**Problem:** Get from Arad to Bucharest efficiently (for example: quickly or cheaply).

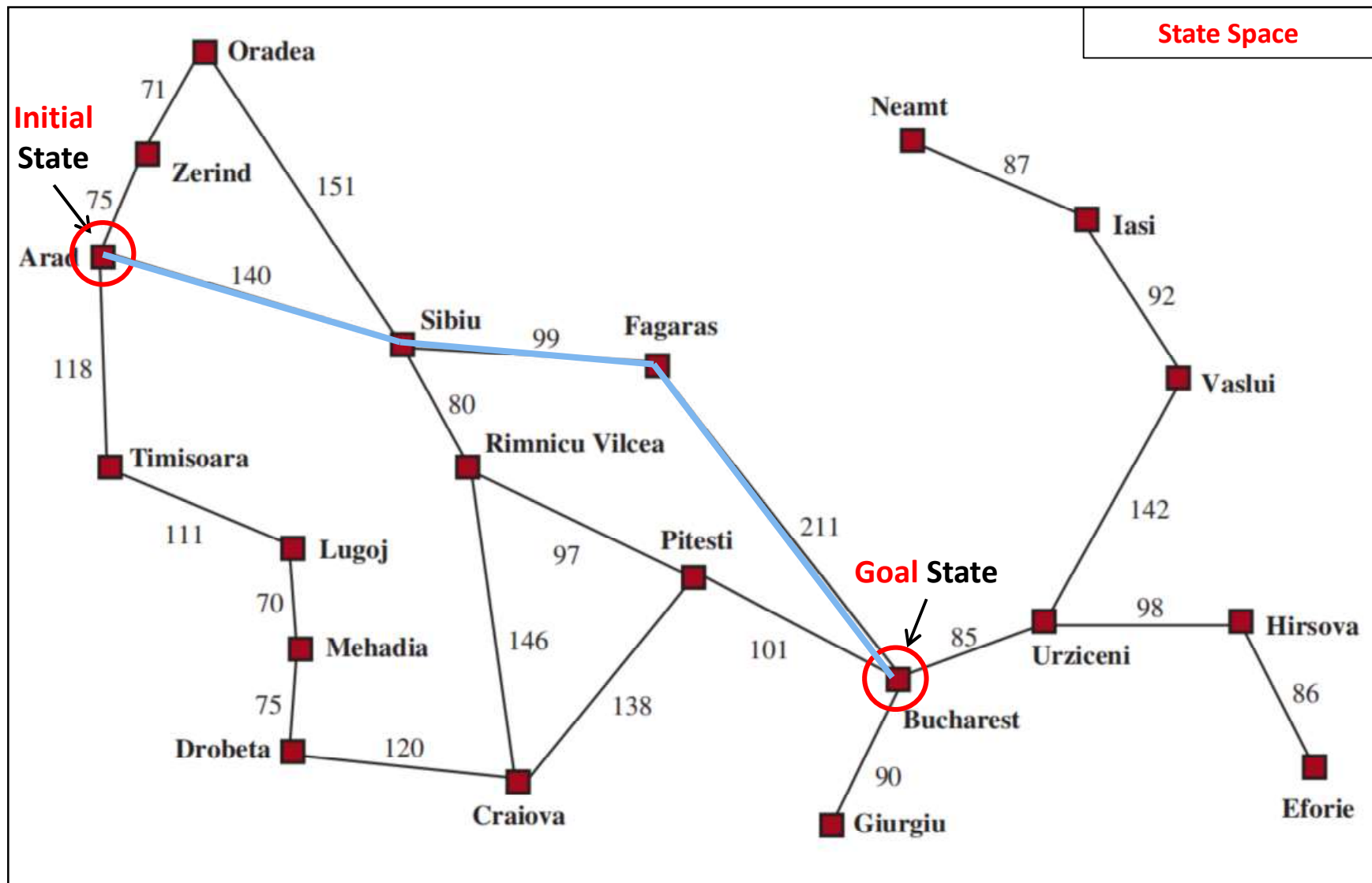
# Search Problem: Dracula's Roadtrip

- 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:  **$ActionCost(Arad, ToSibiu, Sibiu) = 140$**

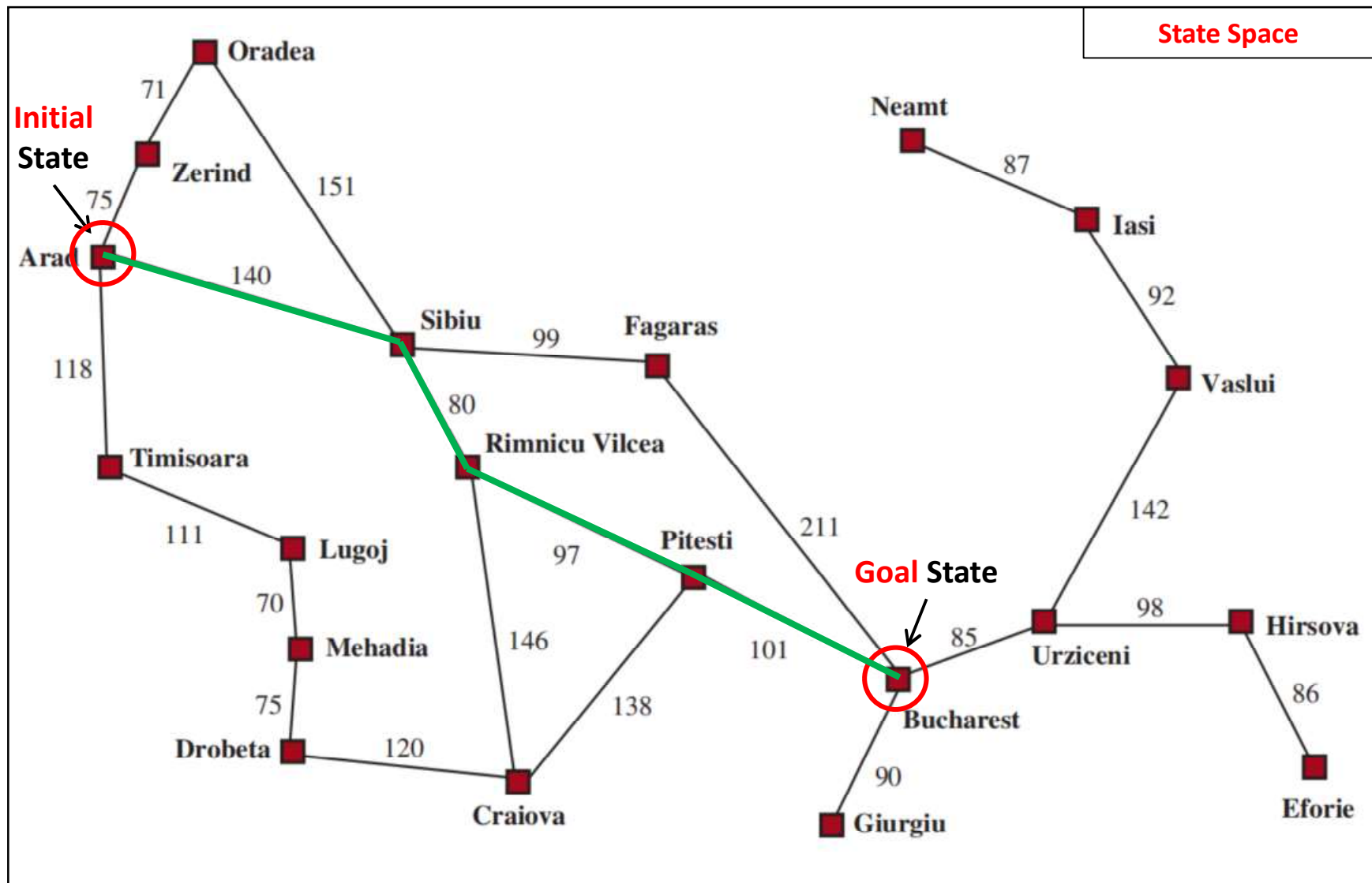
# Sample Problem: Dracula's Roadtrip



# Dracula's Roadtrip: Potential Solution

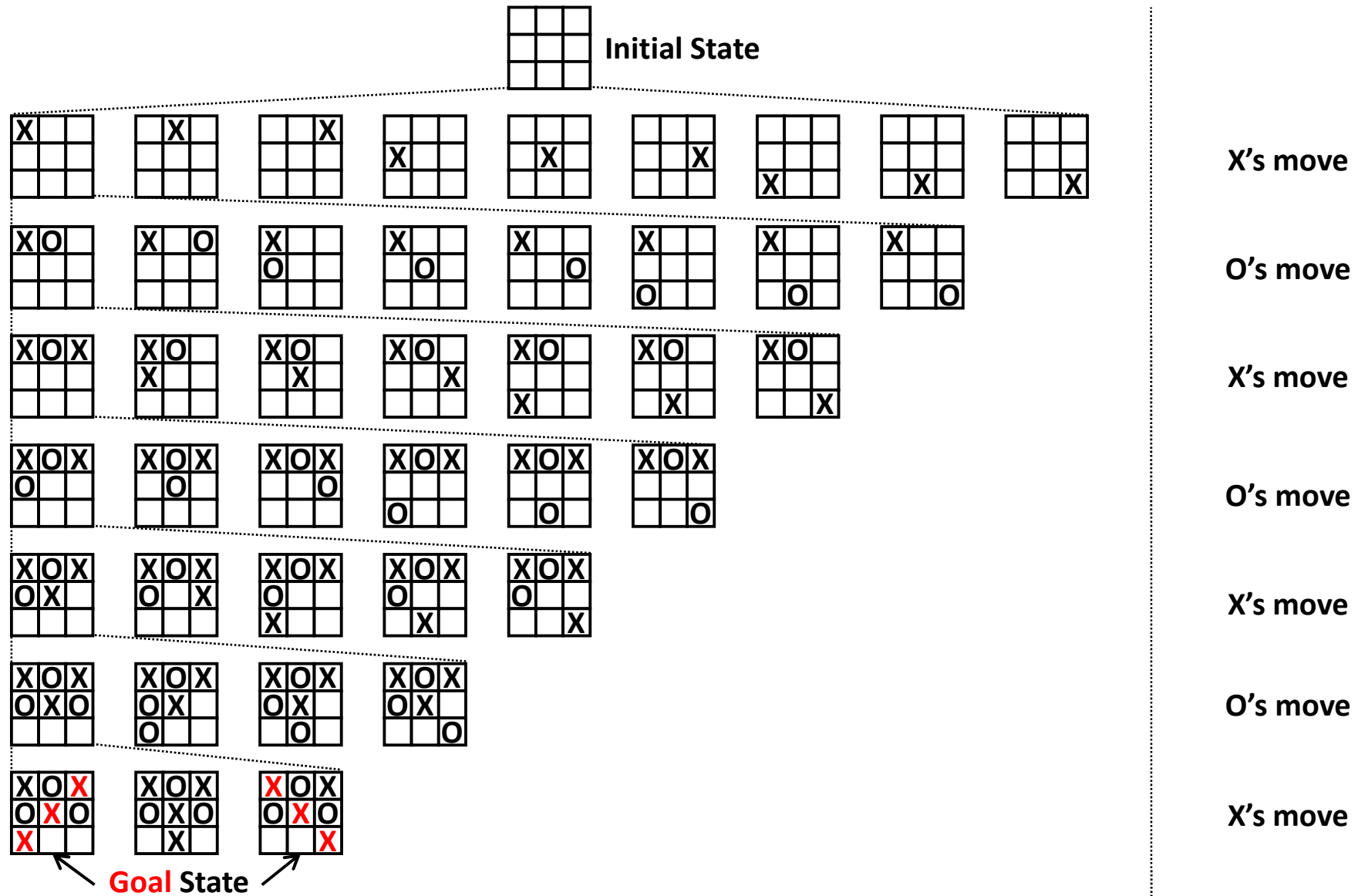


# Dracula's Roadtrip: Potential Solution



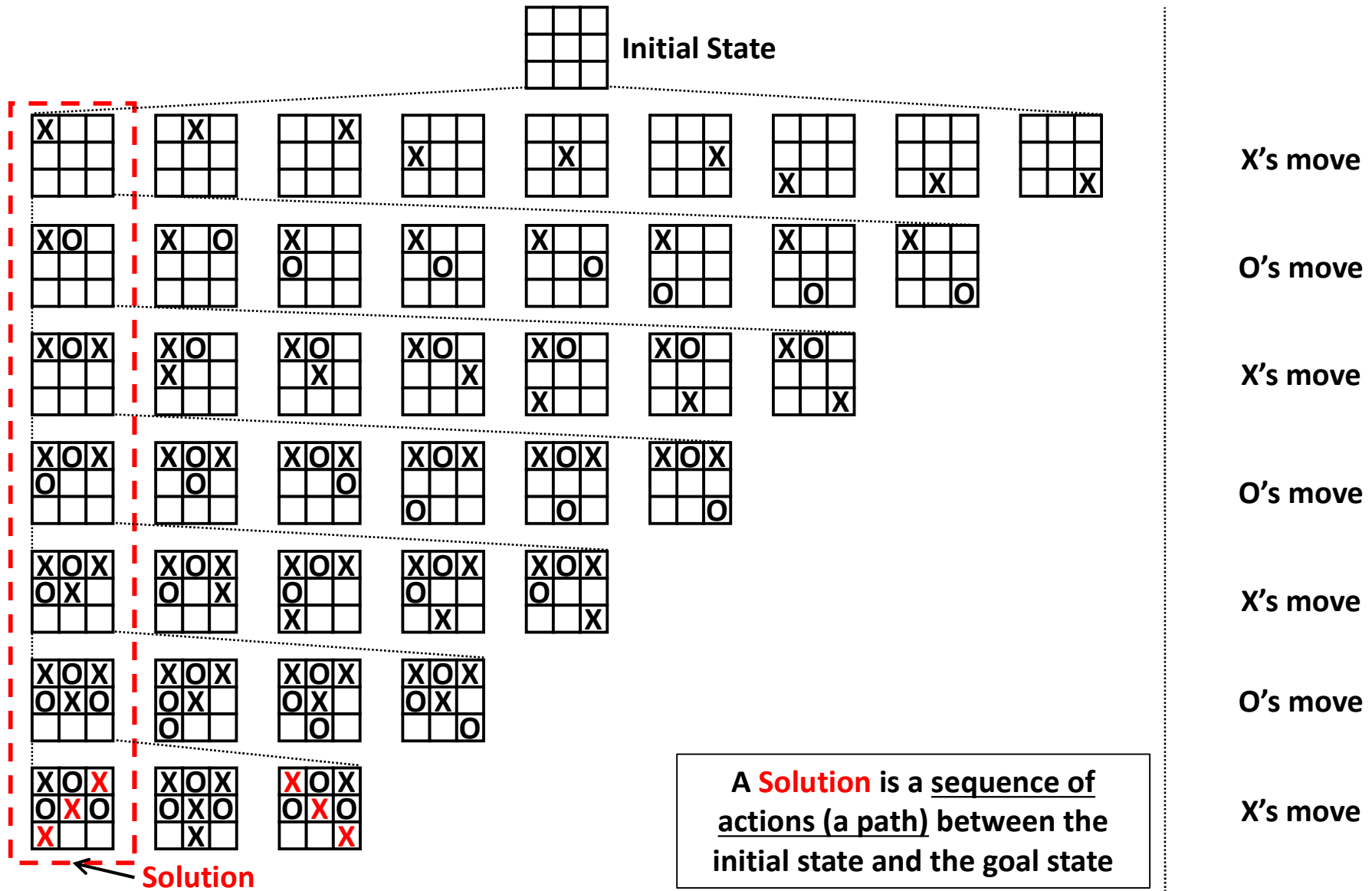


# Tic Tac Toe: (Partial) State Space





# Tic Tac Toe: Solution

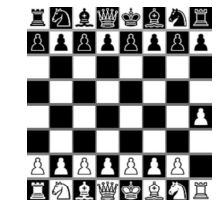
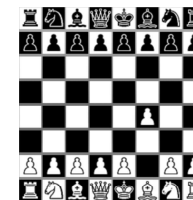
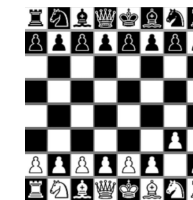
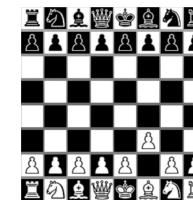
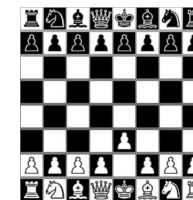
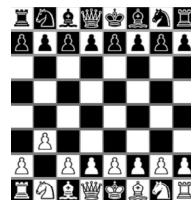


# Chess: (First Move) State Space

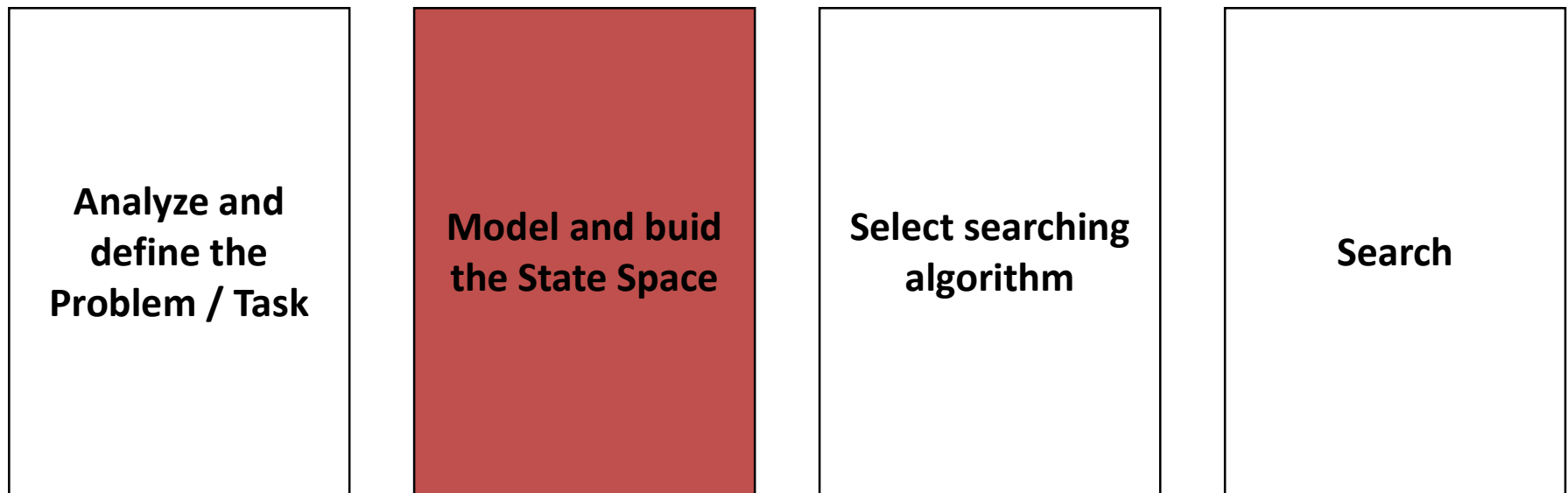
Initial  
State



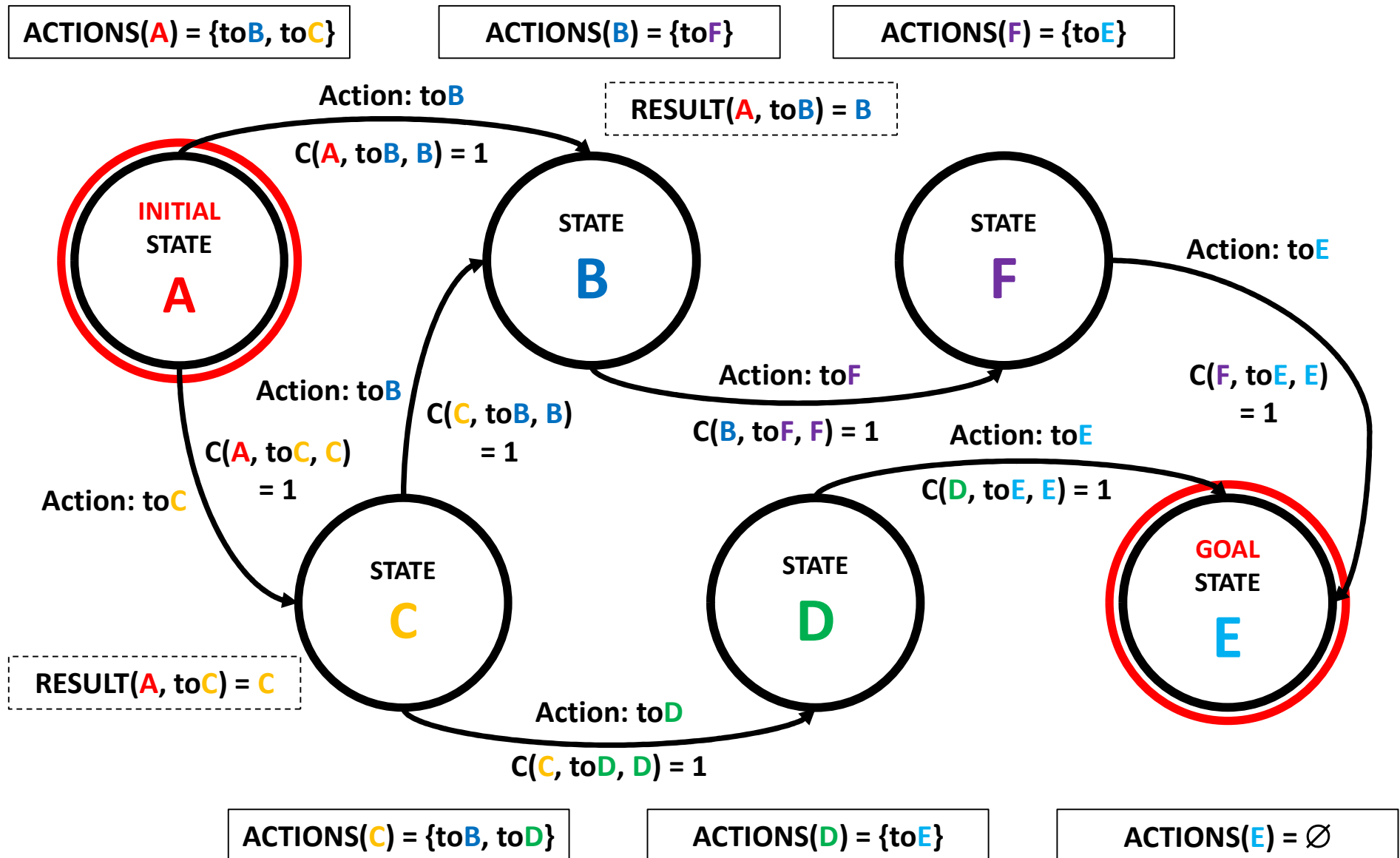
20 Possible **legal** first moves:  
16 pawn moves  
4 knight moves



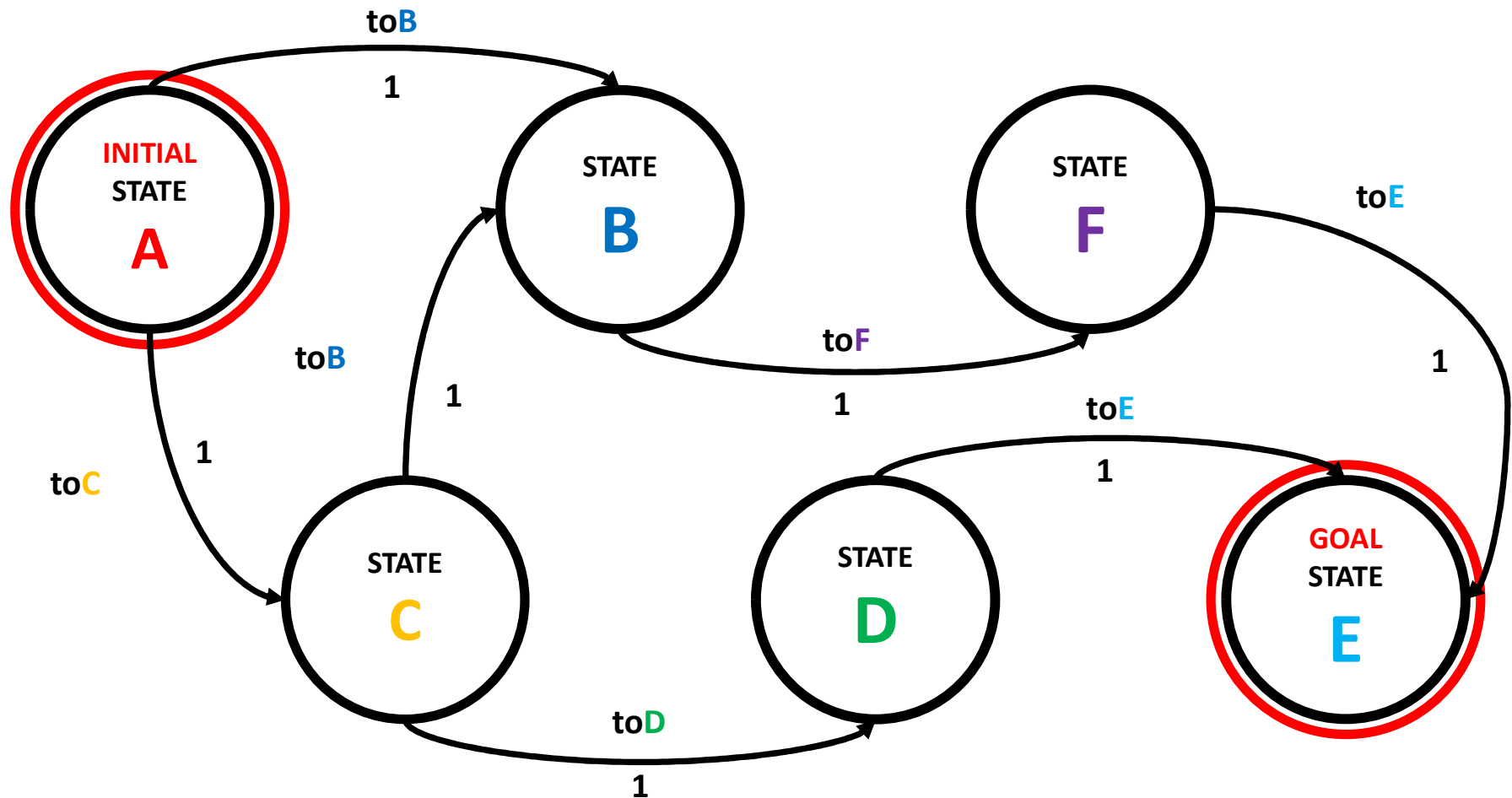
# Designing the Searching Problem



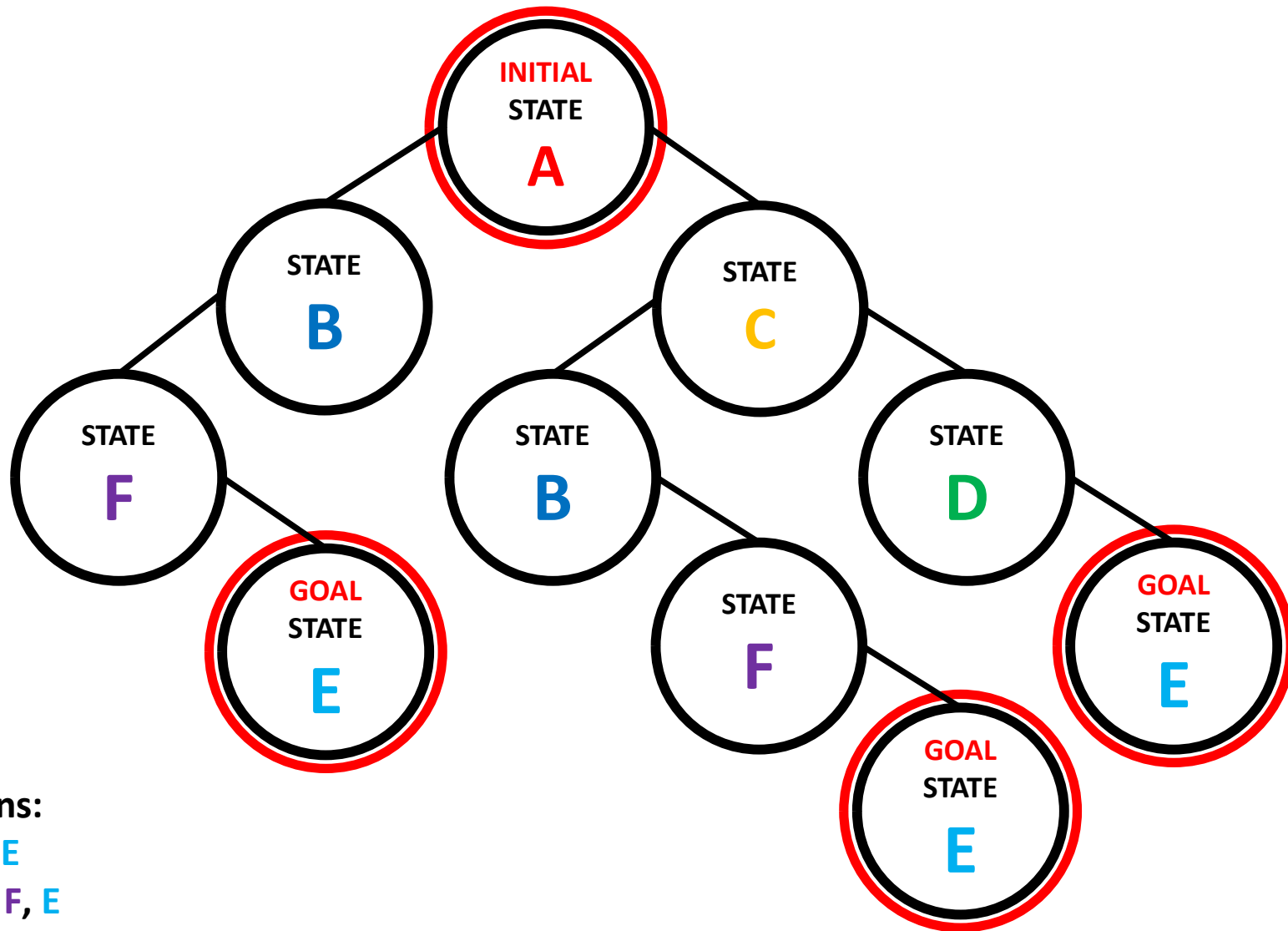
# State Space Model: A Graph



# State Space Model: A Graph



# Searching State Space: Search Tree



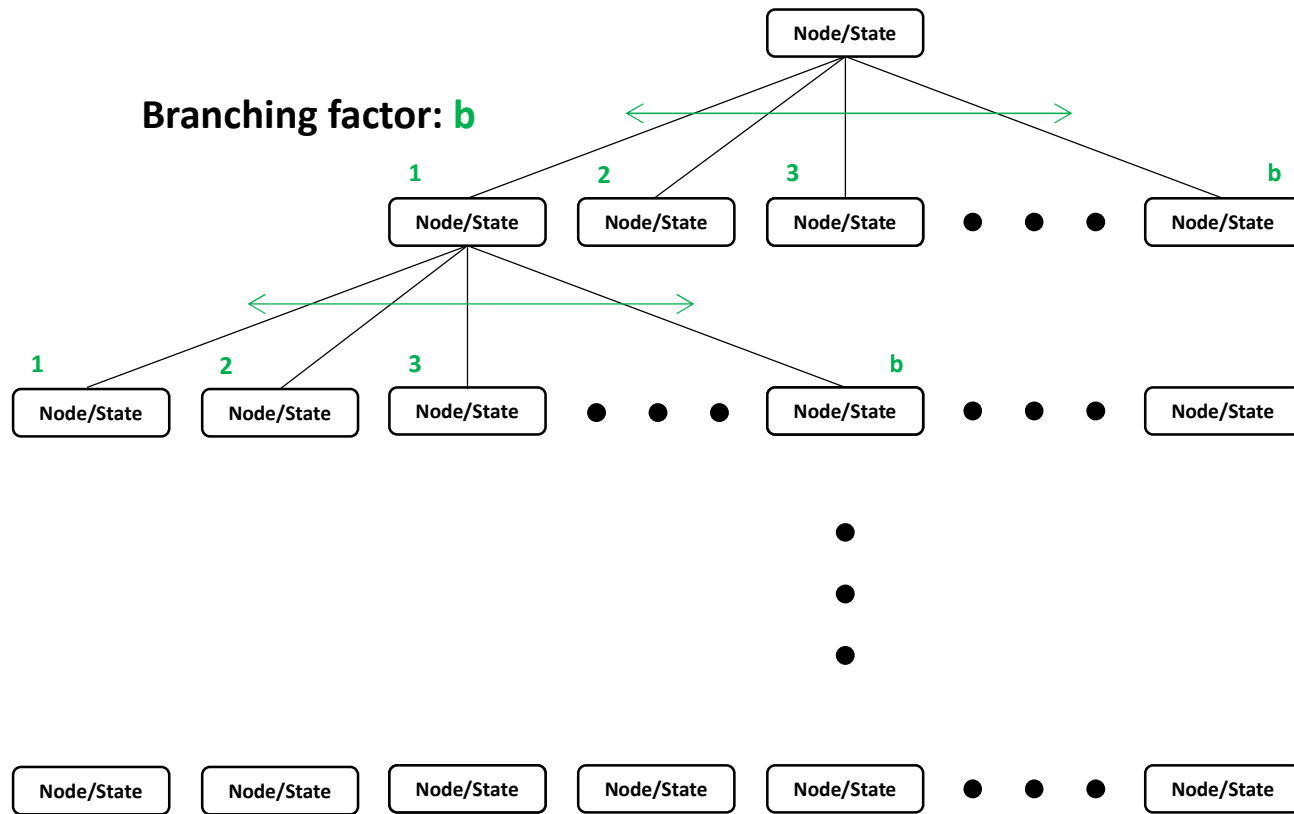
Solutions:

A, B, F, E

A, C, B, F, E

A, C, D, E

# Search Tree Challenges: Size



Depth: 0 |  $N_0 = 1$

Depth: 1 |  $N_1 = b$

Depth: 2 |  $N_2 = b^2$

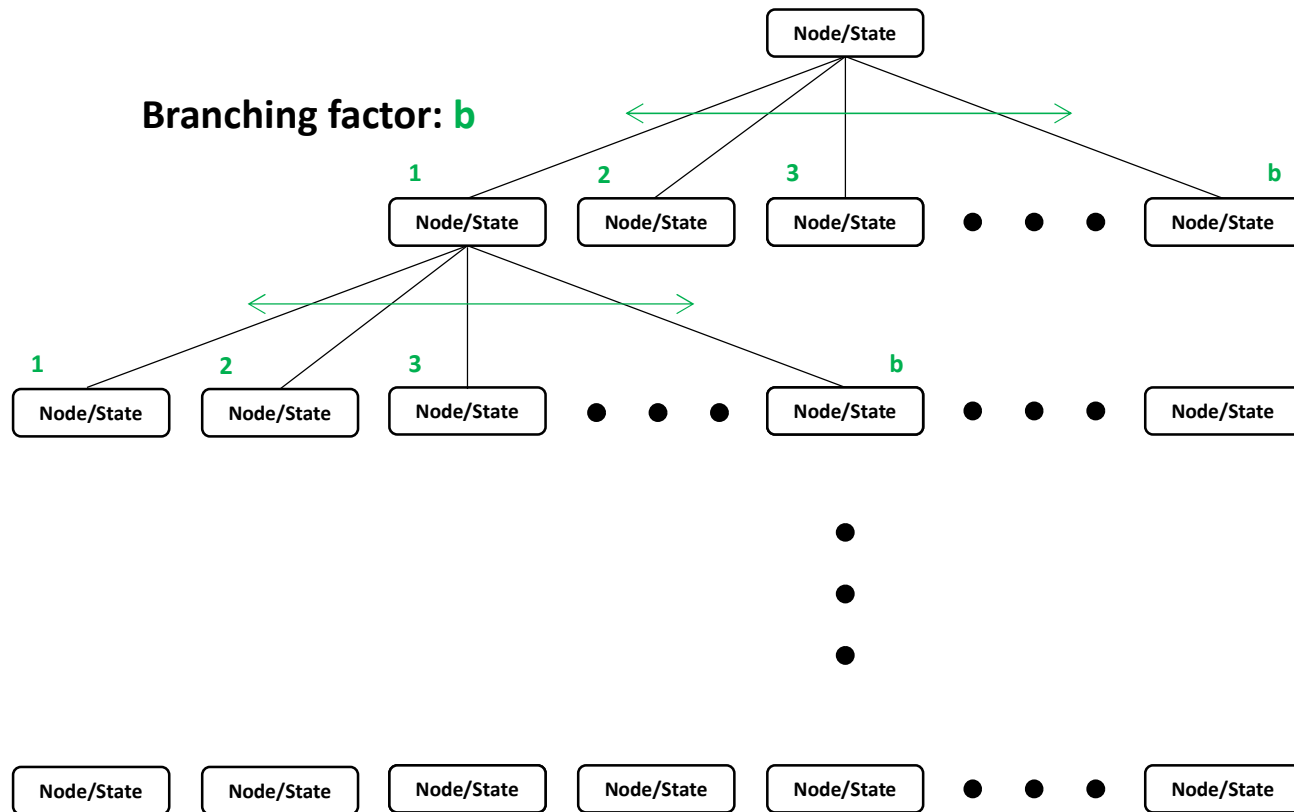
•  
•  
•

Depth:  $d$  |  $N_d = b^d$

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

Quickly becomes unmanageable and impossible to search with brute force!

# Search Tree Challenges: Infiniteness



## Depth: 0

## Depth: 1

### Depth: 2

•

•

•

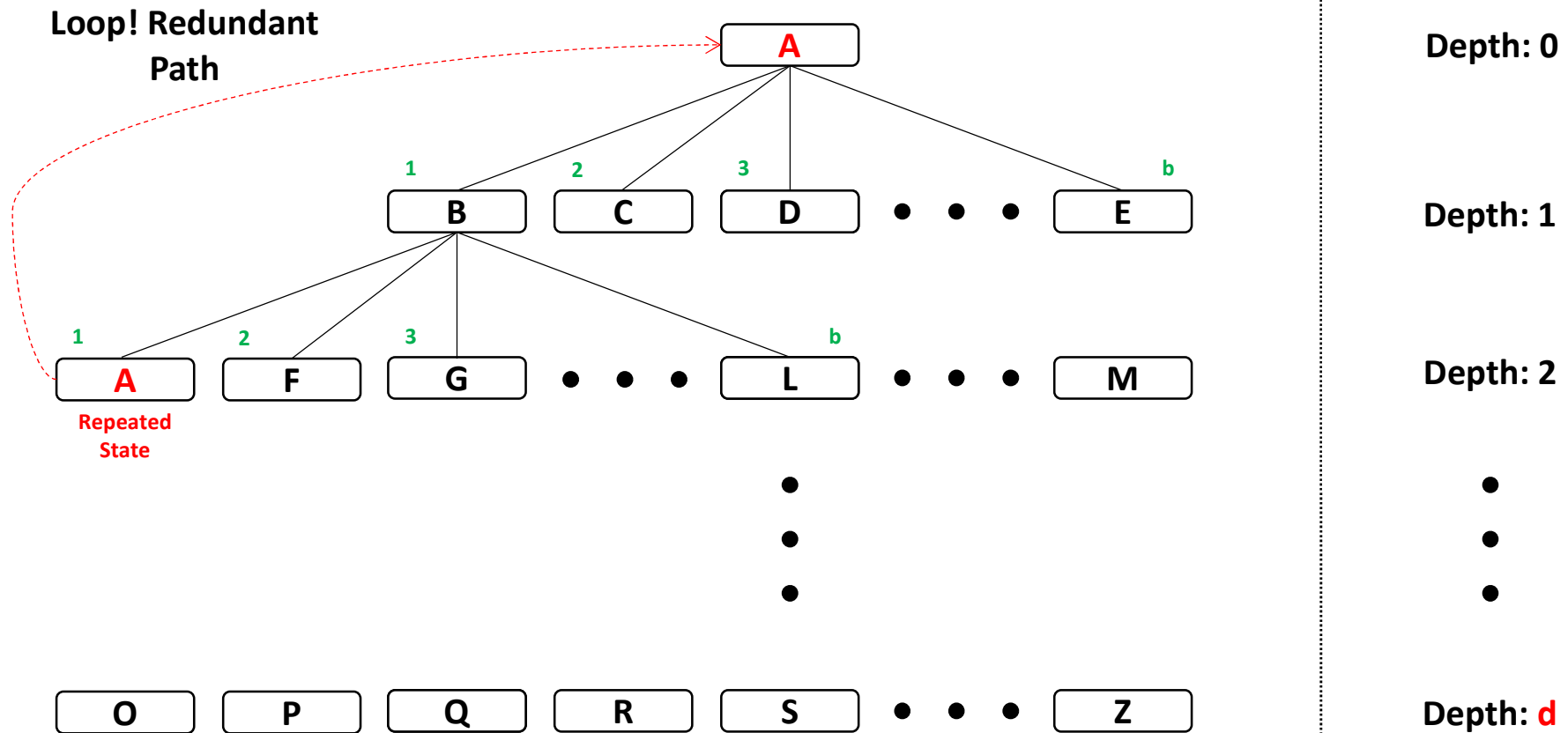
**Depth:**  $d \rightarrow \infty$

## Unmanageable and impossible to search with brute force!

## Memory and time use grows quickly!



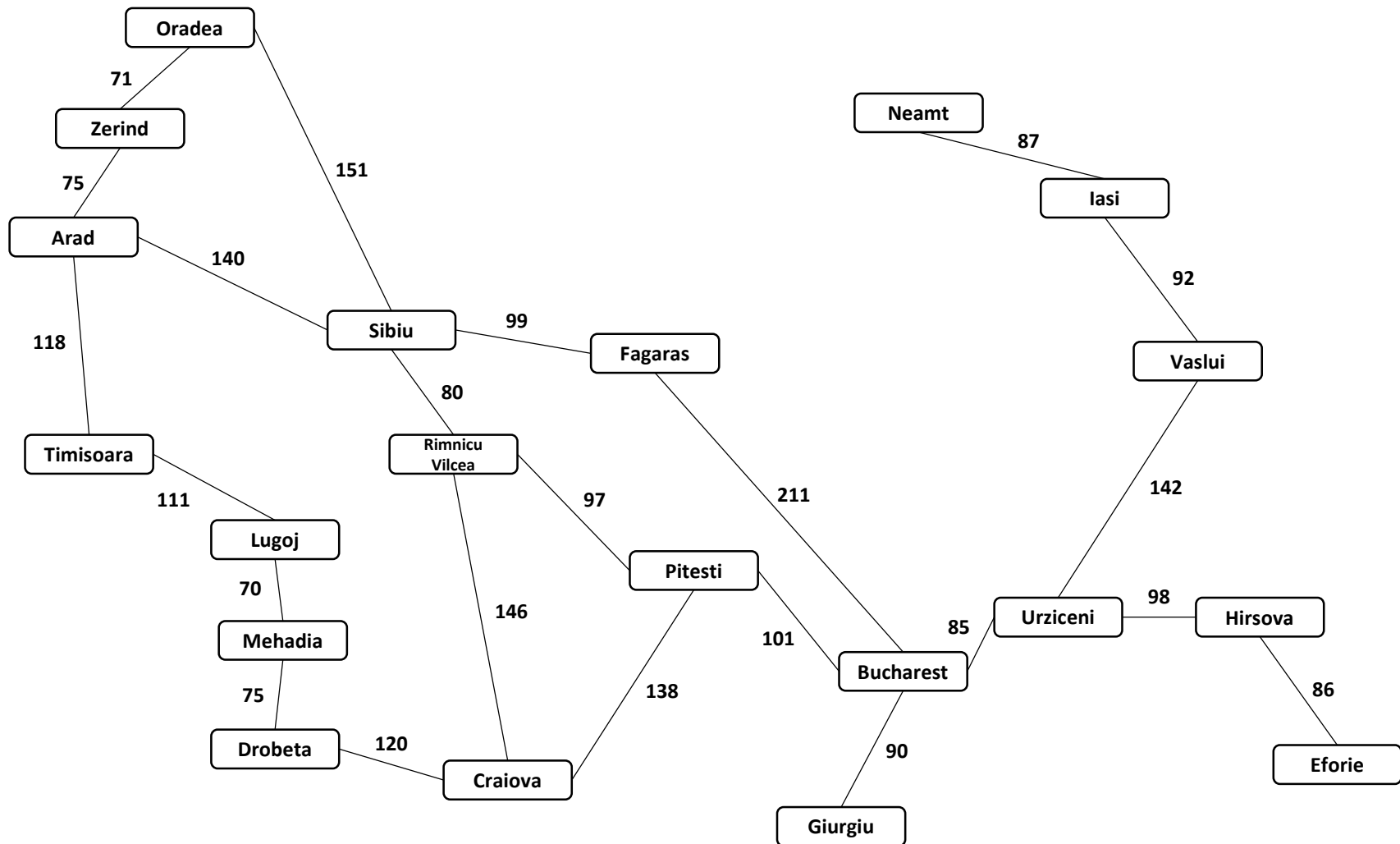
# Search Tree Challenges: Loops



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

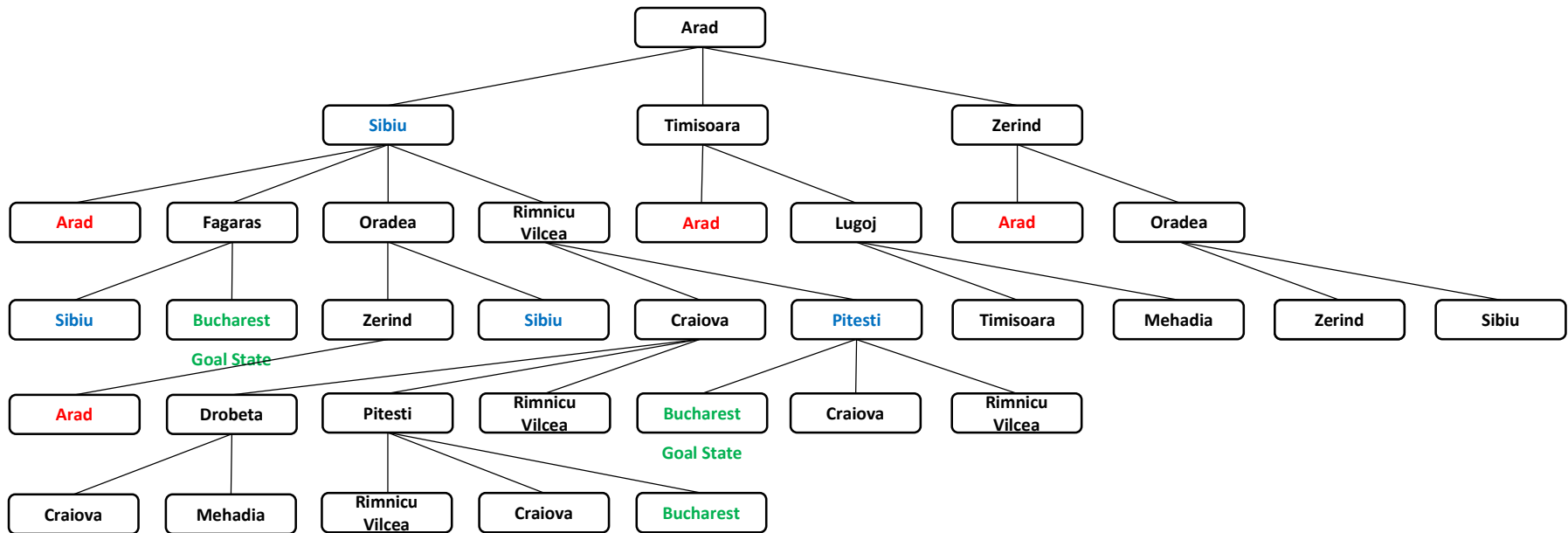
Memory and time use grows quickly!

# Sample Problem: Dracula's Roadtrip



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

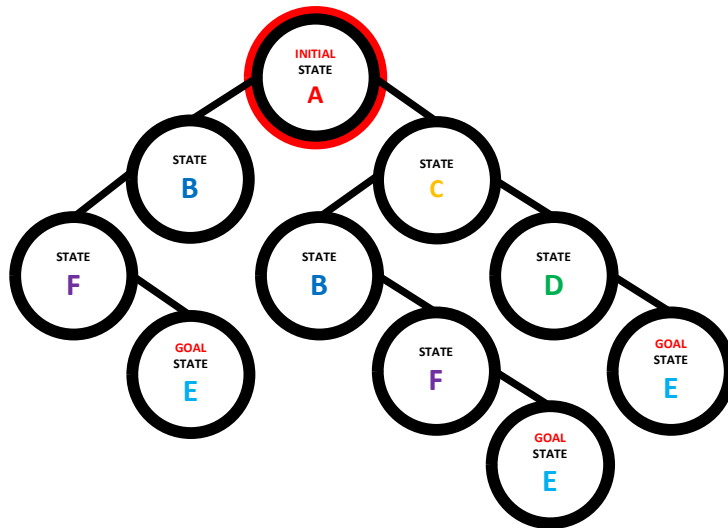
# Dracula's Roadtrip as a Tree



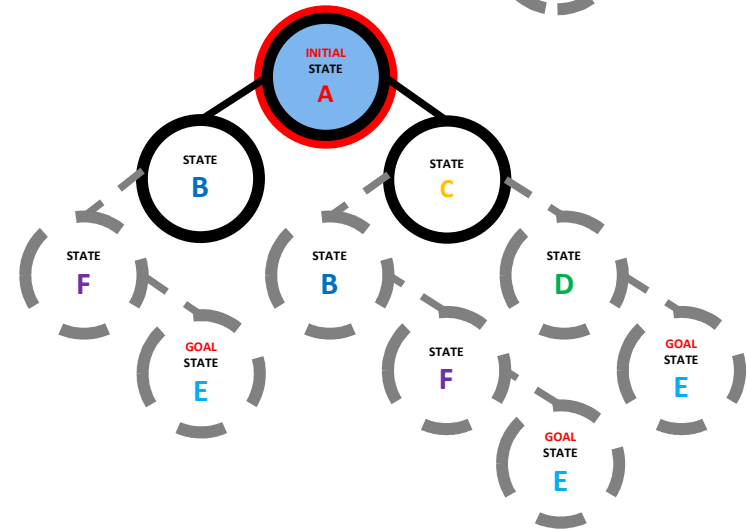
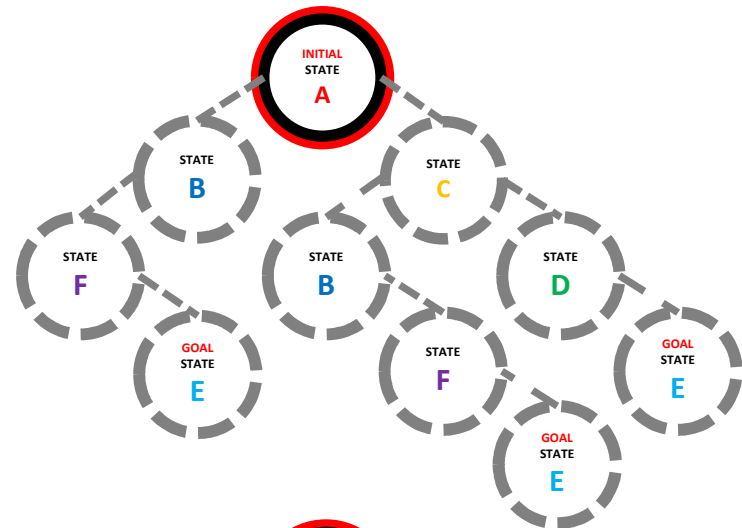
INCOMPLETE! I need to redraw it in smarter way

# Search Tree: Implementations

Build entire search tree



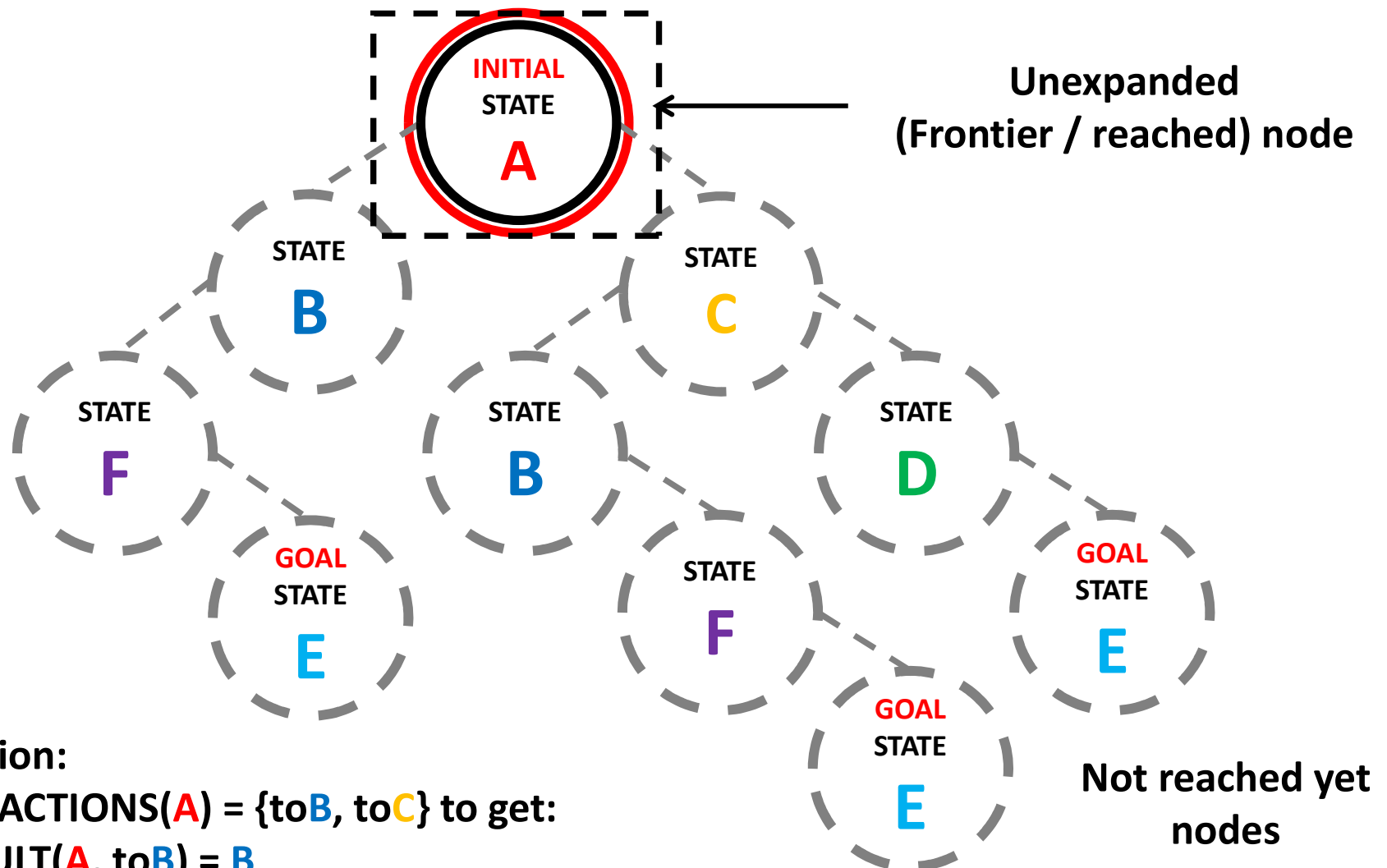
Expand/generate nodes as you go



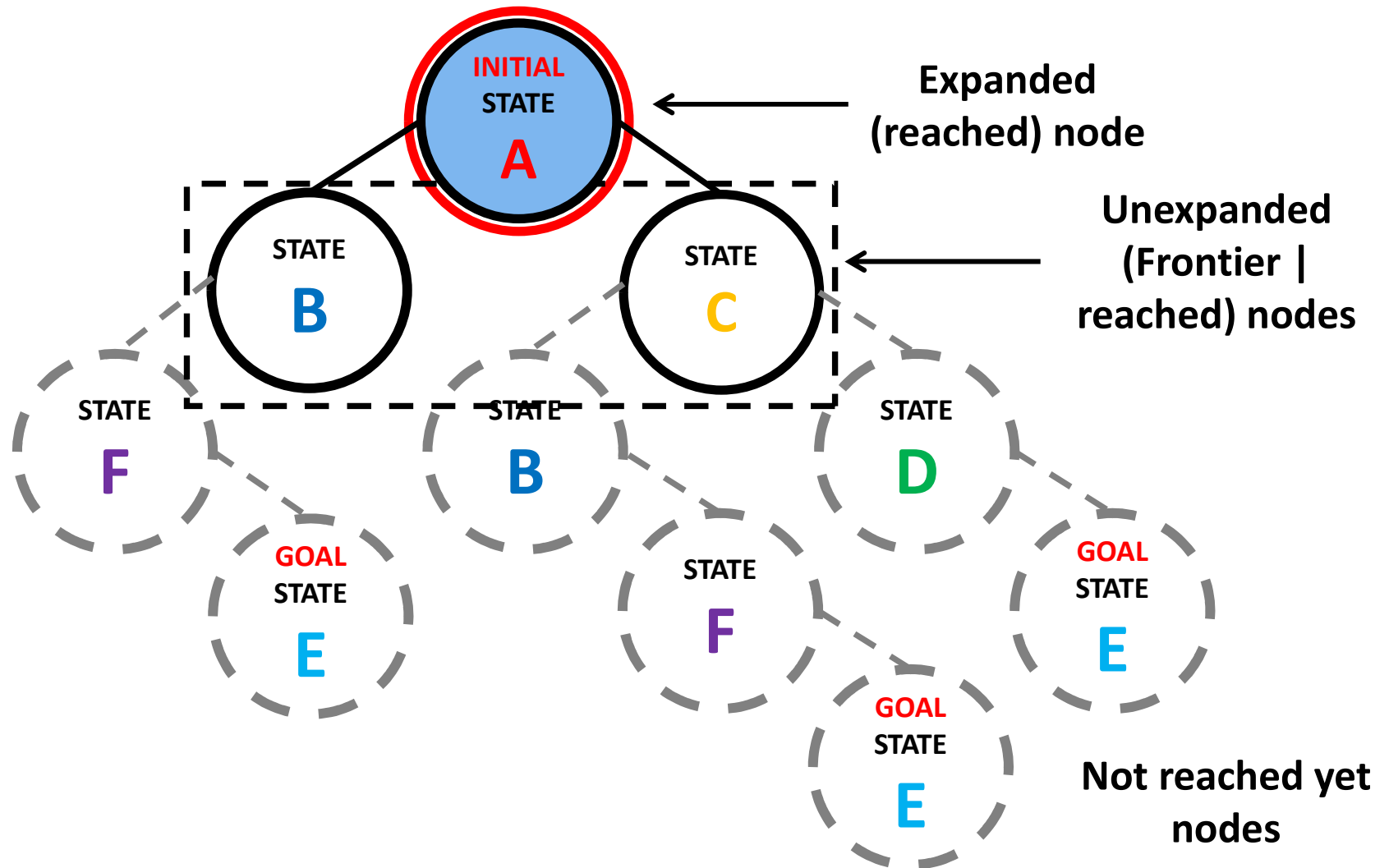
Challenges:

- memory requirements
- impossible for infinite number of states

# Search Tree: Node Expansion



# Search Tree: Node Expansion

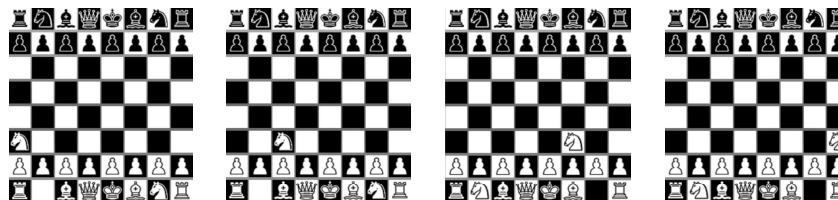


# Chess: State Node Expansion

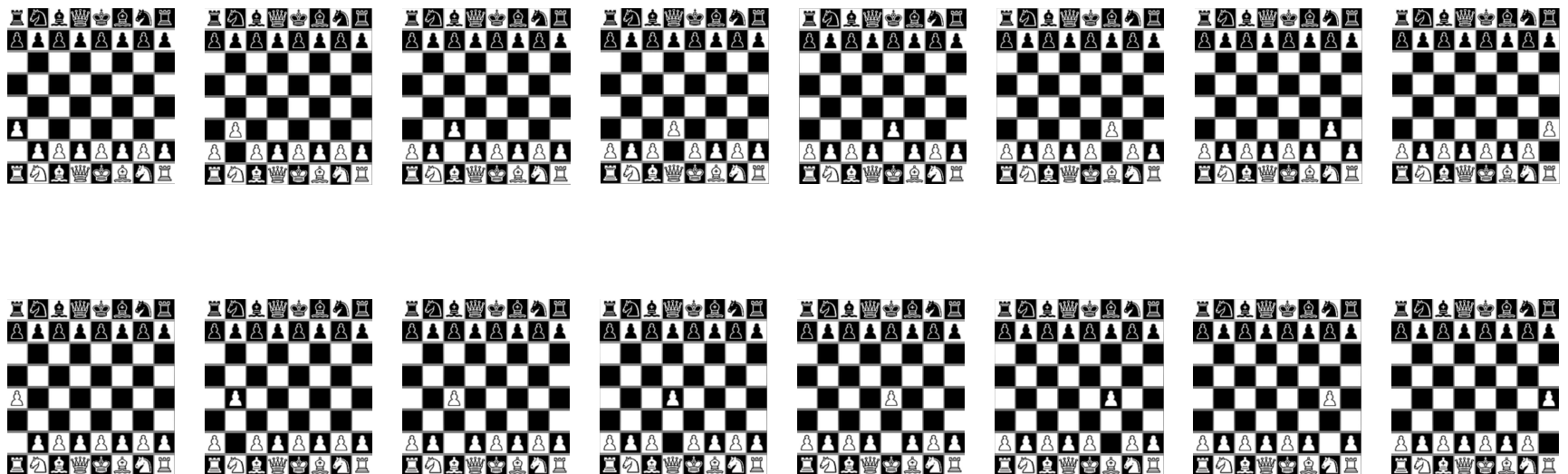
Initial  
State



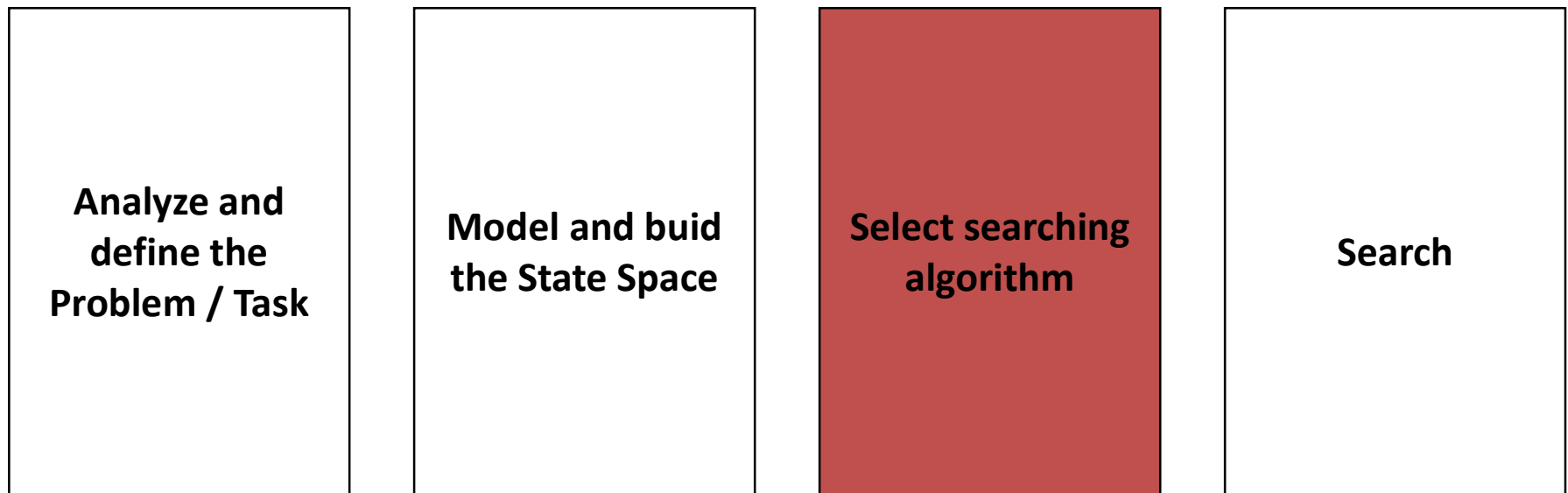
Use game rules to generate subsequent possible game  
tree states / nodes!



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



# Designing the Searching Problem



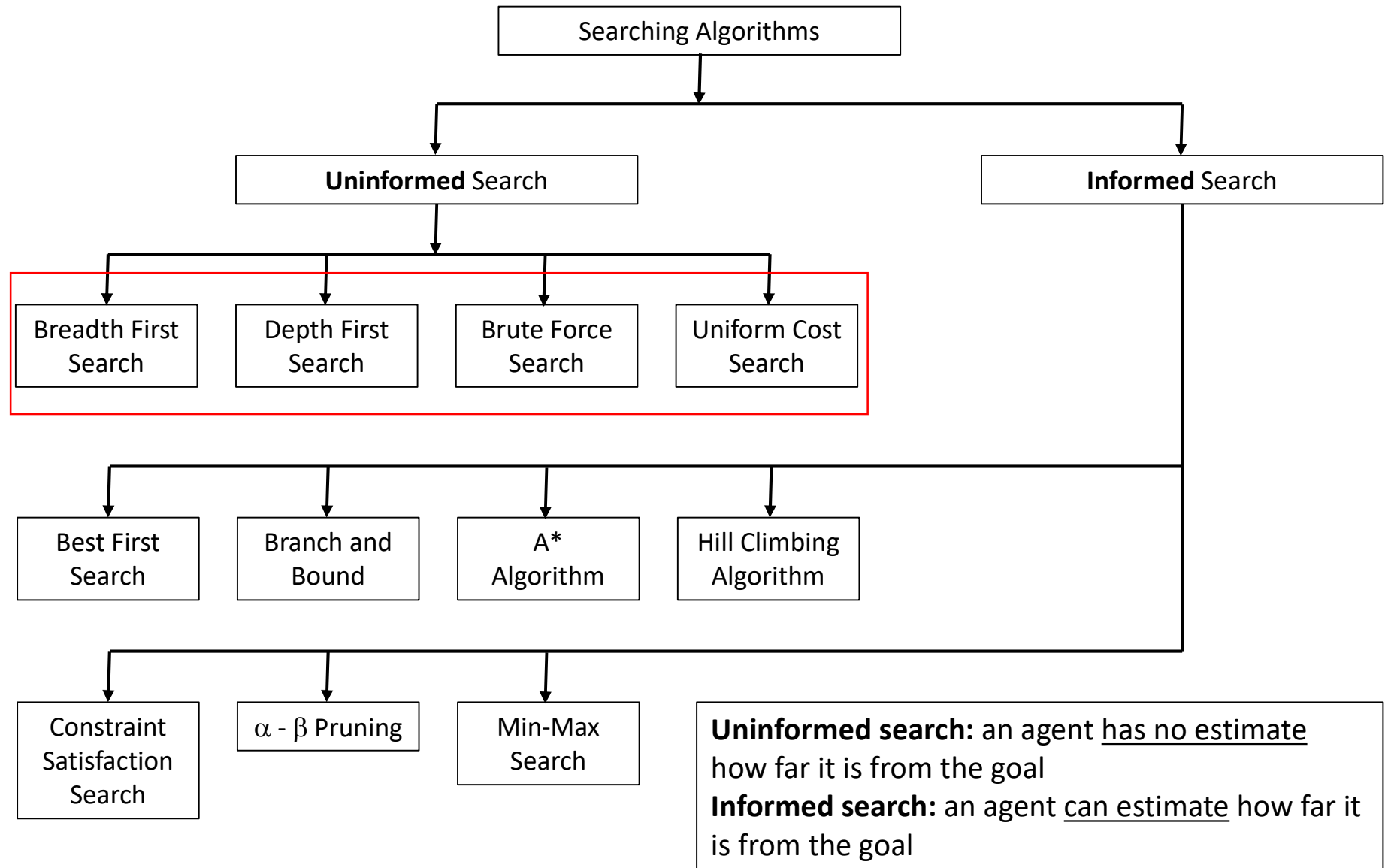


# Measuring Searching Performance

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?

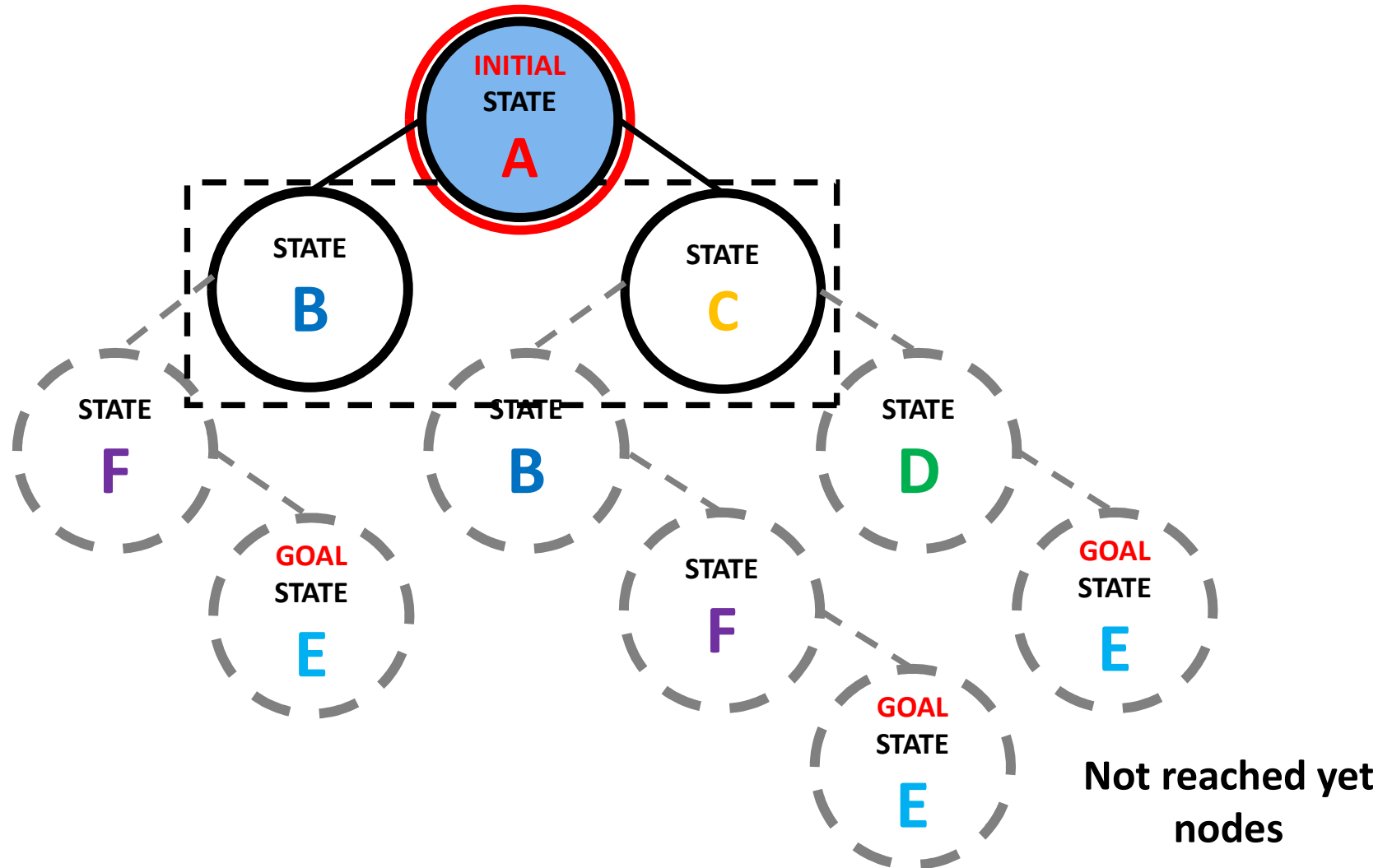
# Selected Searching Algorithms



# Uninformed 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:**

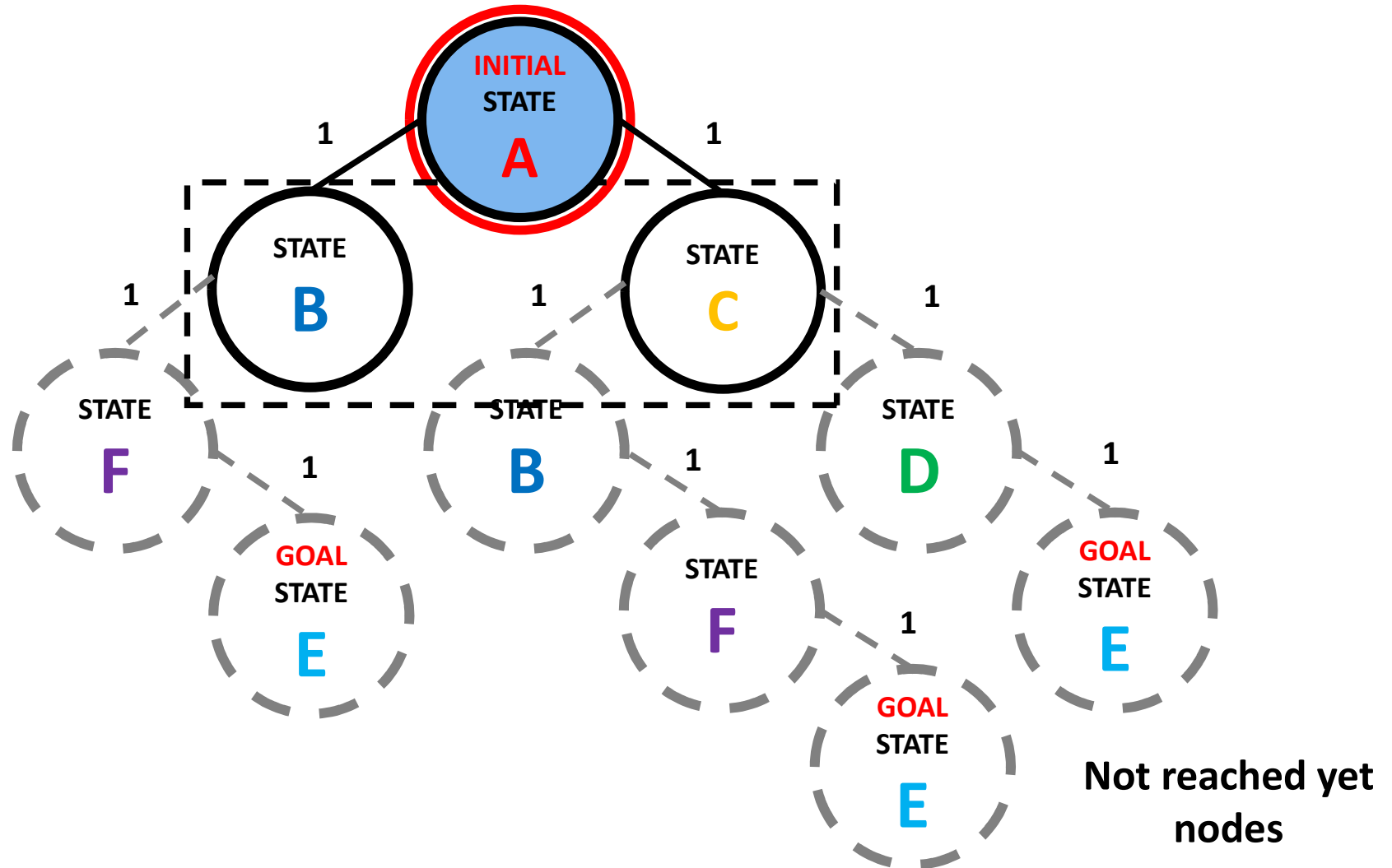
$$f(n) = f(\text{State } n)$$

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

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

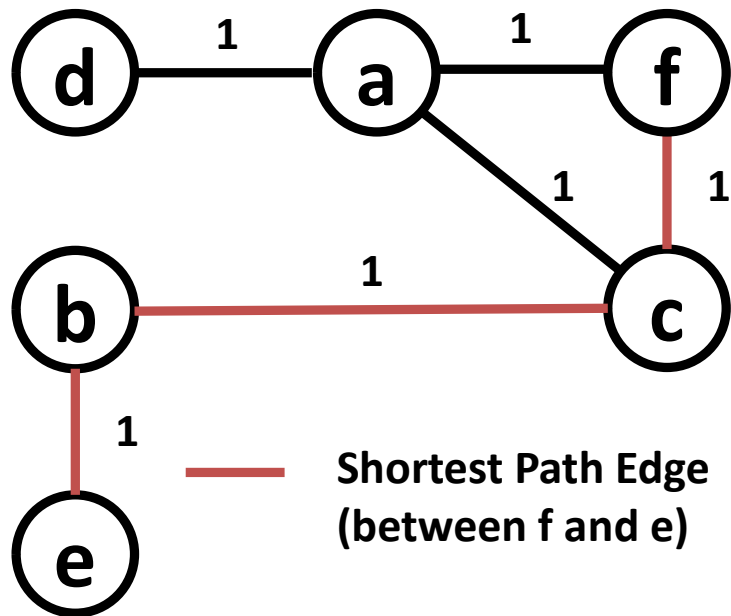
**What about ties?**

# 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*  $\leftarrow$  NODE(*problem*.INITIAL)

**if** *problem*.IS-GOAL(*node*.STATE) **then return** *node*

*frontier*  $\leftarrow$  a FIFO queue, with *node* as an element

*reached*  $\leftarrow$  {*problem*.INITIAL}

**while not** IS-EMPTY(*frontier*) **do**

*node*  $\leftarrow$  POP(*frontier*)

**for each** *child* **in** EXPAND(*problem*, *node*) **do**

*s*  $\leftarrow$  *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*

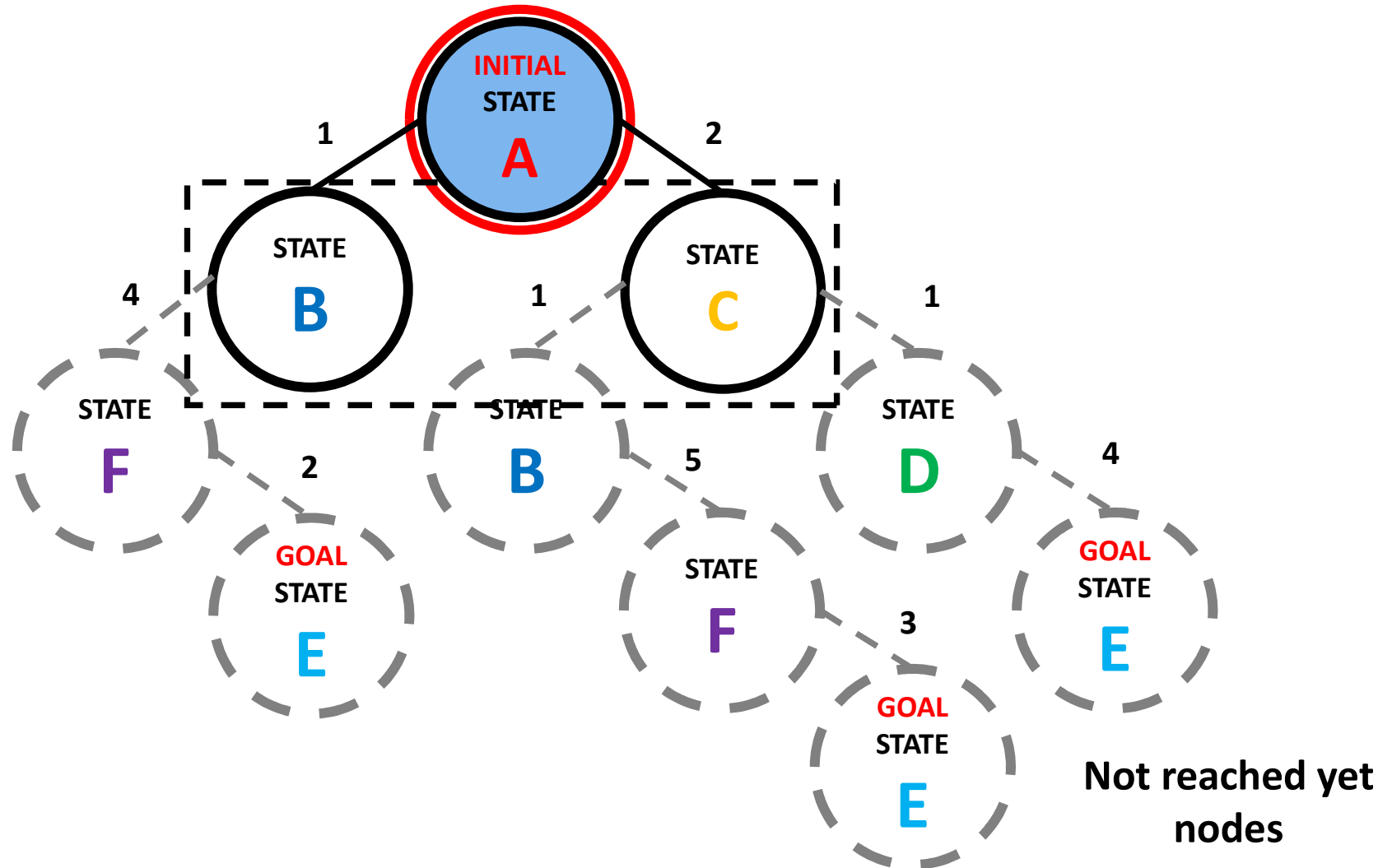
**return** *failure*

**function** UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*

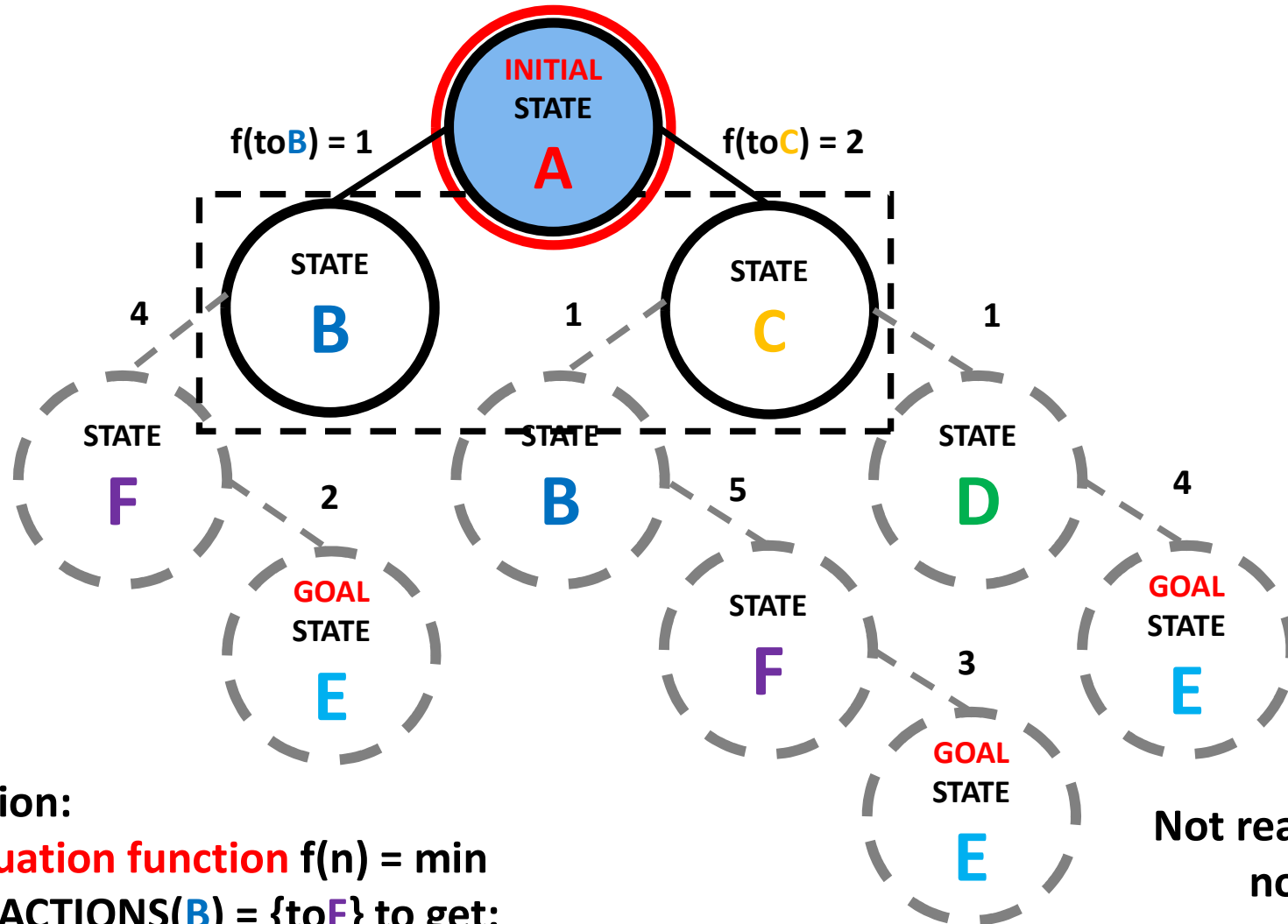
**return** BEST-FIRST-SEARCH(*problem*, PATH-COST)



# Search Tree: Variable Action Cost



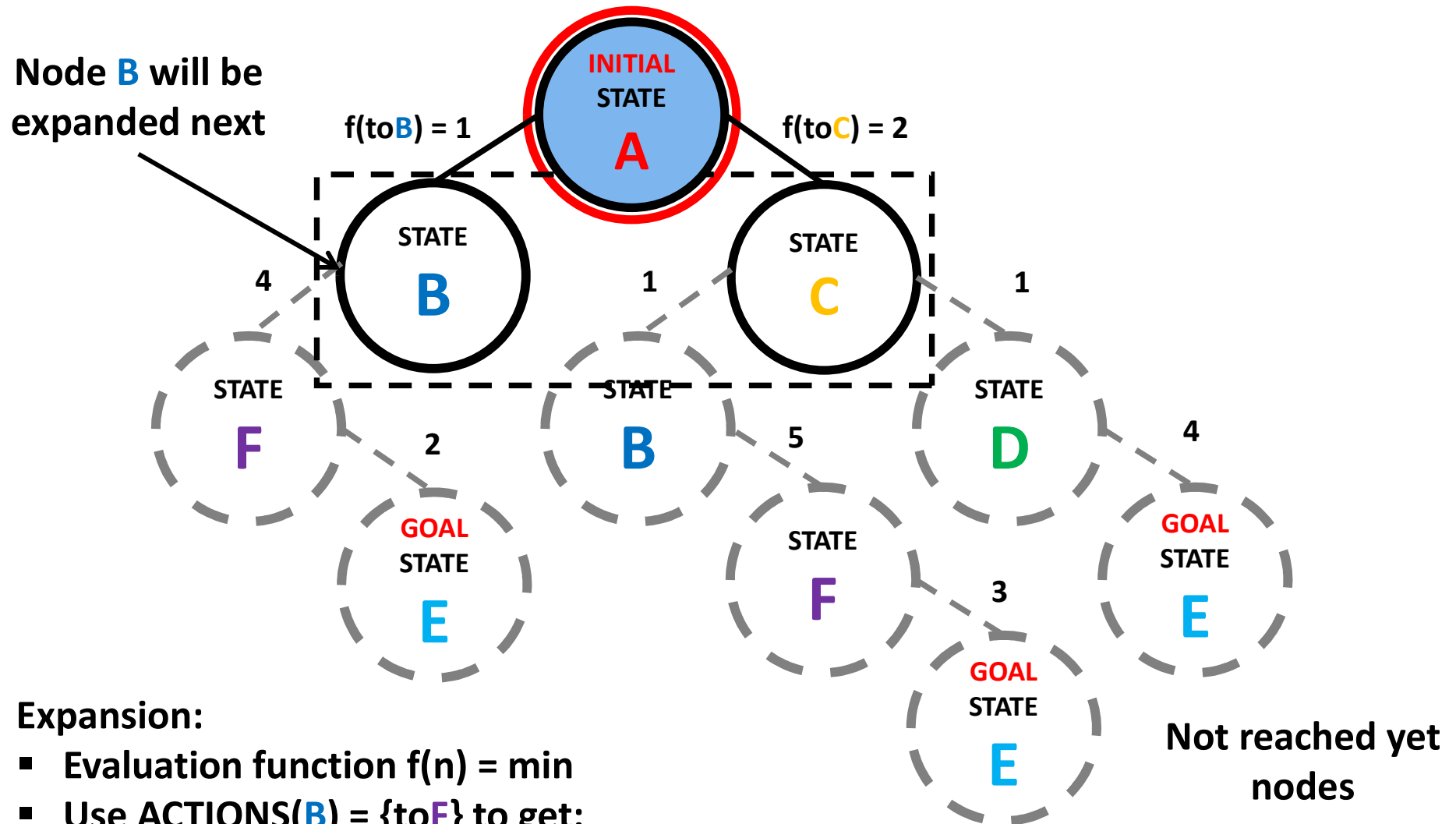
# Search Tree: Variable Action Cost



Expansion:

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

# Search Tree: Best-First Search



Expansion:

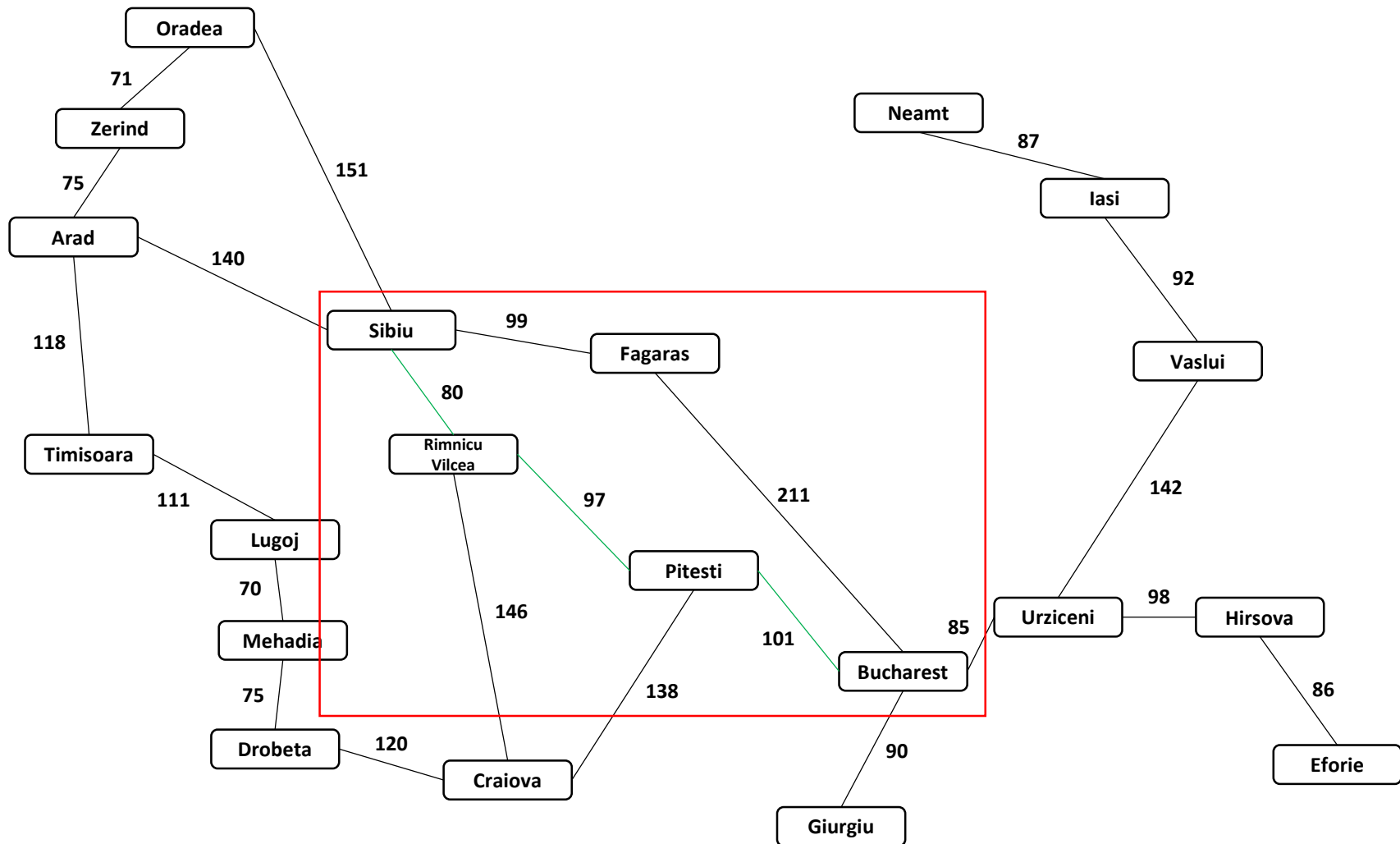
- Evaluation function  $f(n) = \min$
- Use  $\text{ACTIONS}(\mathbf{B}) = \{\text{toF}\}$  to get:
- $\text{RESULT}(\mathbf{B}, \text{toF}) = \mathbf{F}$

# Best-First Search: Pseudocode

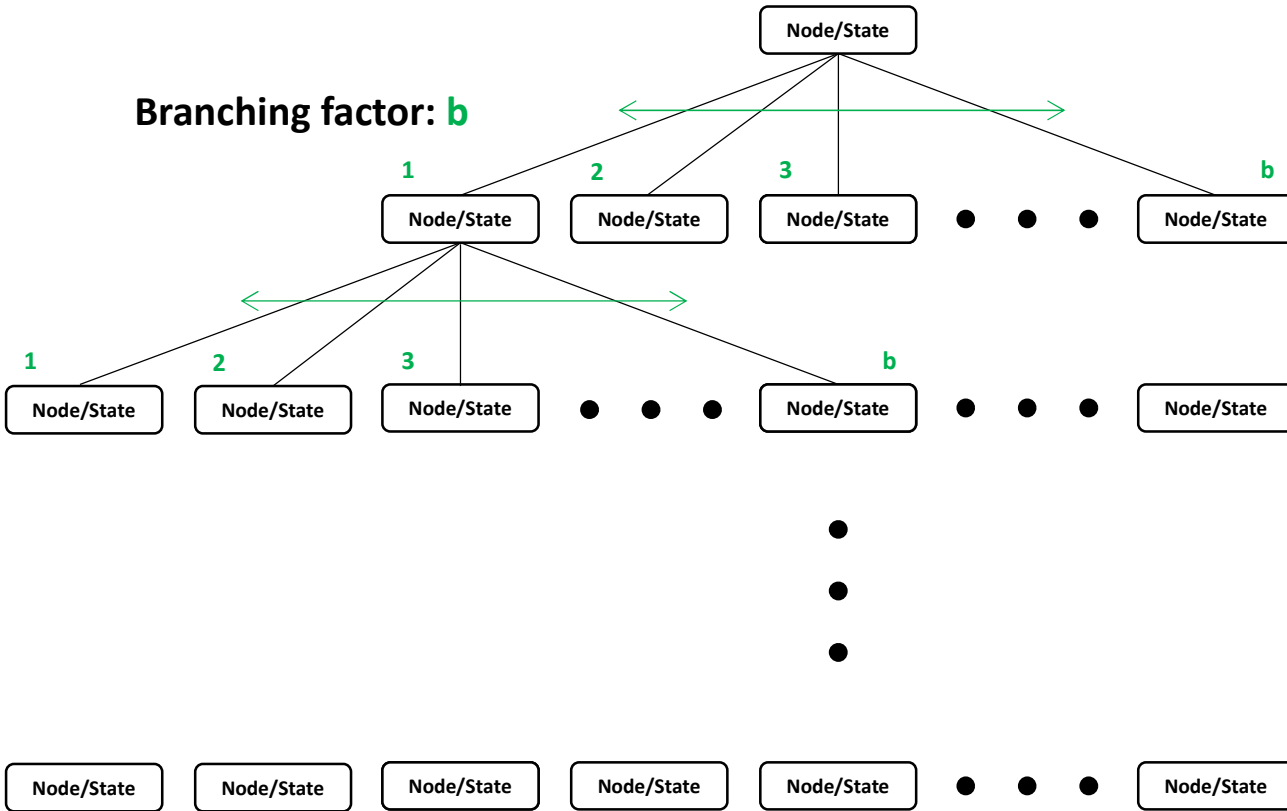
```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure  
  node  $\leftarrow$  NODE(STATE=problem.INITIAL)  
  frontier  $\leftarrow$  a priority queue ordered by f, with node as an element  
  reached  $\leftarrow$  a lookup table, with one entry with key problem.INITIAL and value node  
  while not IS-EMPTY(frontier) do  
    node  $\leftarrow$  POP(frontier)  
    if problem.IS-GOAL(node.STATE) then return node  
    for each child in EXPAND(problem, node) do  
      s  $\leftarrow$  child.STATE  
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then  
        reached[s]  $\leftarrow$  child  
        add child to frontier  
  return failure
```

```
function EXPAND(problem, node) yields nodes  
  s  $\leftarrow$  node.STATE  
  for each action in problem.ACTIONS(s) do  
    s'  $\leftarrow$  problem.RESULT(s, action)  
    cost  $\leftarrow$  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**  $\infty$  **do**  
    *result*  $\leftarrow$  DEPTH-LIMITED-SEARCH(*problem*, *depth*)  
    **if** *result*  $\neq$  *cutoff* **then return** *result*

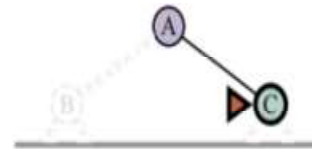
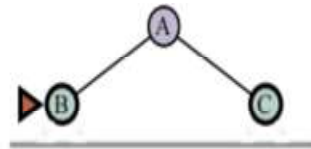
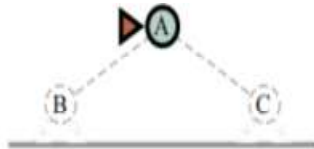
**function** DEPTH-LIMITED-SEARCH(*problem*,  $\ell$ ) **returns** a node or *failure* or *cutoff*  
  *frontier*  $\leftarrow$  a LIFO queue (stack) with NODE(*problem*.INITIAL) as an element  
  *result*  $\leftarrow$  *failure*  
  **while not** IS-EMPTY(*frontier*) **do**  
    *node*  $\leftarrow$  POP(*frontier*)  
    **if** *problem*.IS-GOAL(*node*.STATE) **then return** *node*  
    **if** DEPTH(*node*) >  $\ell$  **then**  
      *result*  $\leftarrow$  *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

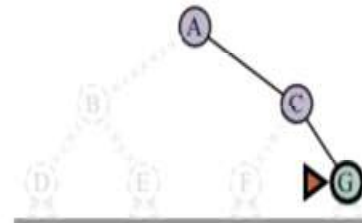
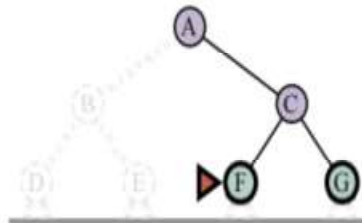
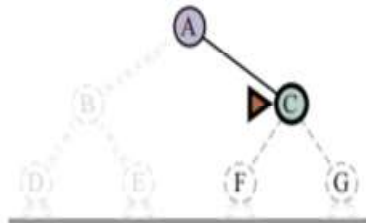
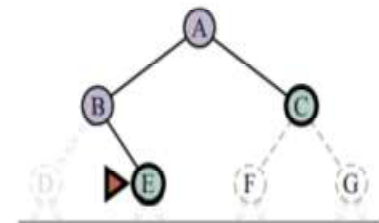
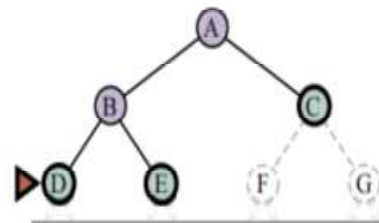
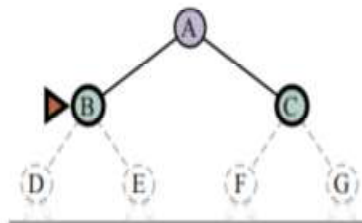
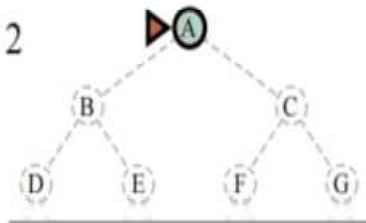
limit: 0



limit: 1



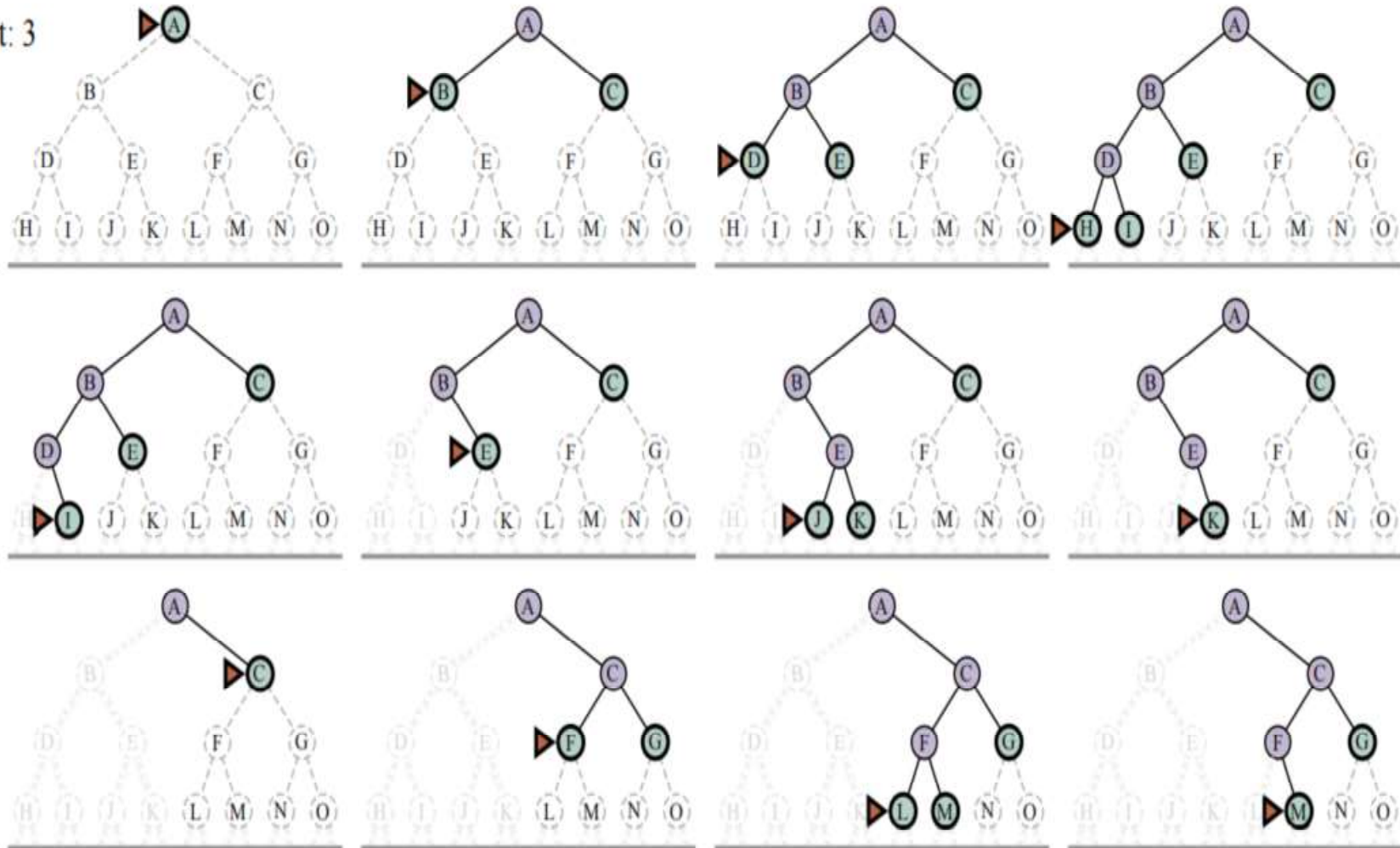
limit: 2





# Iterative Deepening DFS: Illustration

limit: 3

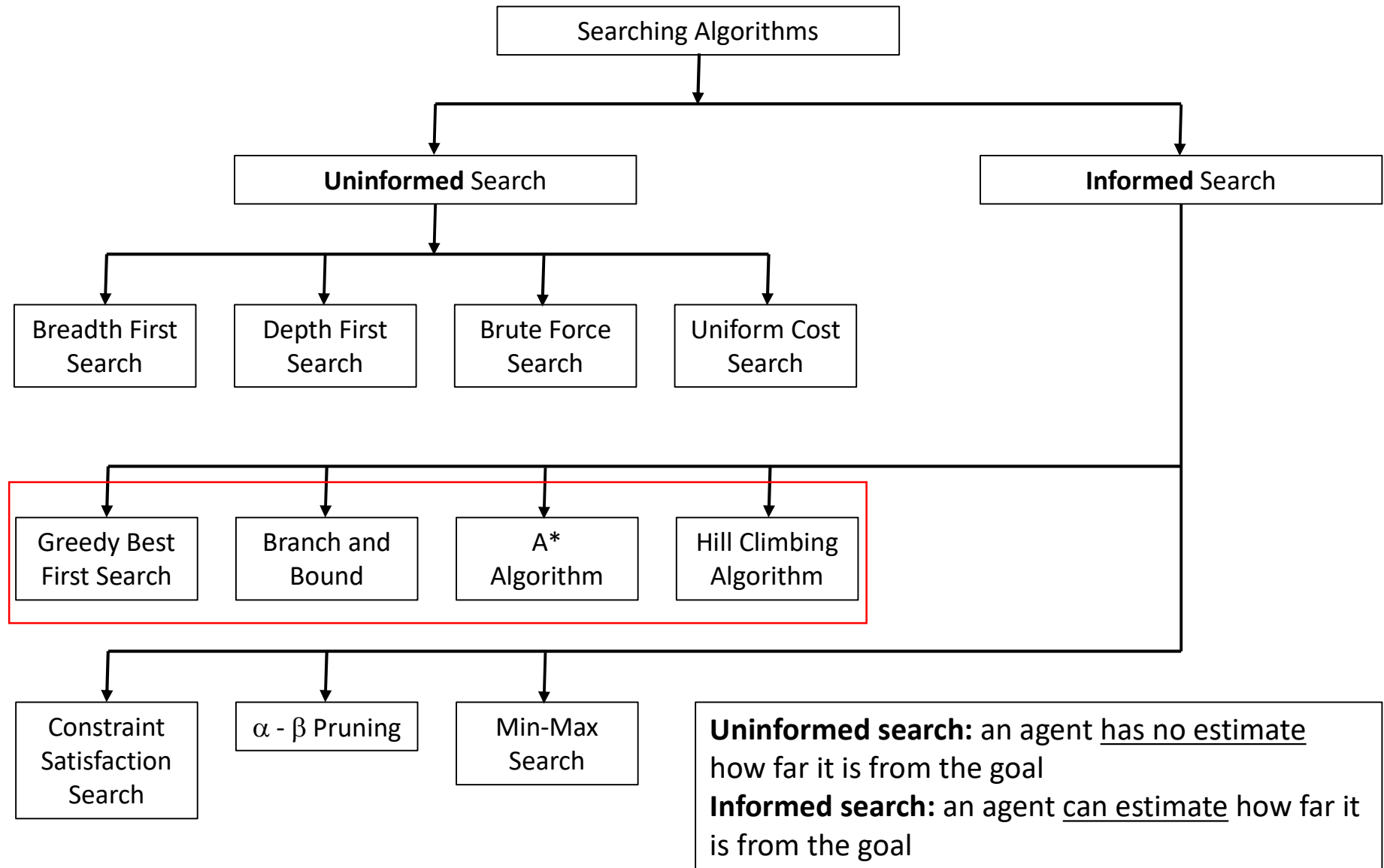


# Uninformed Search Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes <sup>1</sup>	Yes <sup>1,2</sup>	No	No	Yes <sup>1</sup>	Yes <sup>1,4</sup>
Optimal cost?	Yes <sup>3</sup>	Yes	No	No	Yes <sup>3</sup>	Yes <sup>3,4</sup>
Time	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

**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;  $\ell$  is the depth limit. Superscript caveats are as follows: <sup>1</sup> complete if  $b$  is finite, and the state space either has a solution or is finite. <sup>2</sup> complete if all action costs are  $\geq \epsilon > 0$ ; <sup>3</sup> cost-optimal if action costs are all identical; <sup>4</sup> if both directions are breadth-first or uniform-cost.

# Selected Searching Algorithms



# Informed Search and Heuristics

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

$$h(n) = h(\text{State } n)$$

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

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