CS 480

Introduction to Artificial Intelligence

January 25, 2022

Announcements / Reminders

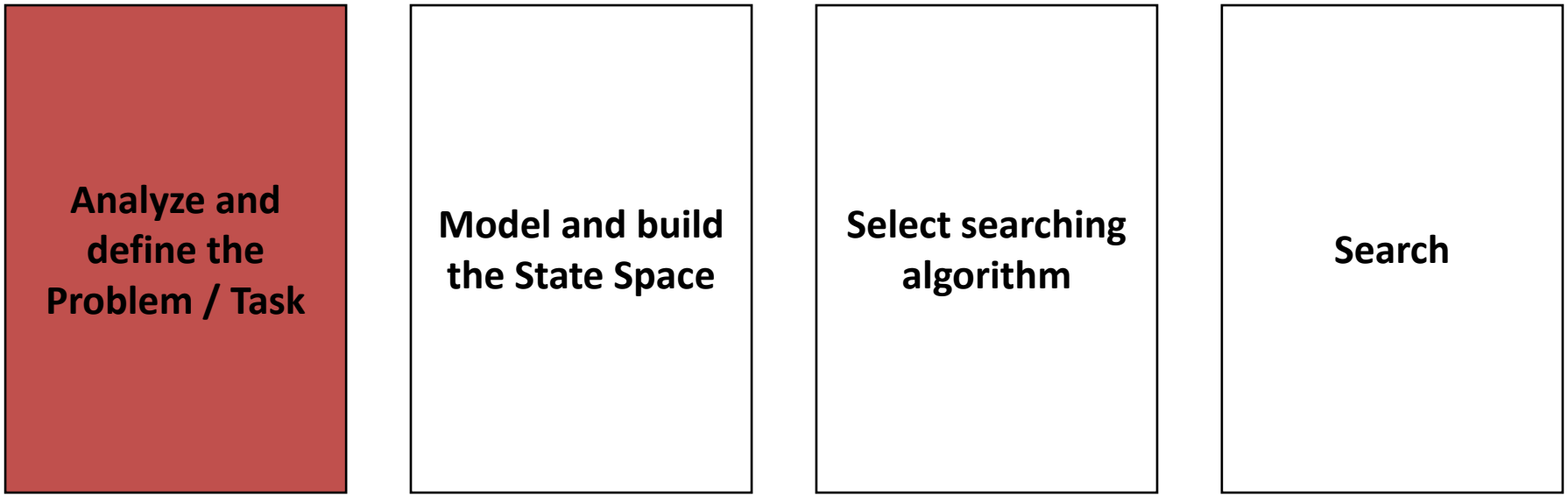
- Please follow the Week 02 To Do List instructions (if you haven't already)

- In-person sessions start THIS week
  - Please follow IIT COVID-19 guidelines
- Contribute to the discussion on Blackboard, please
- Thursday: data structures, etc. review session
- My office hours:
  - Tuesdays 11:30 AM - 01:30 PM in Stuart Building 217E or by appointment

Plan for Today

- **Problem Solving: Searching**

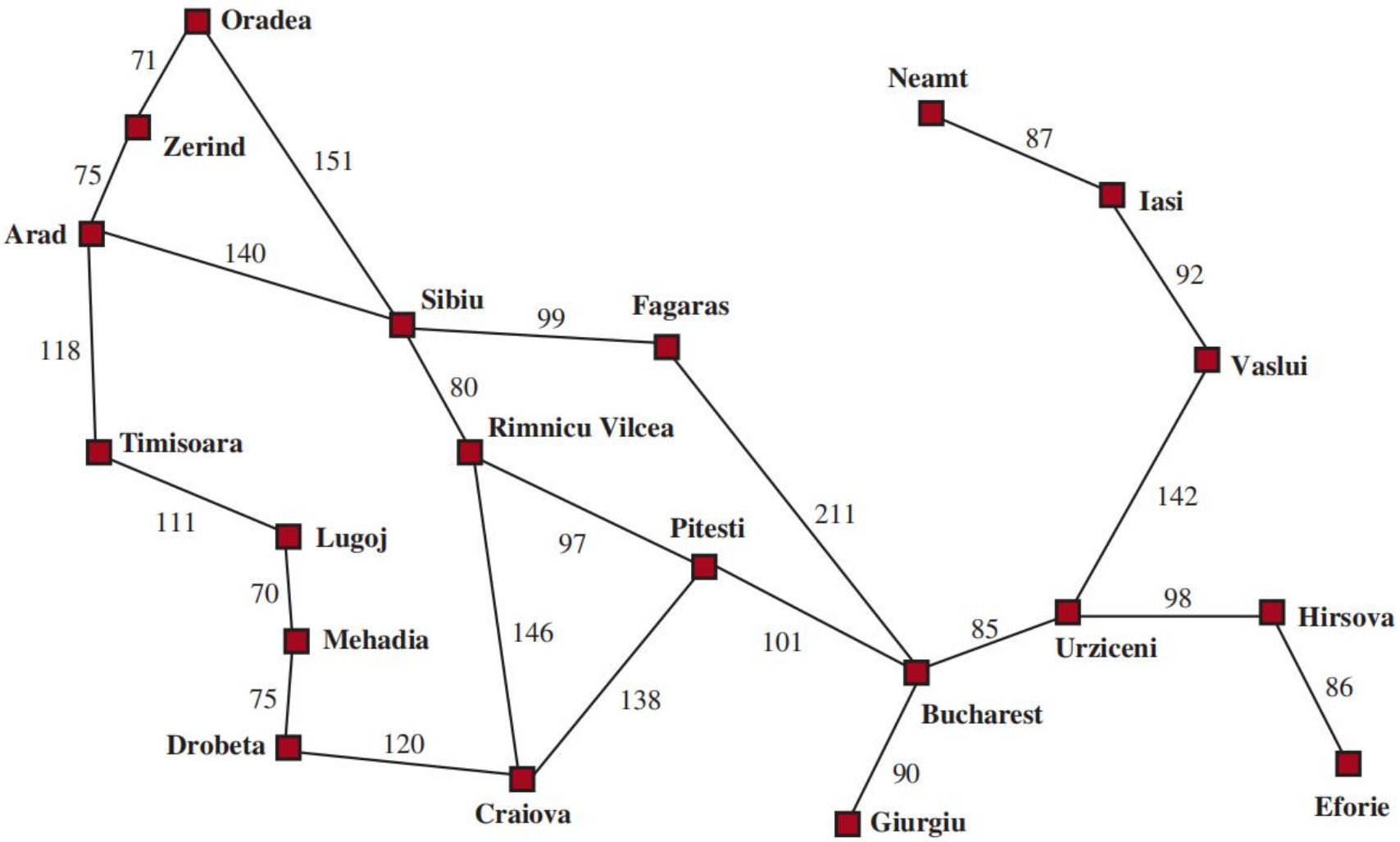
Designing the Searching Problem



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

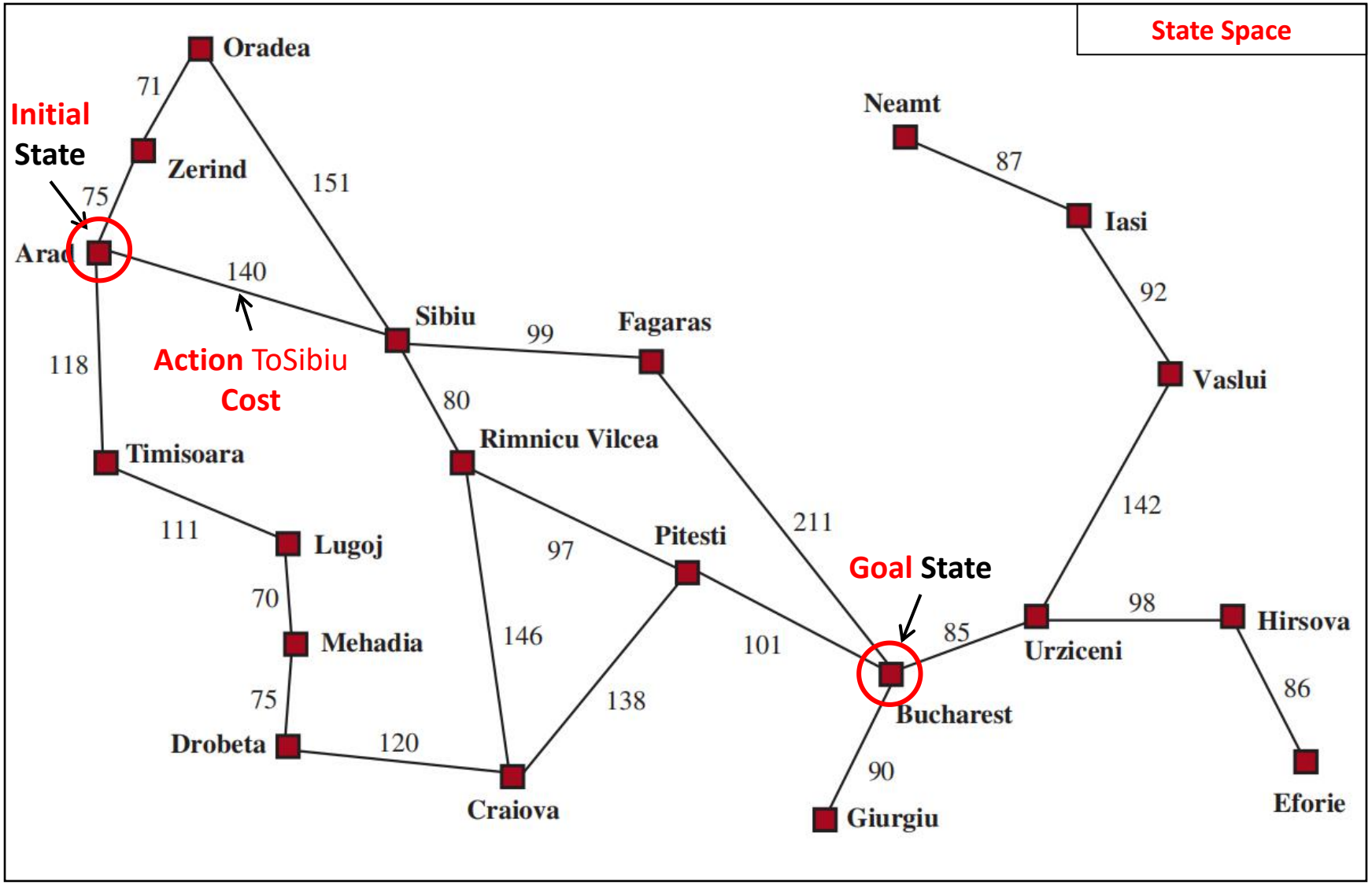


Problem: Get from Arad to Bucharest efficiently (for example: quickly or cheaply).

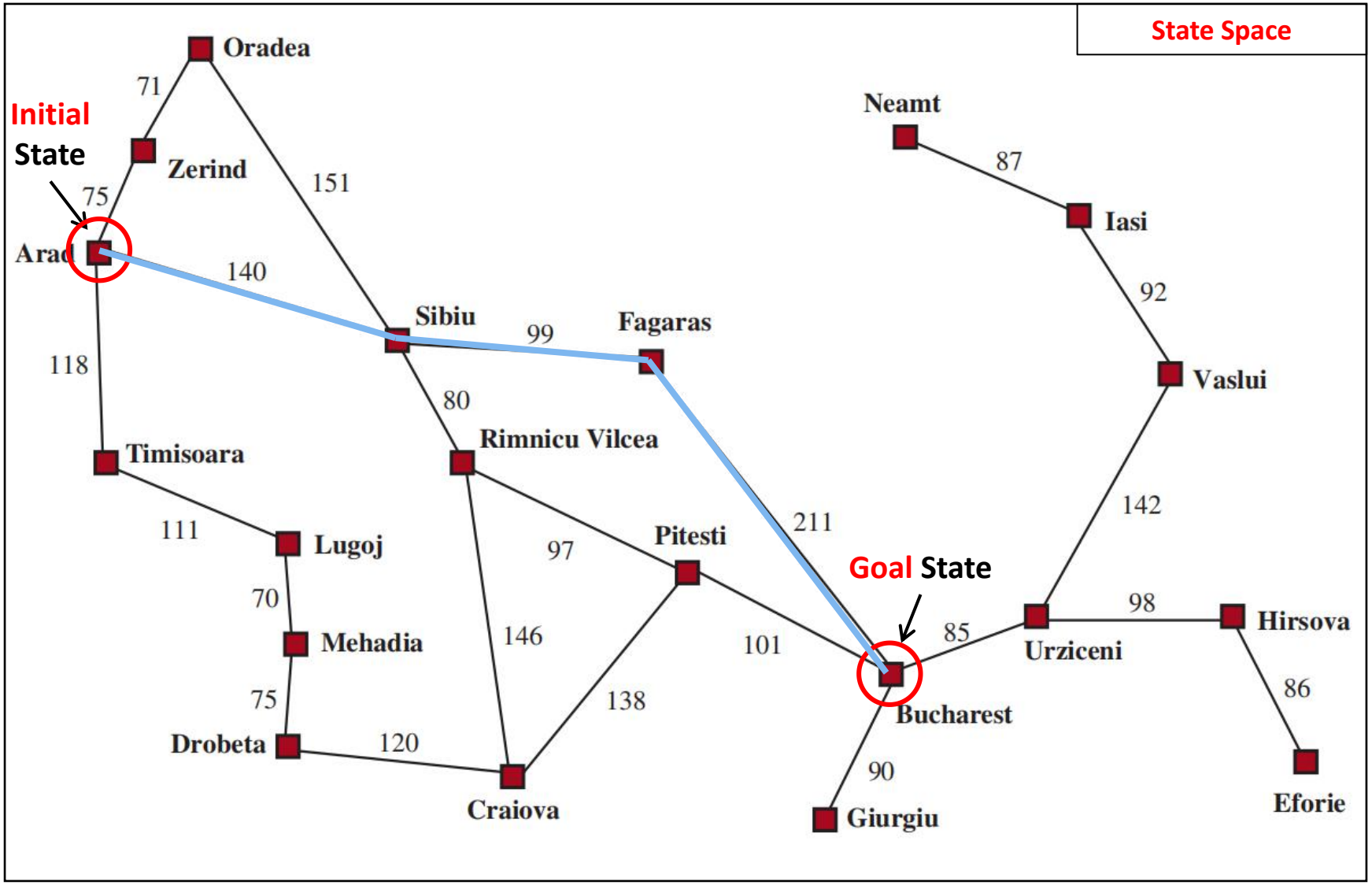
Search Problem: Dracula's Roadtrip

- 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

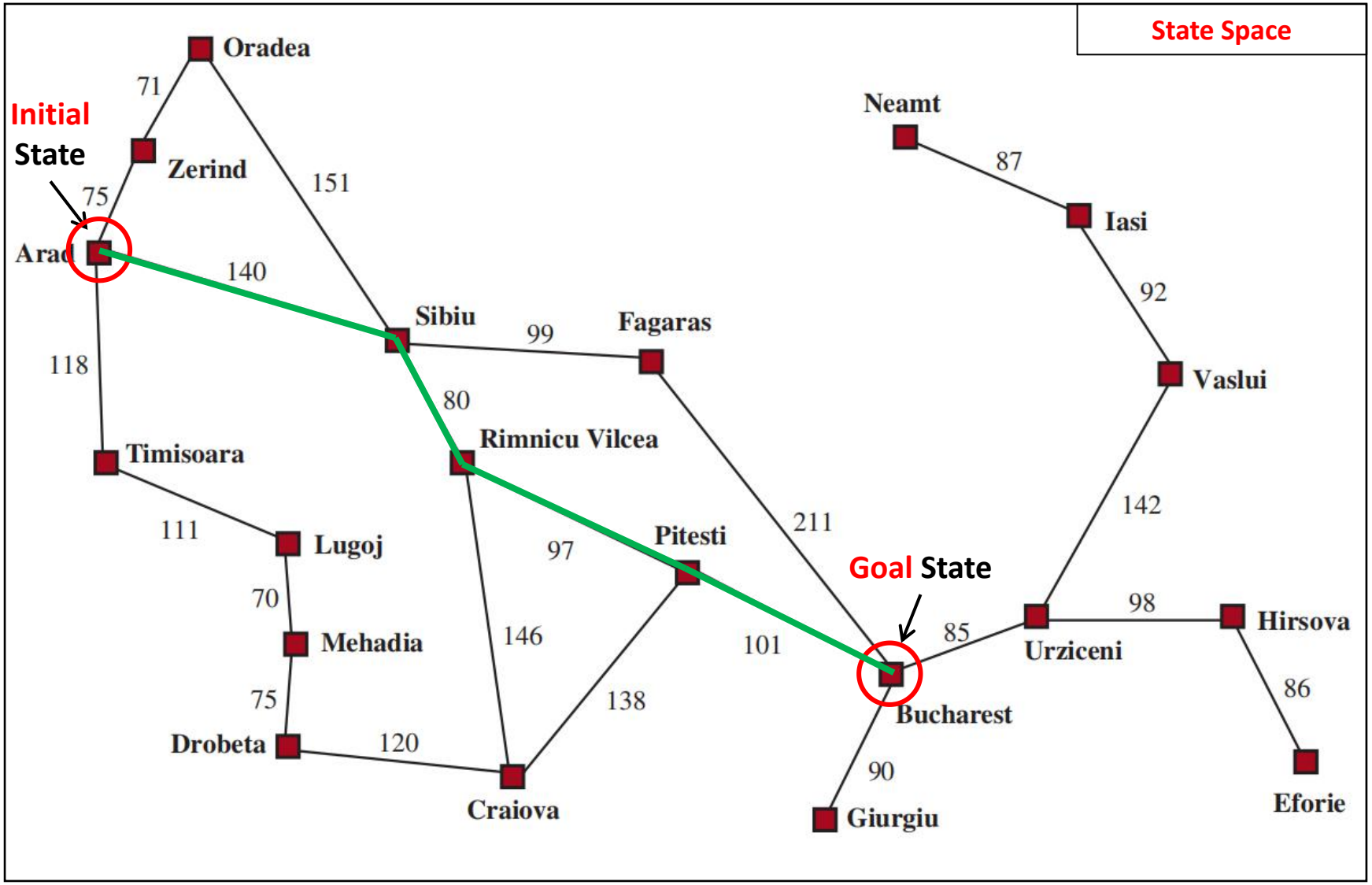
Sample Problem: Dracula's Roadtrip



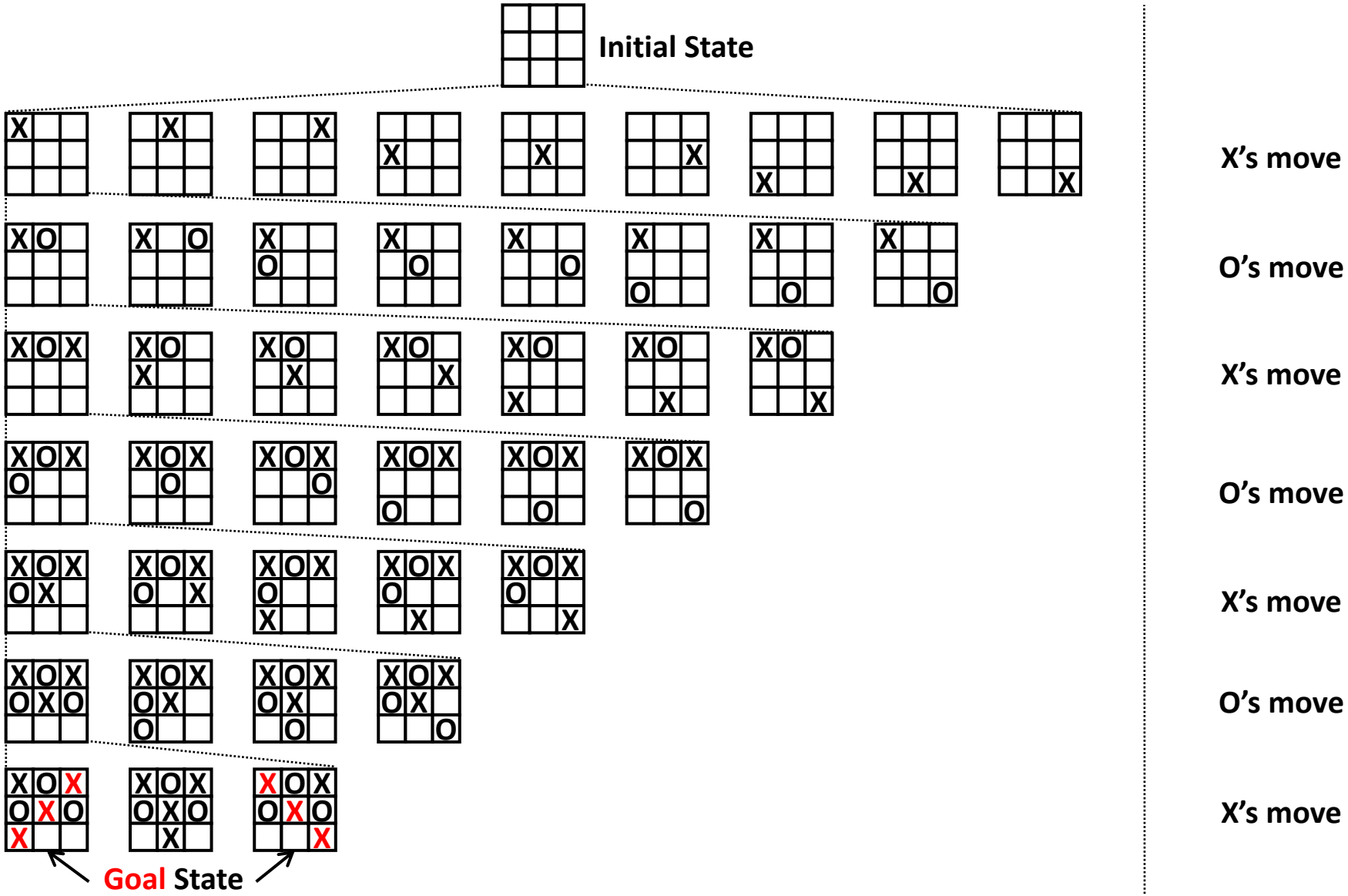
Dracula's Roadtrip: Potential Solution



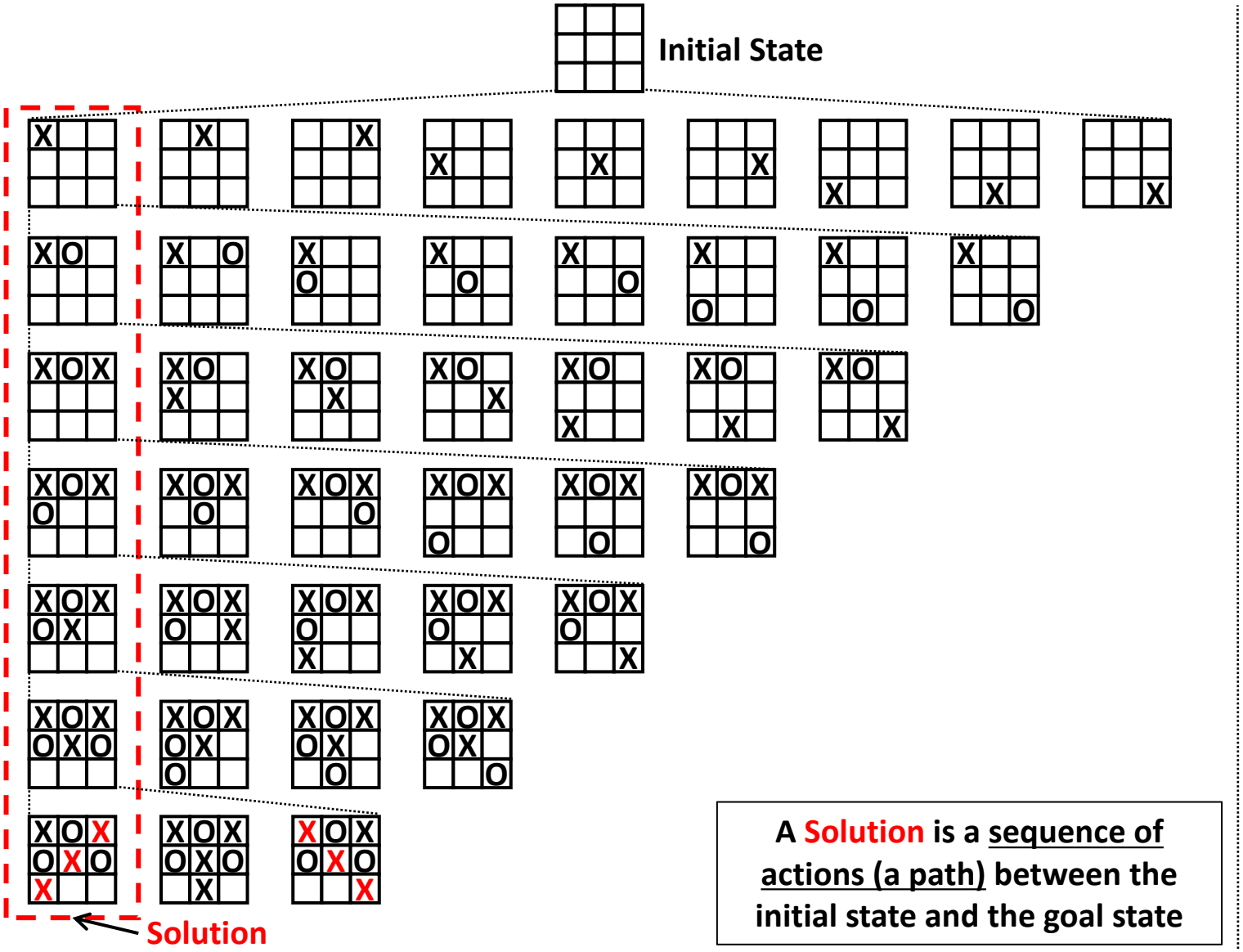
Dracula's Roadtrip: Potential Solution



Tic Tac Toe: (Partial) State Space



Tic Tac Toe: Solution

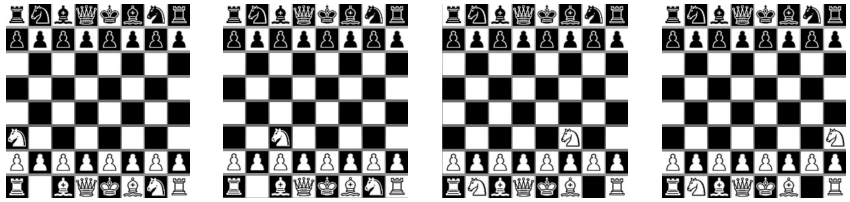


A Solution is a sequence of actions (a path) between the initial state and the goal state

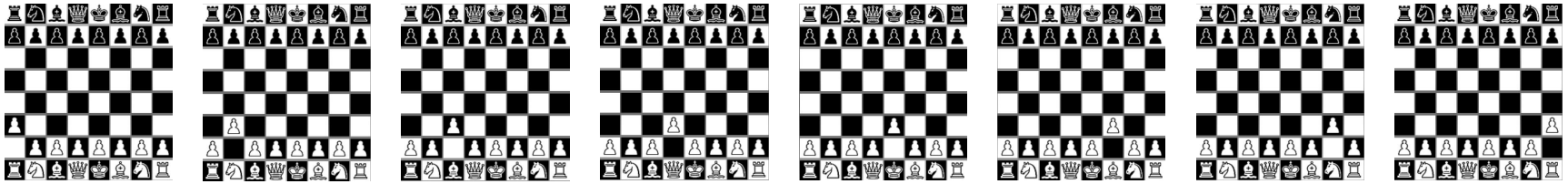
Chess: (First Move) State Space

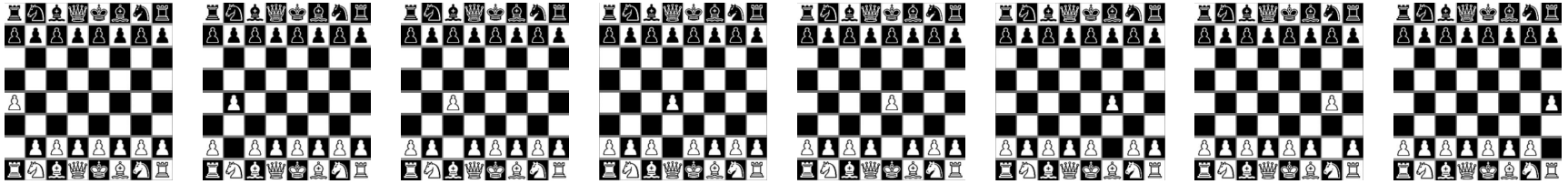
Initial State

Move

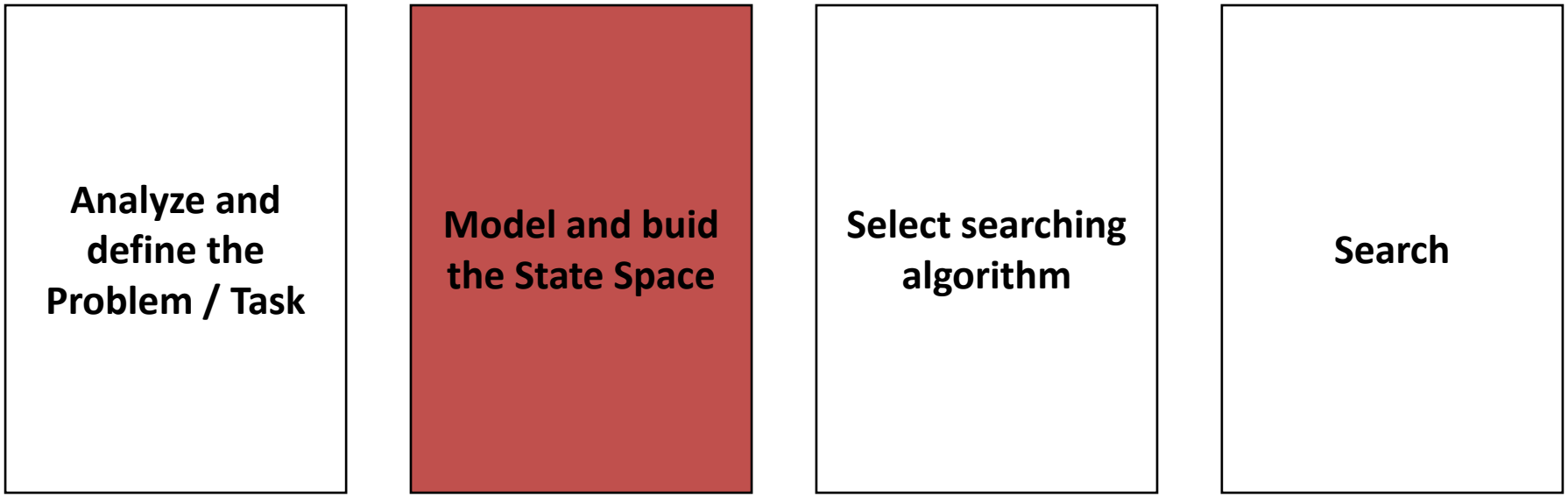


20 Possible legal first moves:
16 pawn moves
4 knight moves

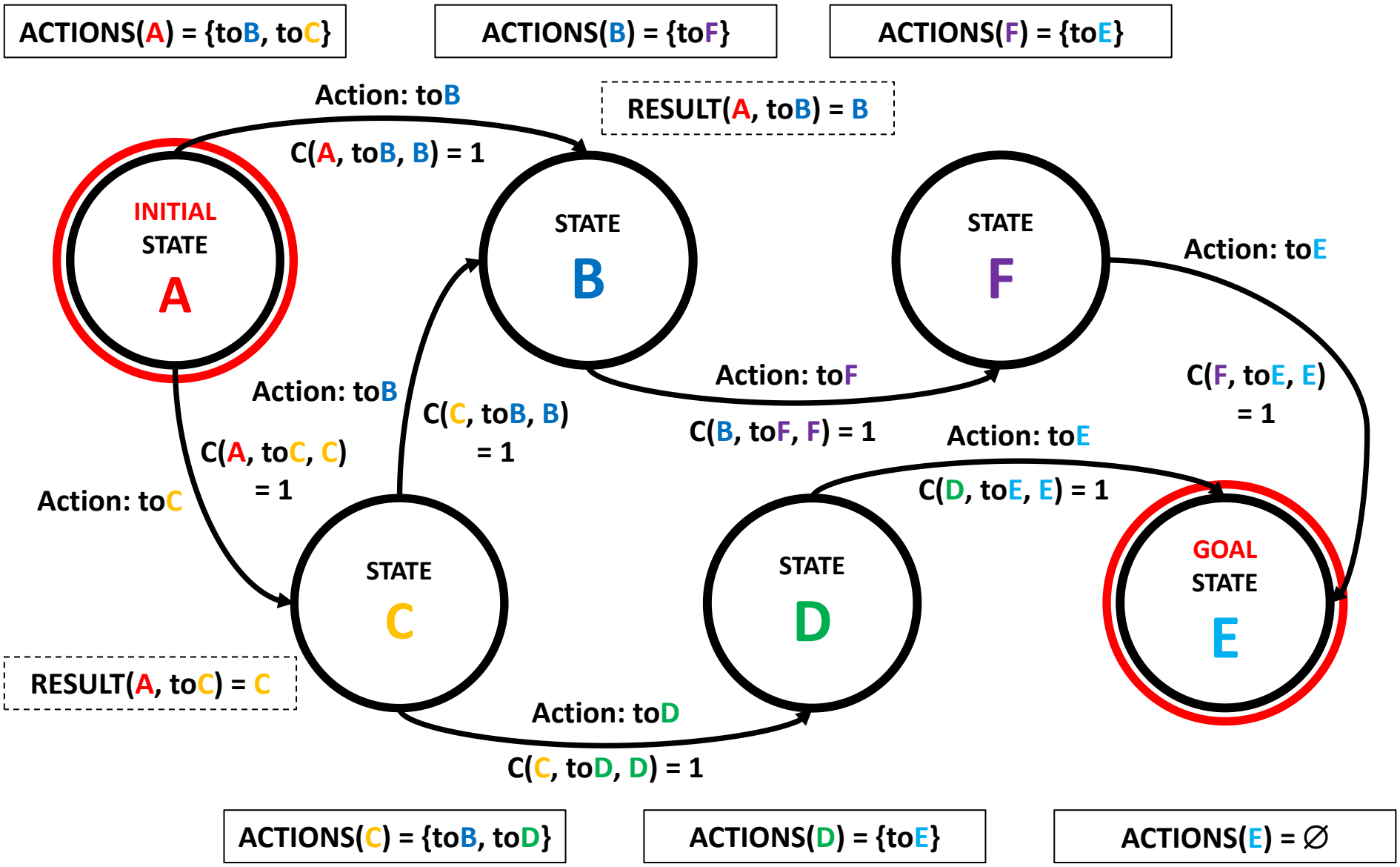




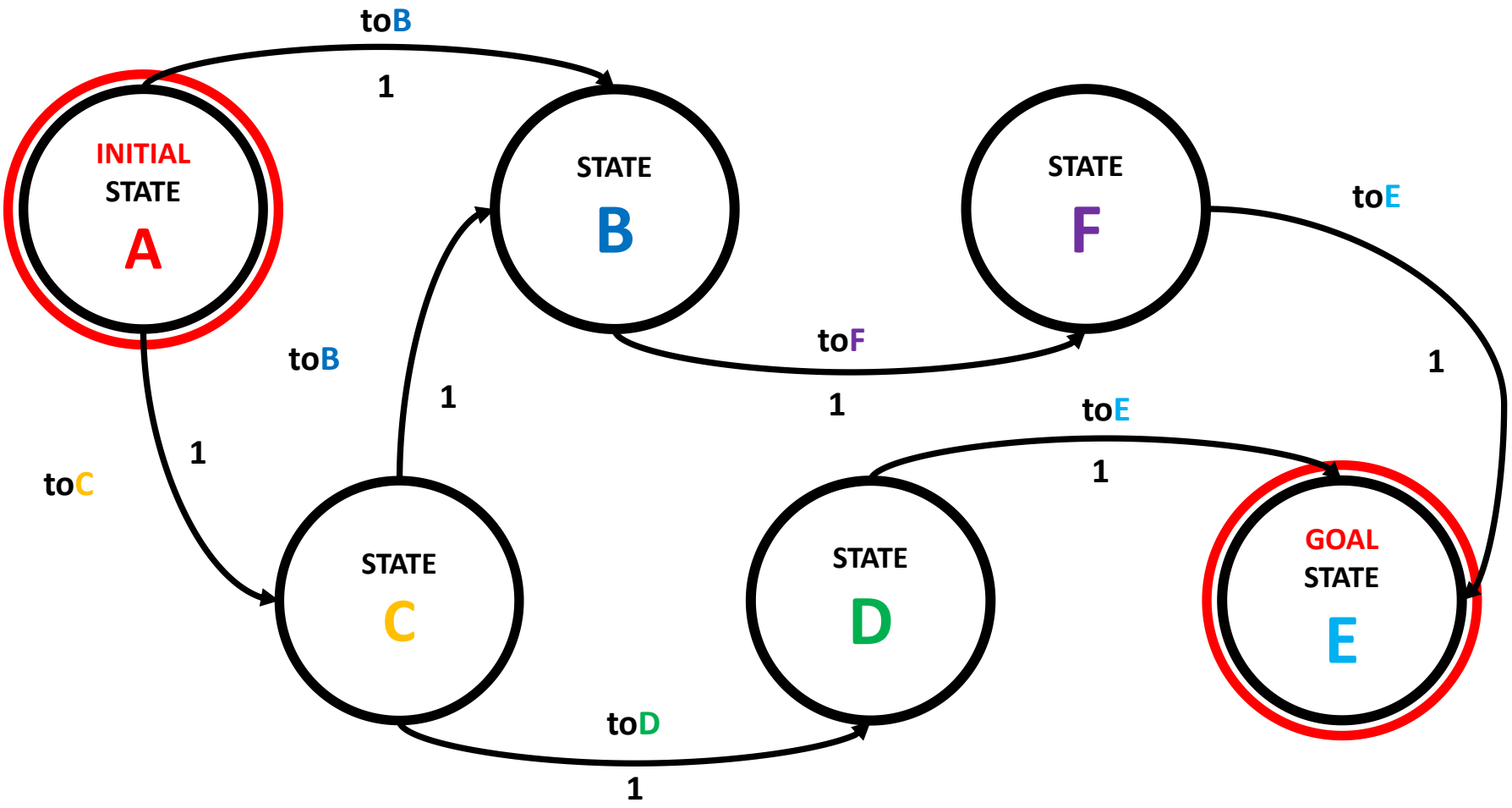
Designing the Searching Problem



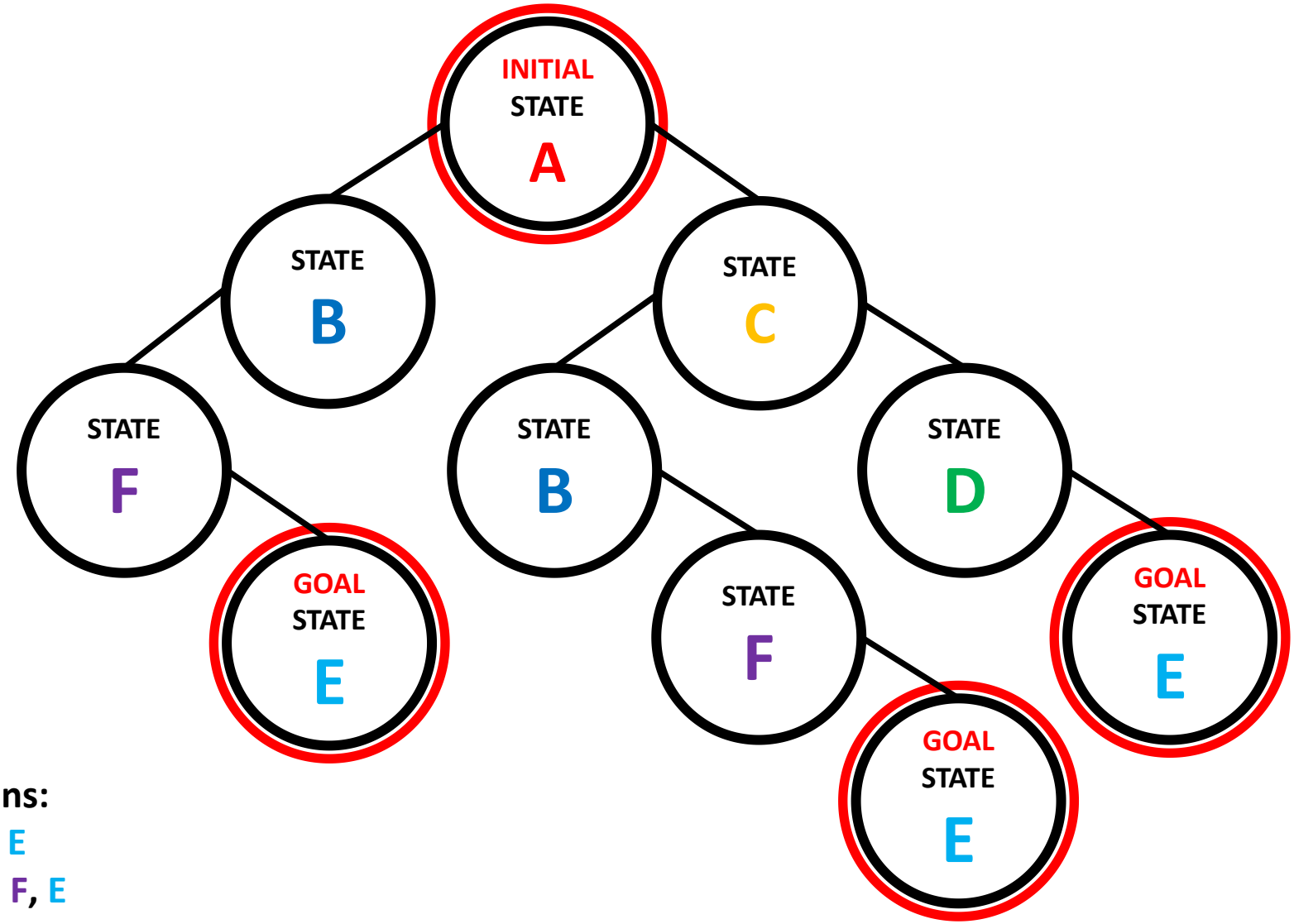
State Space Model: A Graph



State Space Model: A Graph



Searching State Space: Search Tree



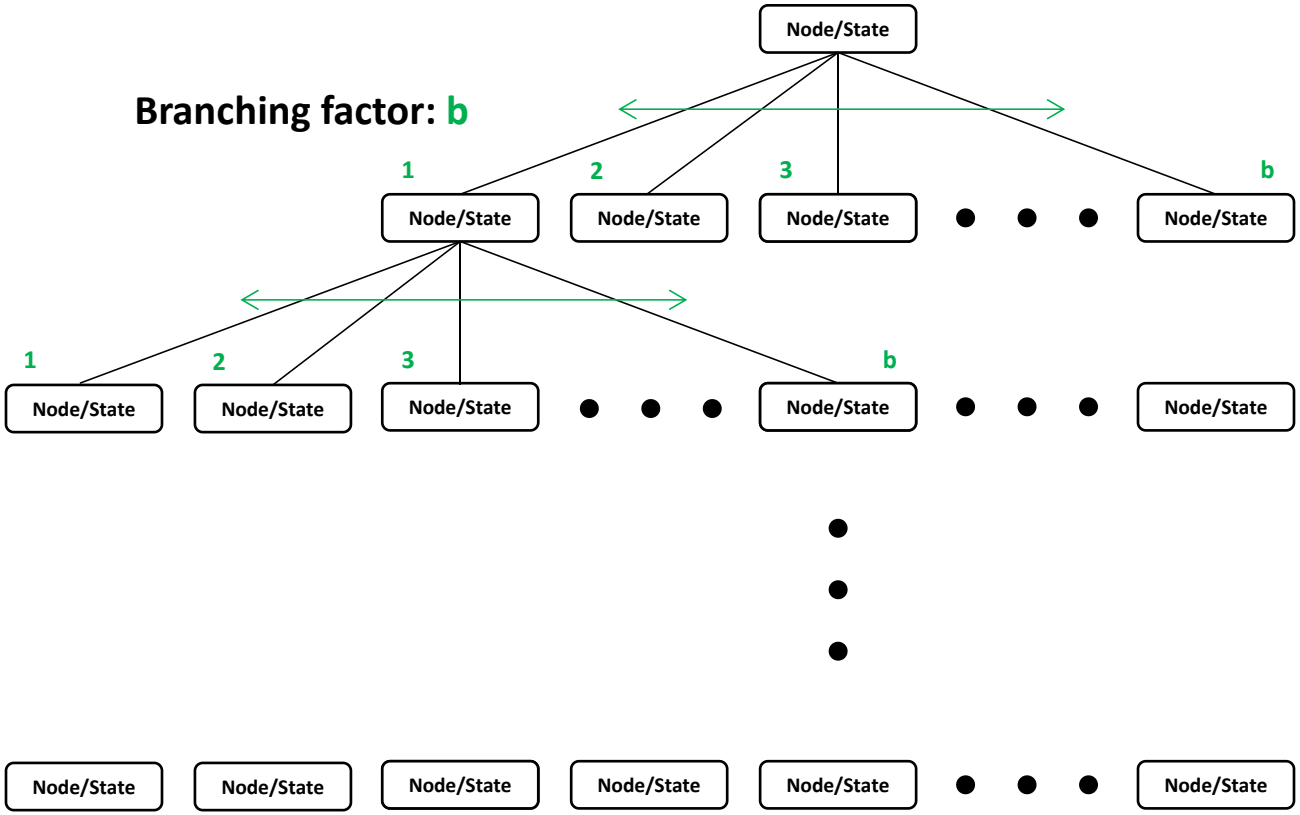
Solutions:

A, B, F, E

A, C, B, F, E

A, C, D, E

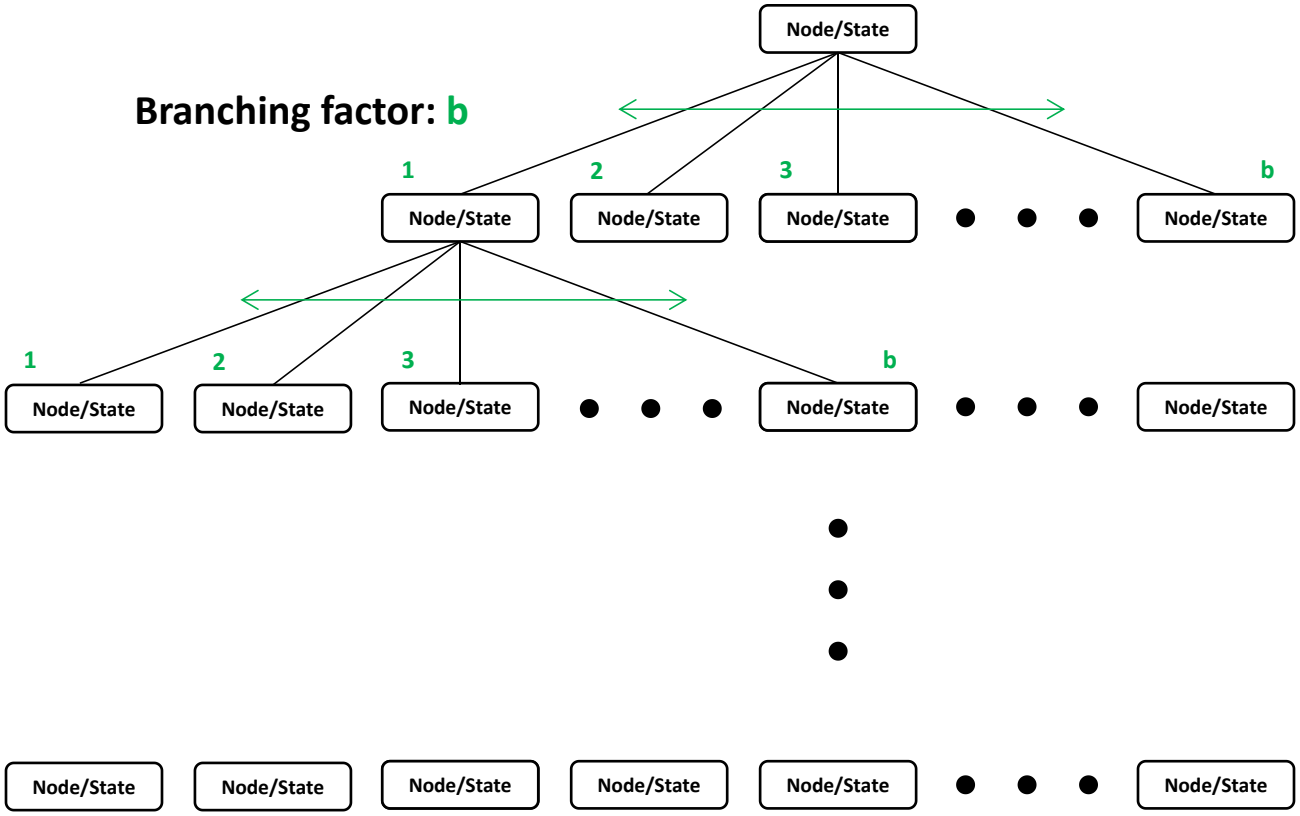
Search Tree Challenges: Size



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!

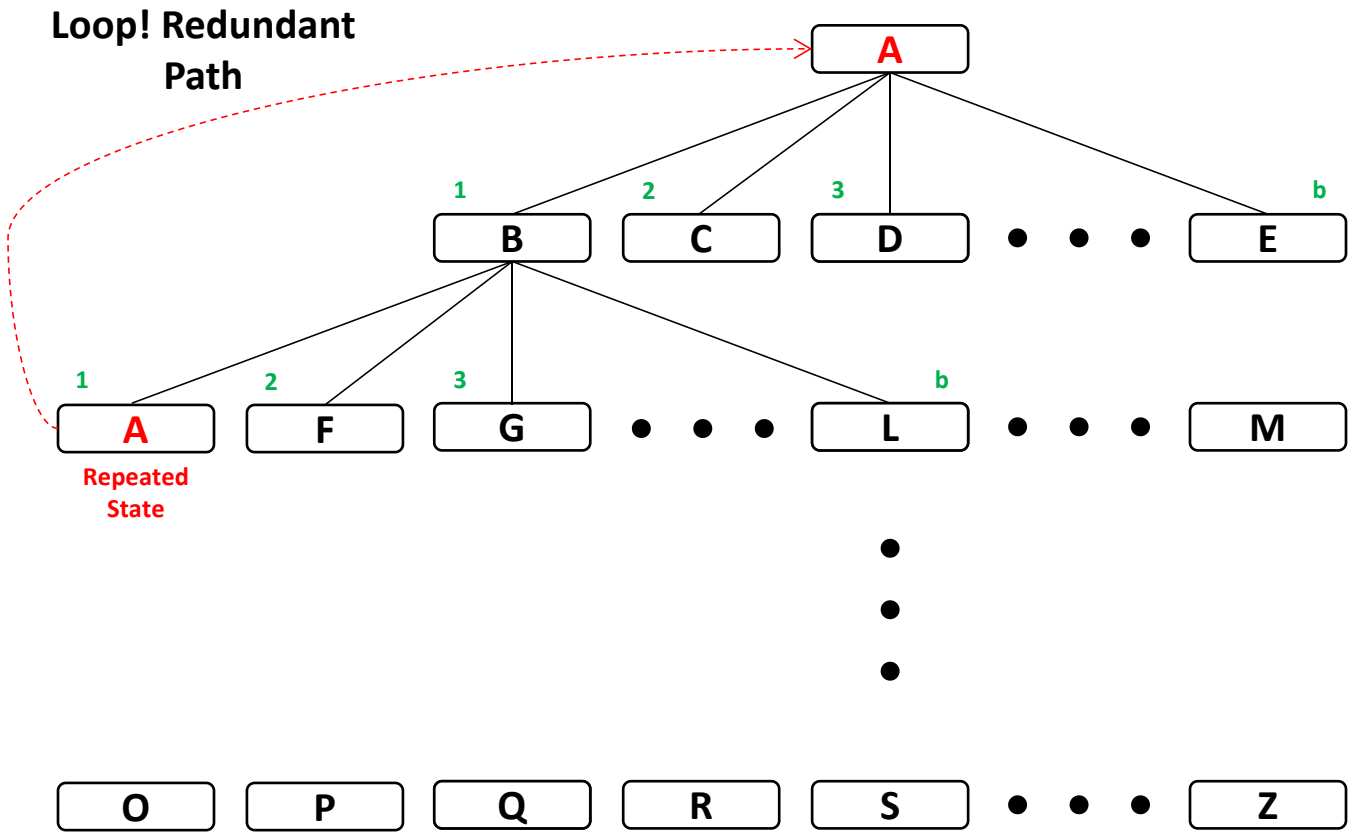
Search Tree Challenges: Infiniteness



Unmanageable and impossible to search with brute force!

Memory and time use grows quickly!

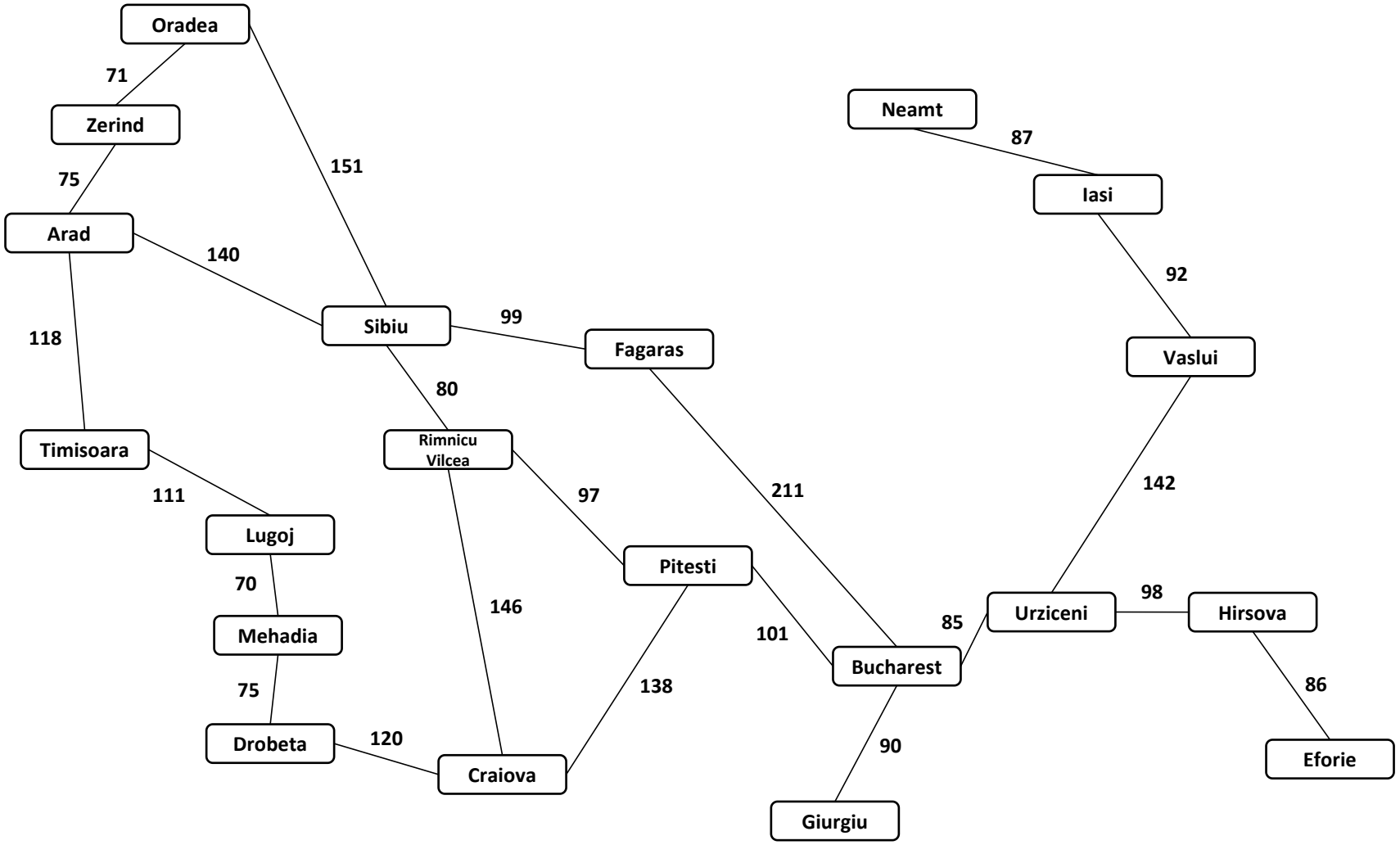
Search Tree Challenges: Loops



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

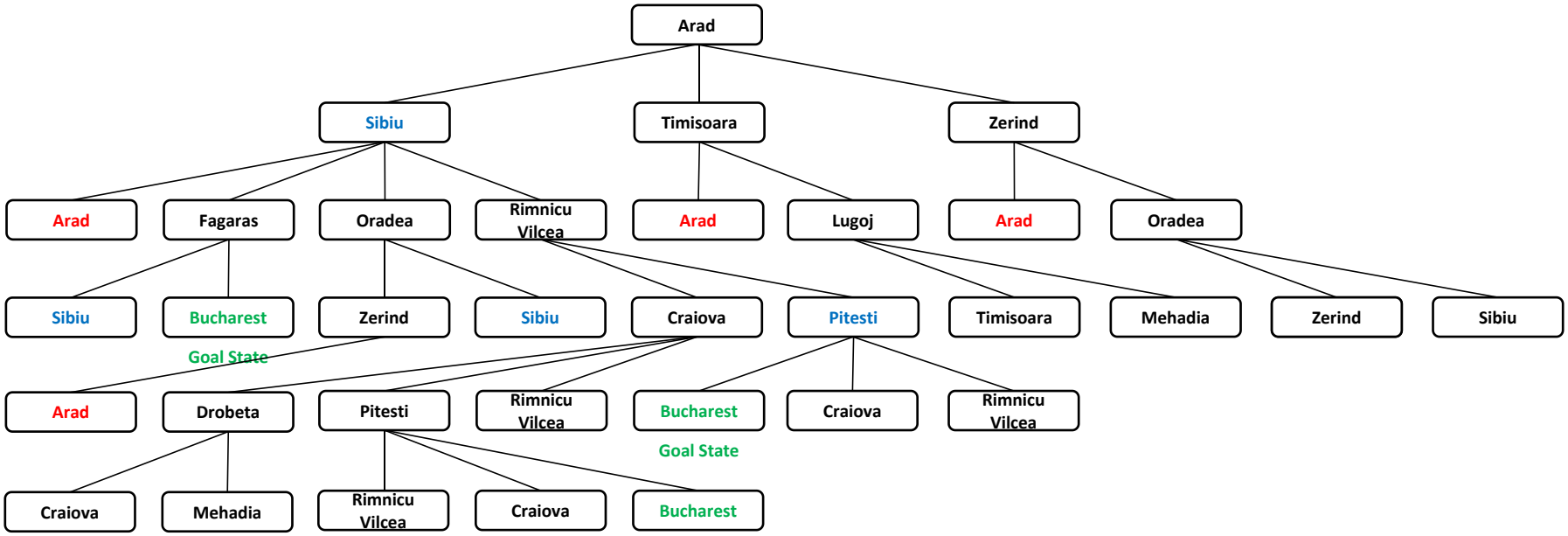
Memory and time use grows quickly!

Sample Problem: Dracula's Roadtrip



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

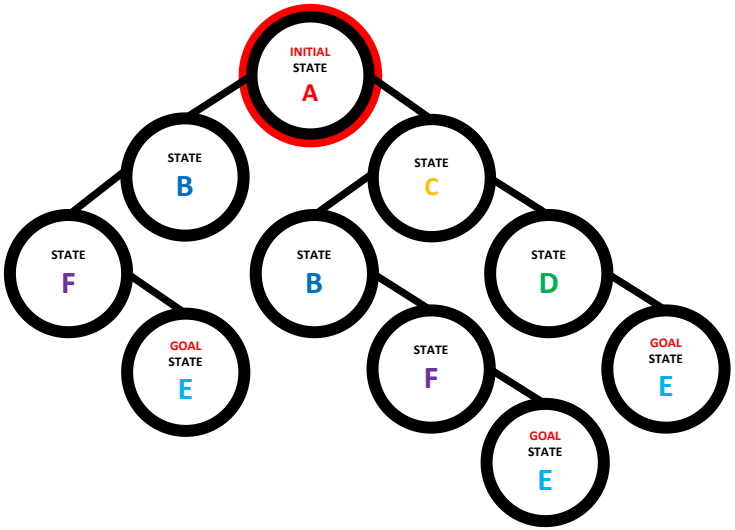
Dracula's Roadtrip as a Tree



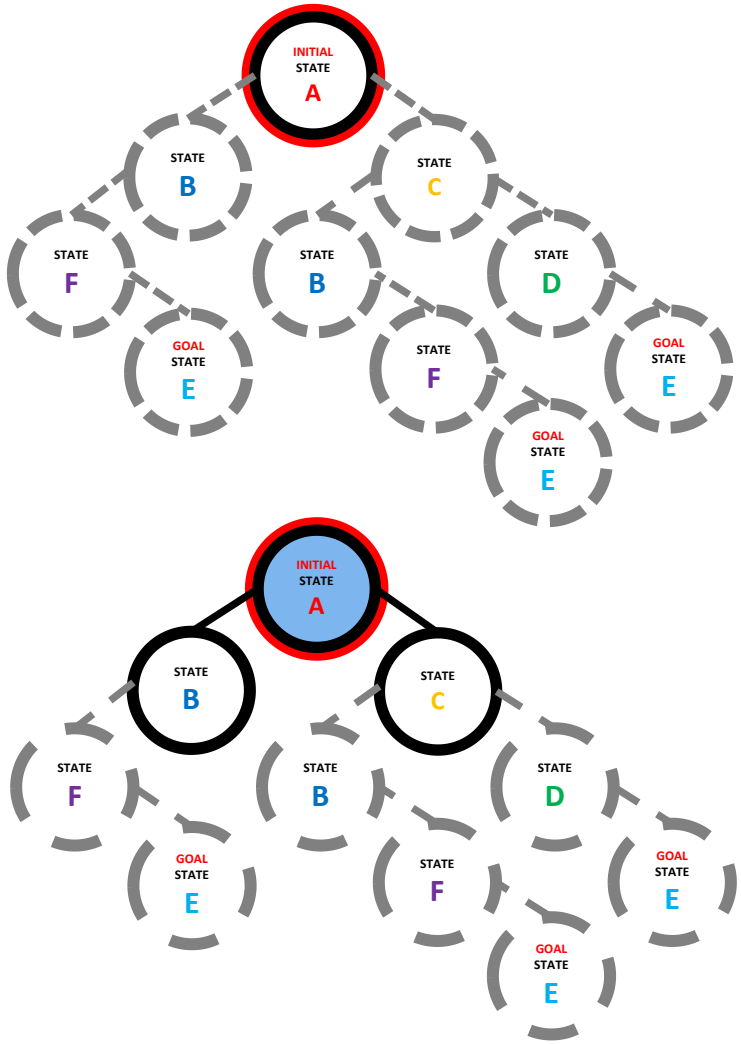
INCOMPLETE! I need to redraw it in smarter way

Search Tree: Implementations

Build entire search tree



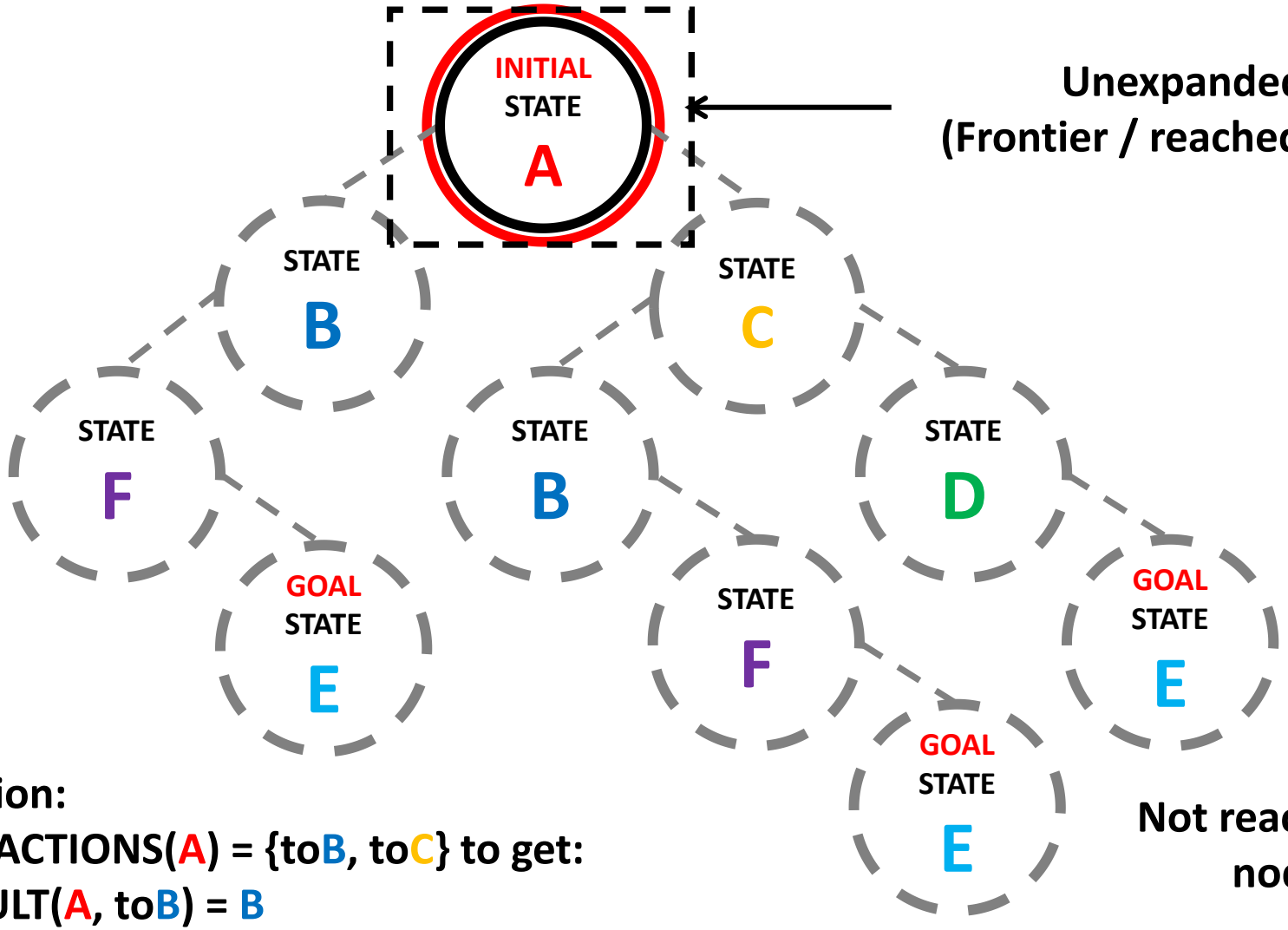
Expand/generate nodes as you go



Challenges:

- memory requirements
- impossible for infinite number of states

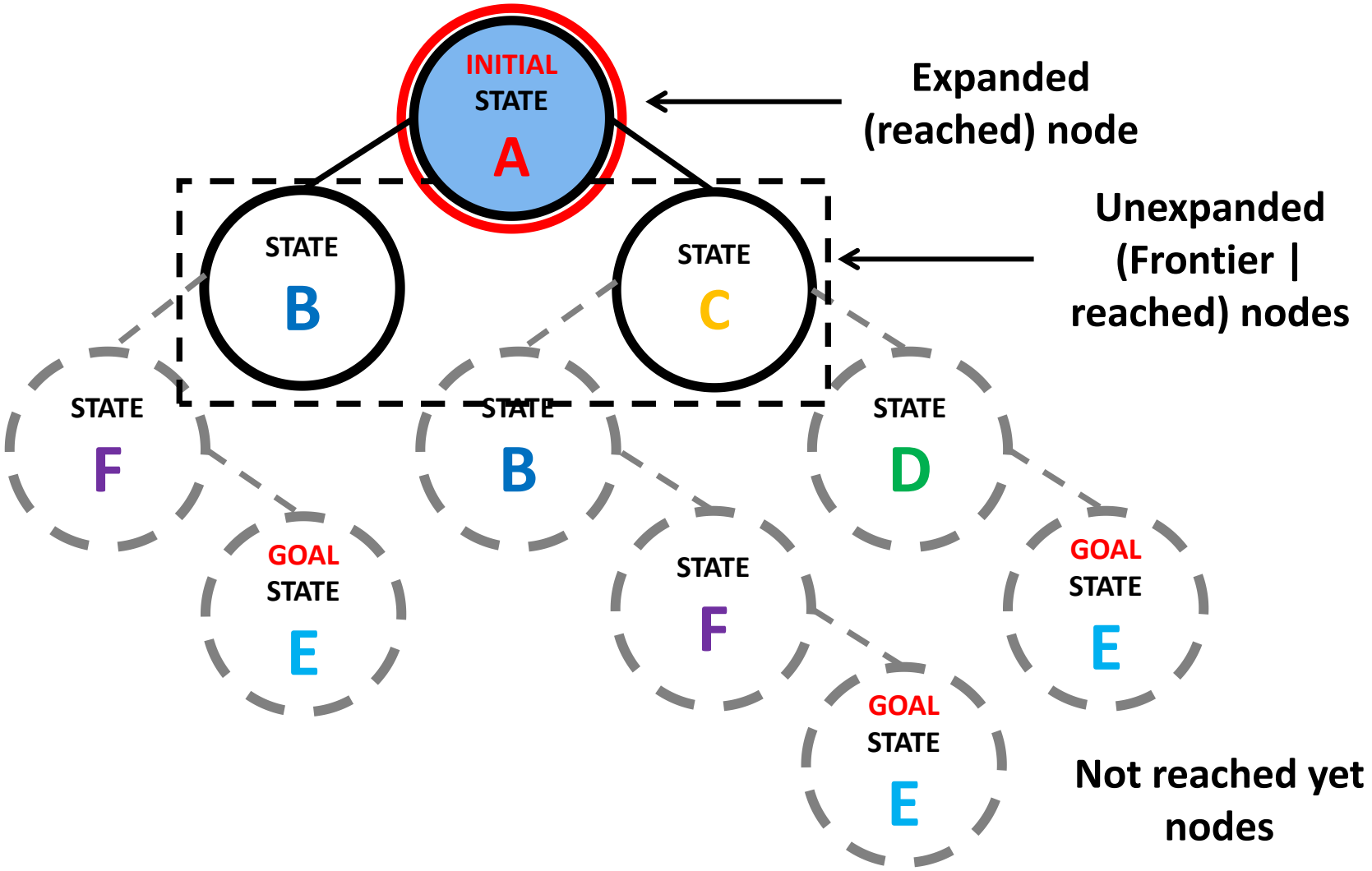
Search Tree: Node Expansion



Expansion:

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

Search Tree: Node Expansion

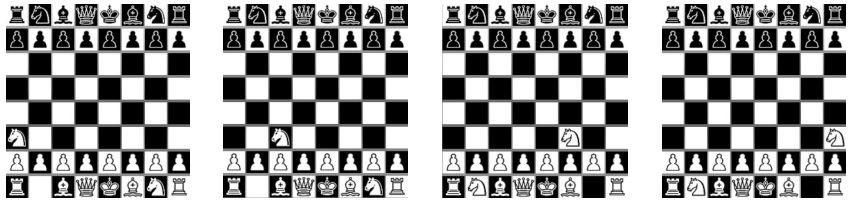


Chess: State Node Expansion

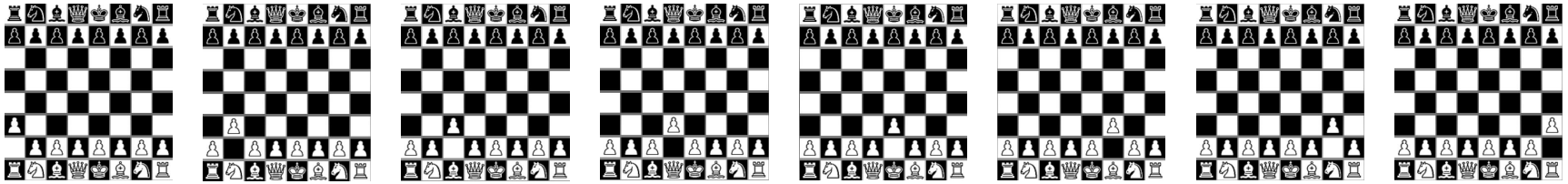
Initial State

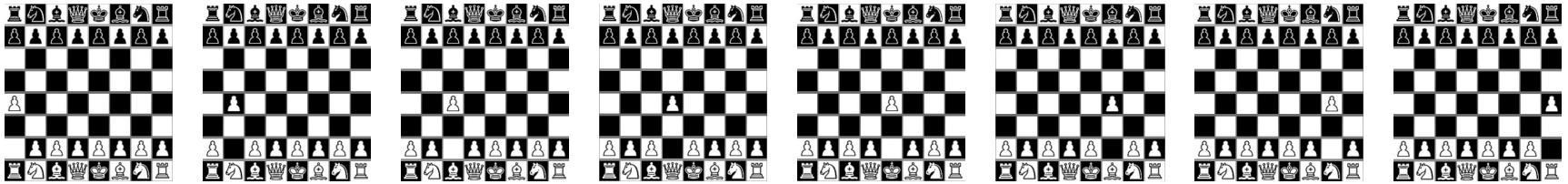
Move

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

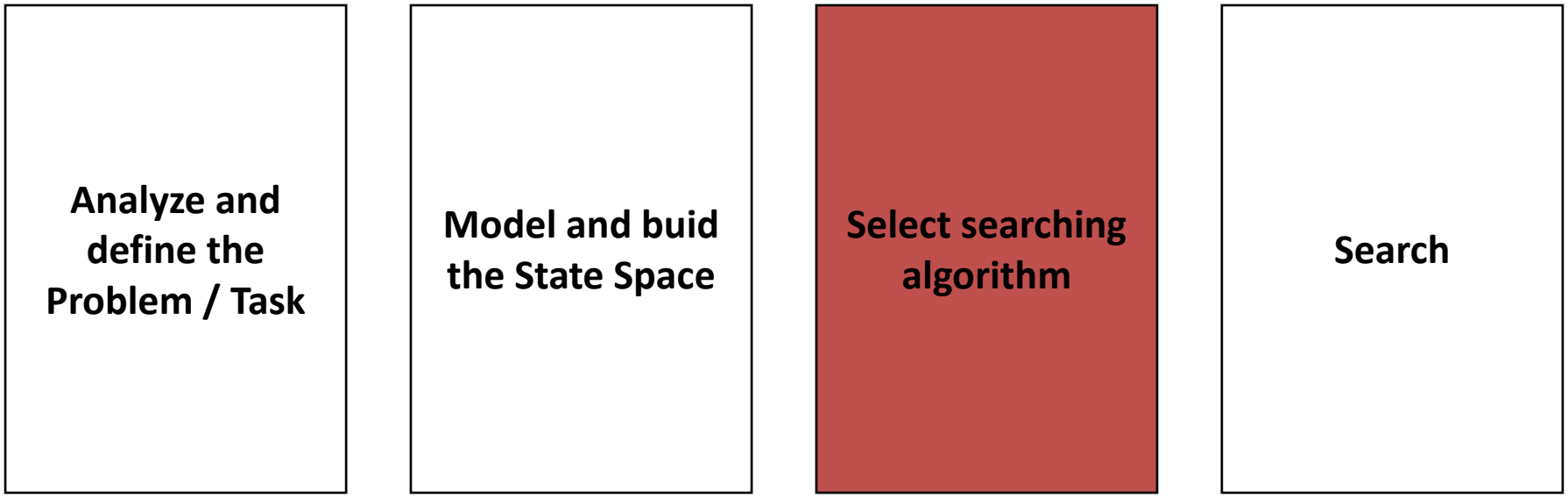


20 Possible legal first moves:
16 pawn moves
4 knight moves





Designing the Searching Problem

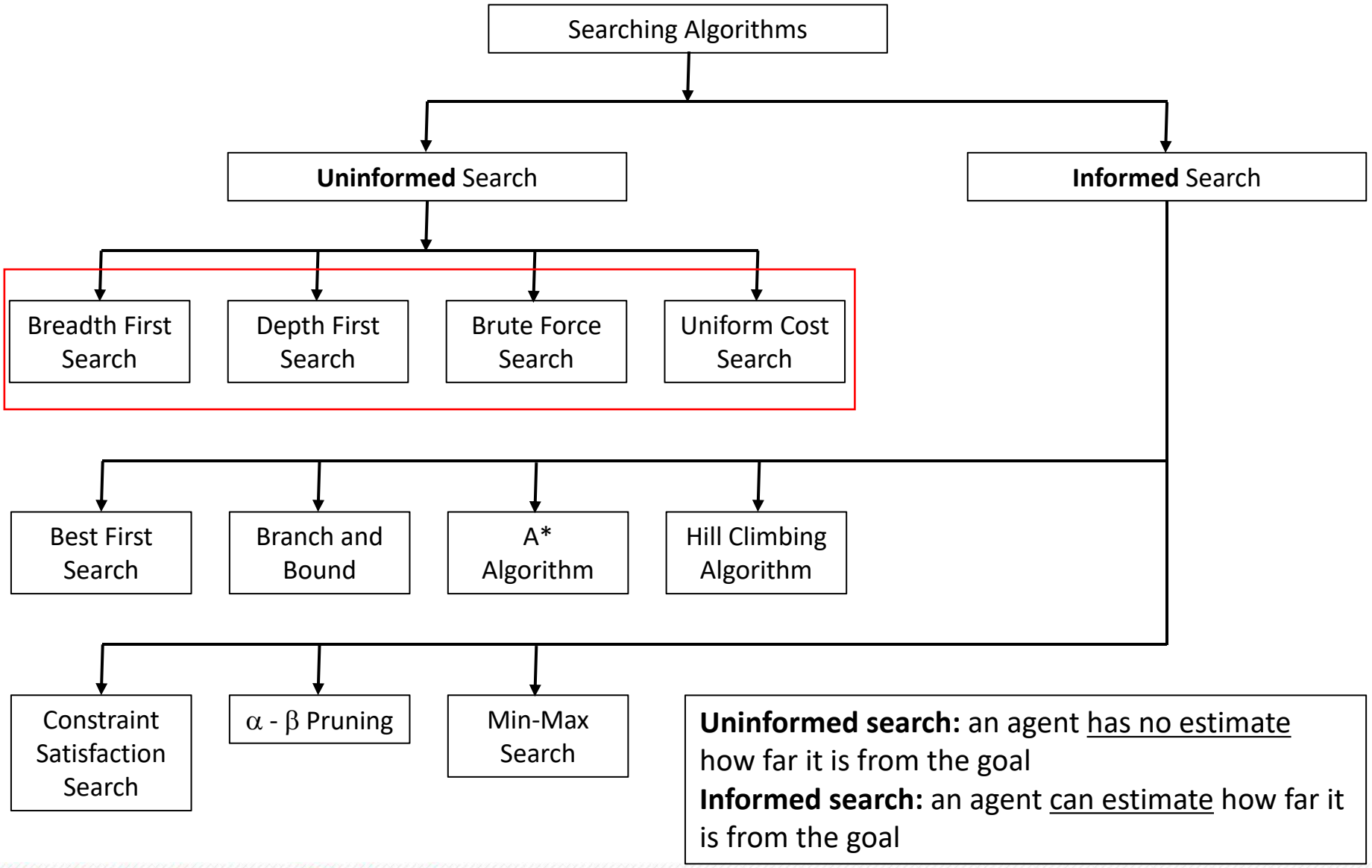


Measuring Searching Performanc

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?

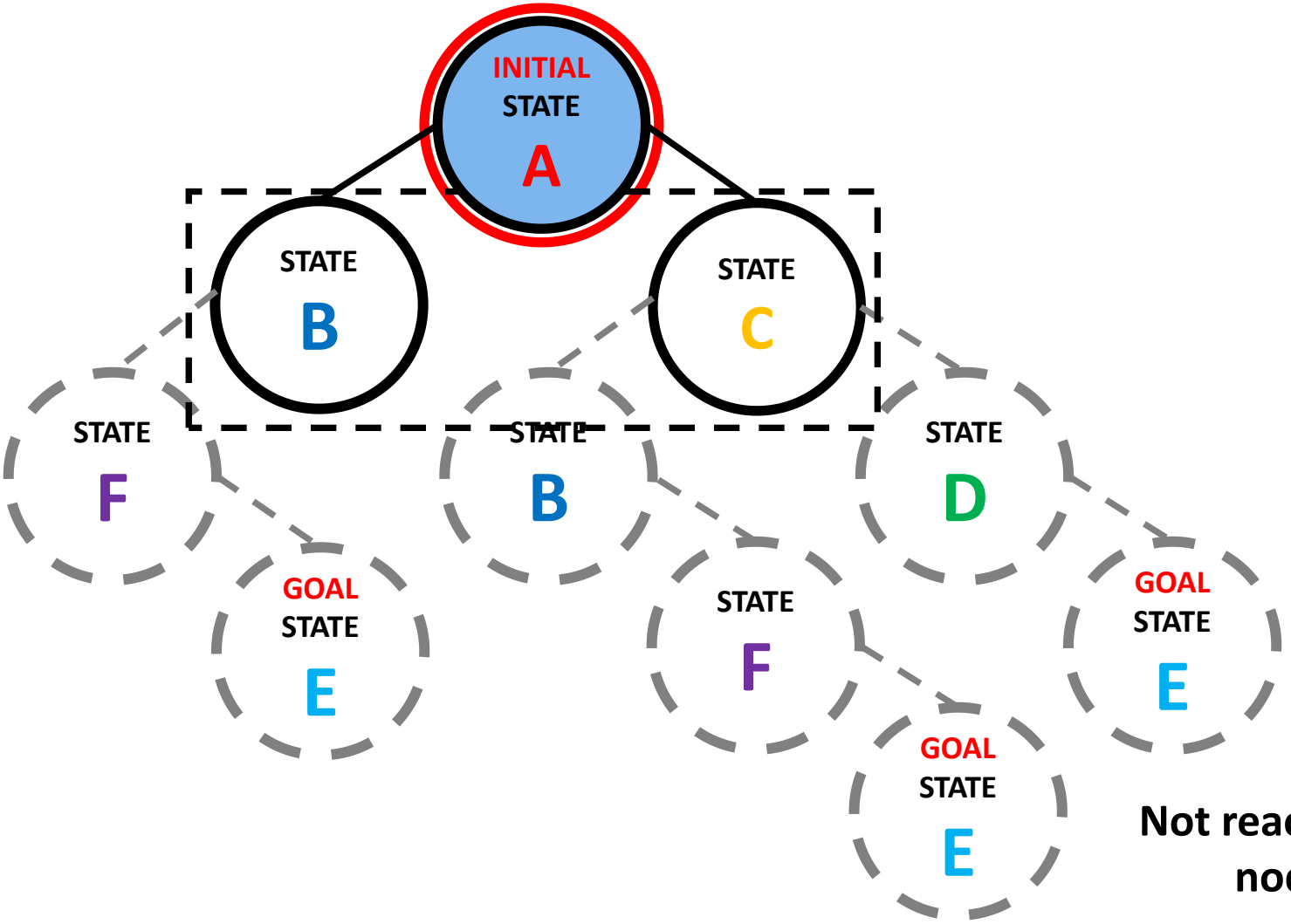
Selected Searching Algorithms



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:

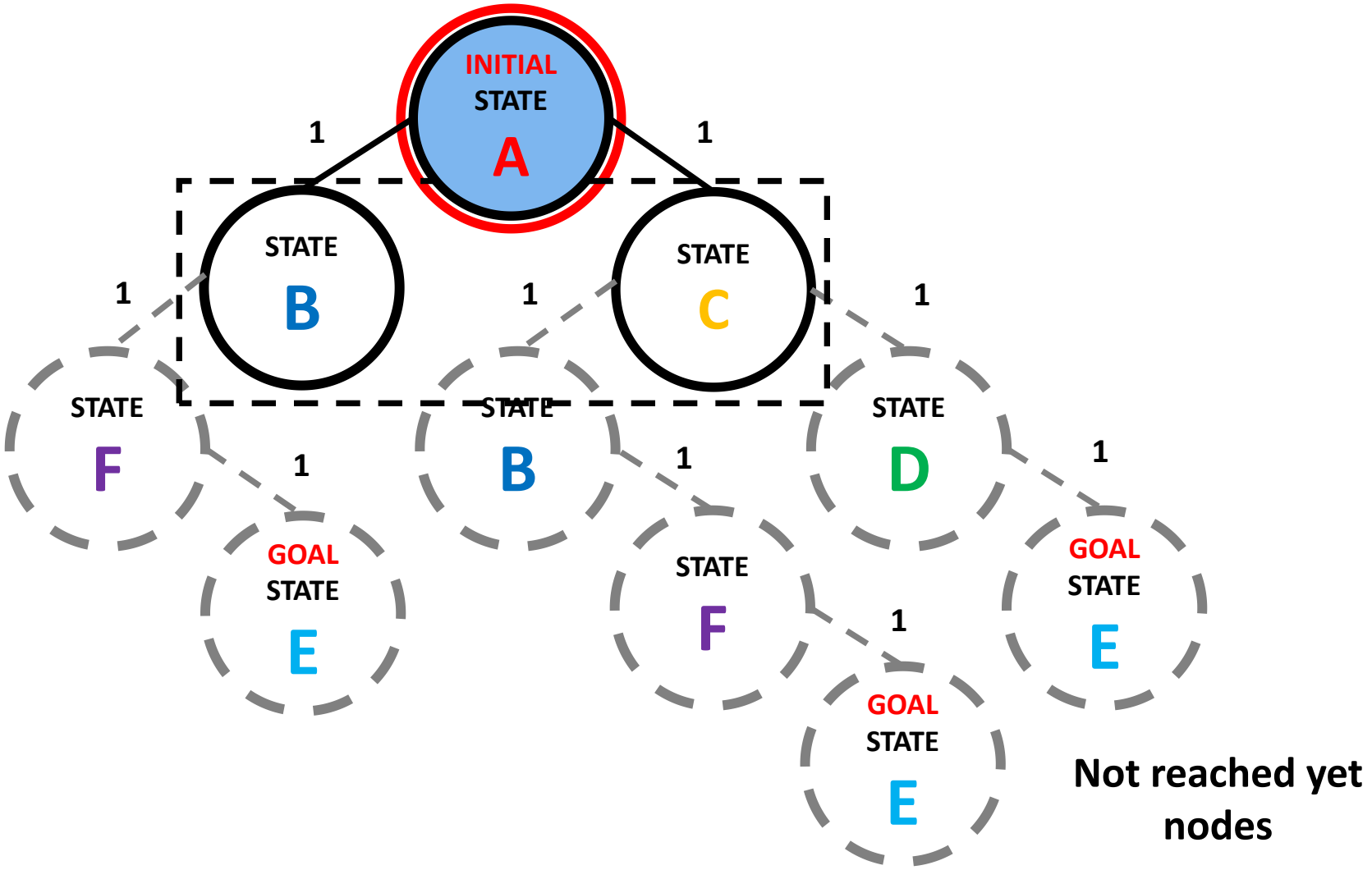
f(n) = f(State n)

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

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

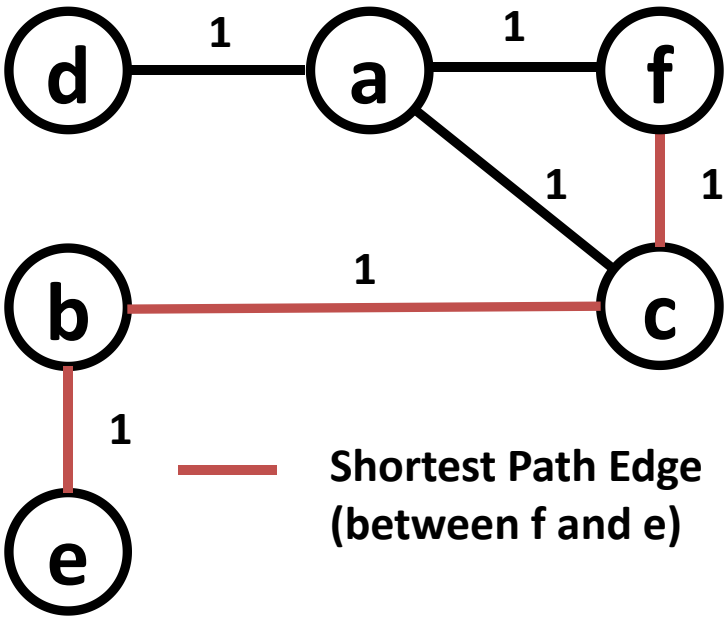
What about ties?

Search Tree: Uniform Action Cost



Uniform Cost Search | Dijkstra's Algo

Weighted Graph G



Popular algorithms:

- **Dijkstra's algorithm**

Shortest Path Problem

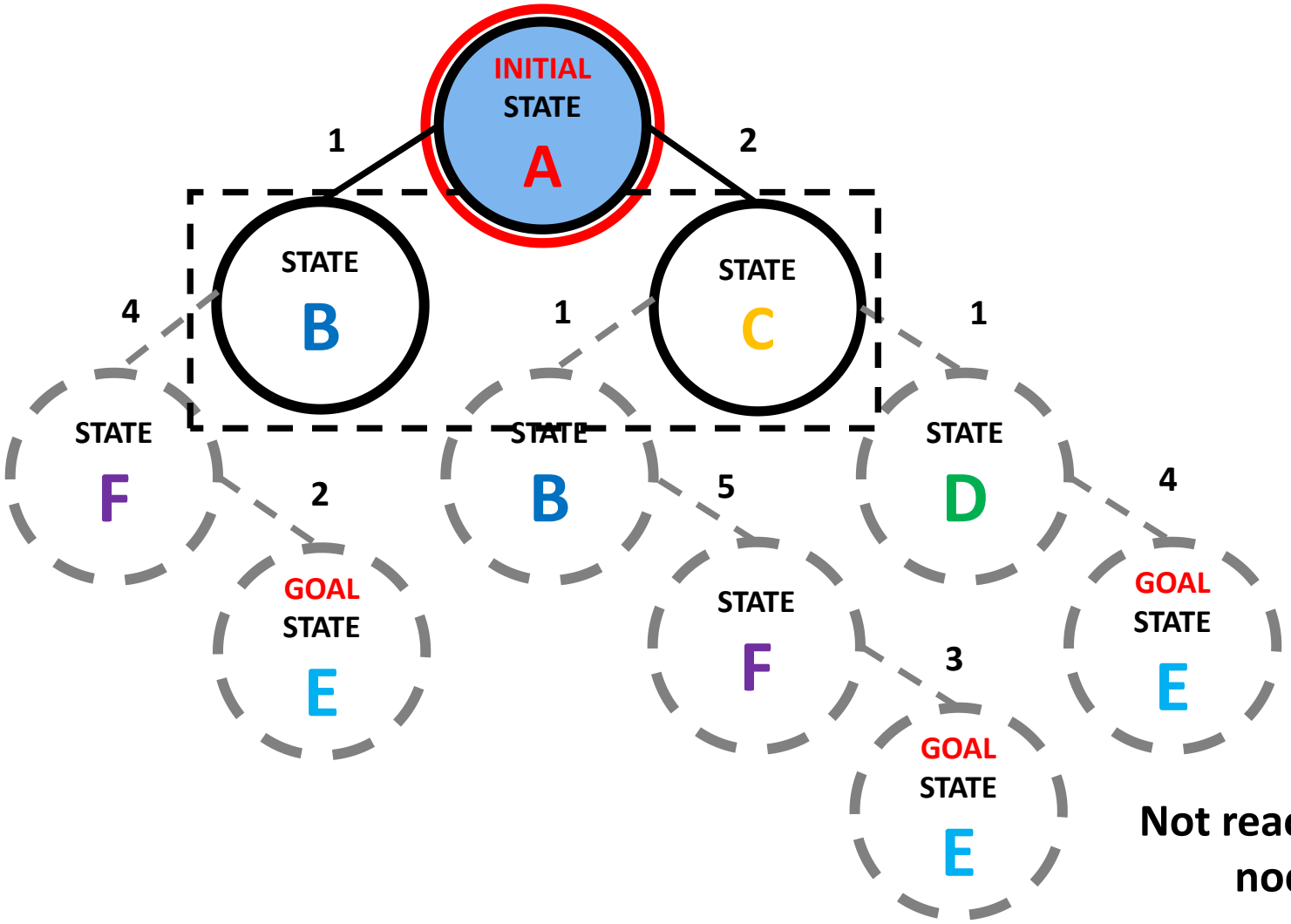
Shortest path problem:
Given a weighted graph $G(V, E, w)$ and two vertices a, b in V, find the shortest path between vertices a and b (**all edge weights are equal**).

BFS and UCS: Pseudocode

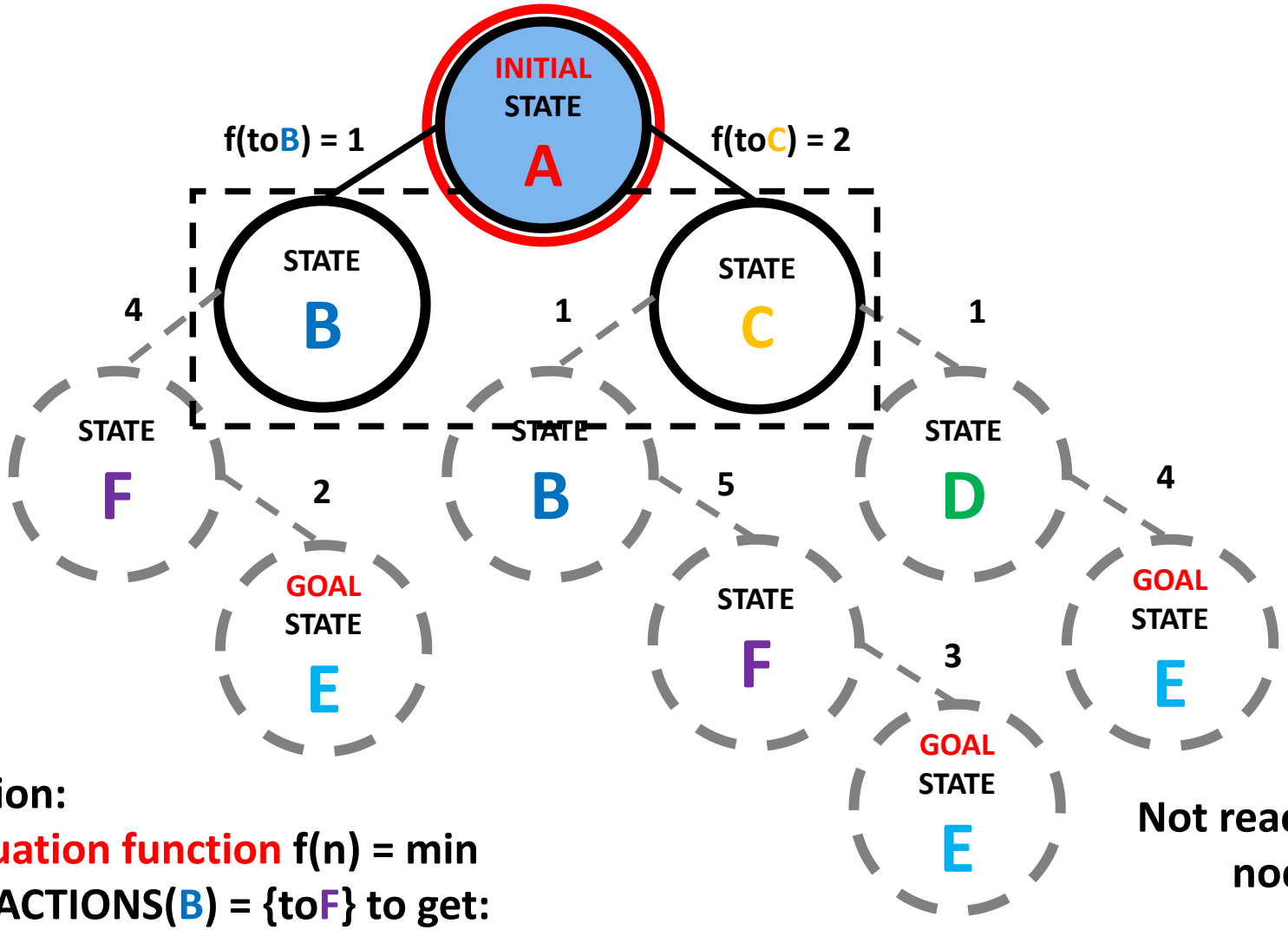
```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
 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



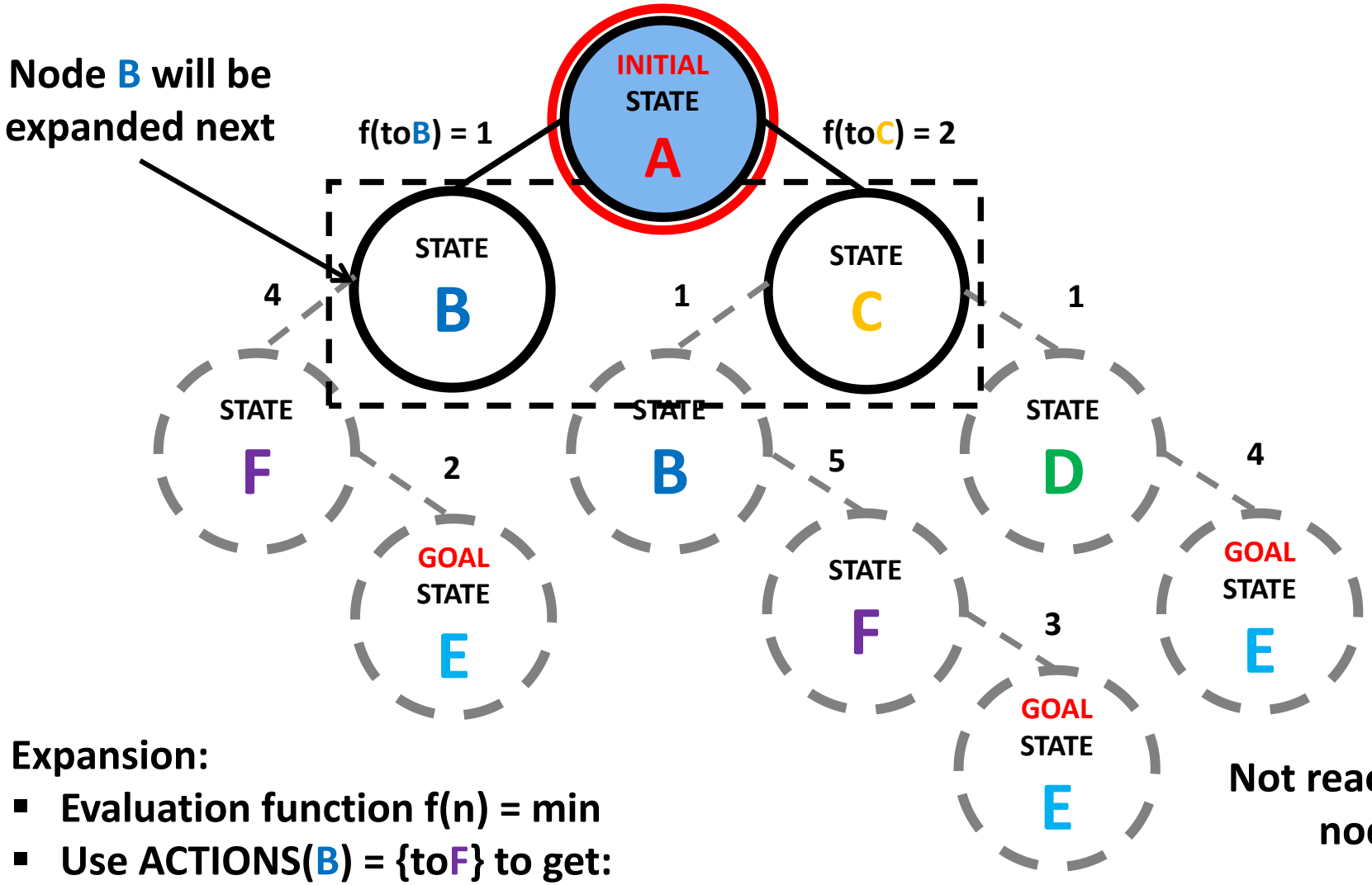
Search Tree: Variable Action Cost



Expansion:

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

Search Tree: Best-First Search



Expansion:

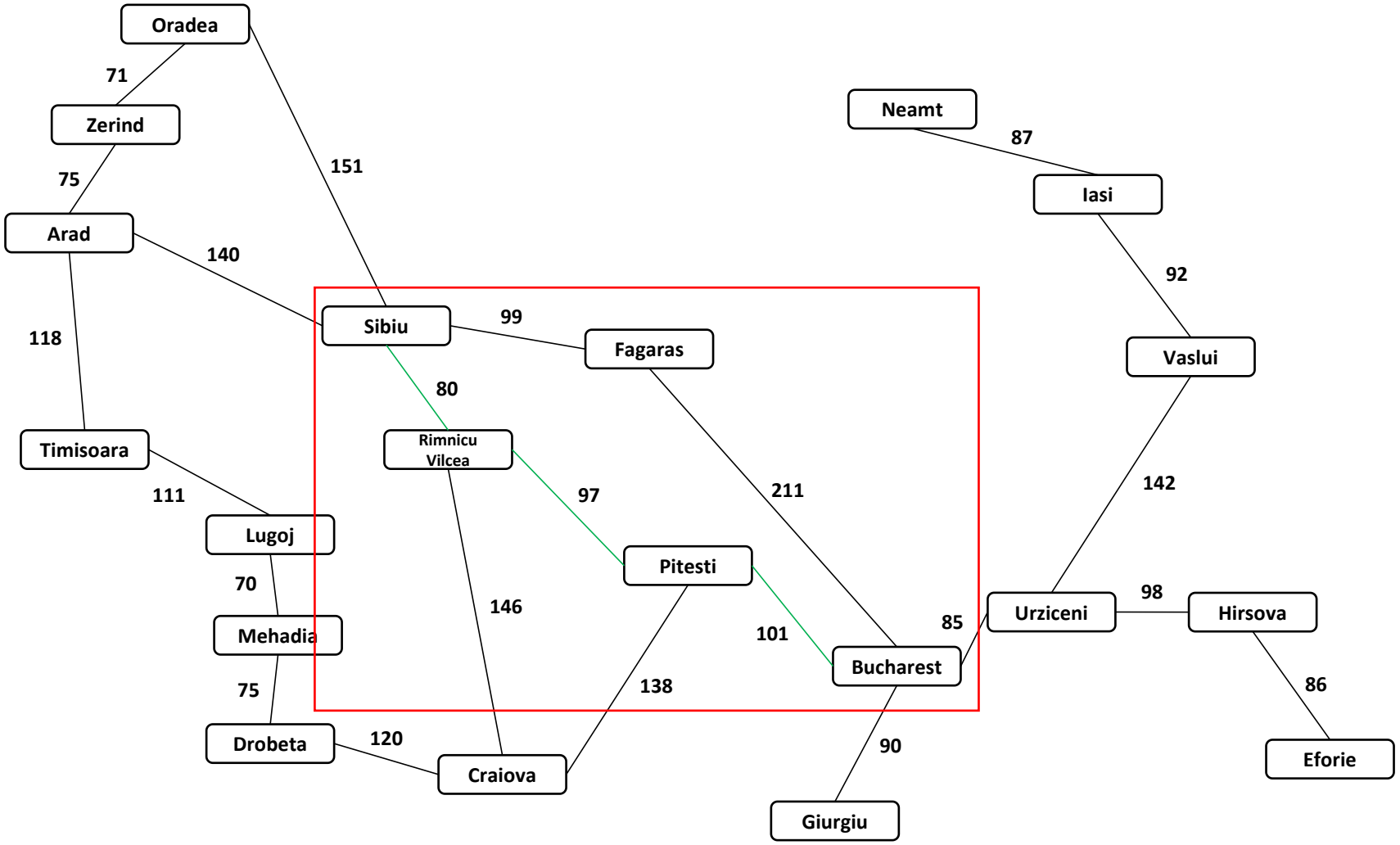
- Evaluation function f(n) = min
- Use ACTIONS(B) = {toF} to get:
- RESULT(B, toF) = F

Best-First Search: Pseudocode

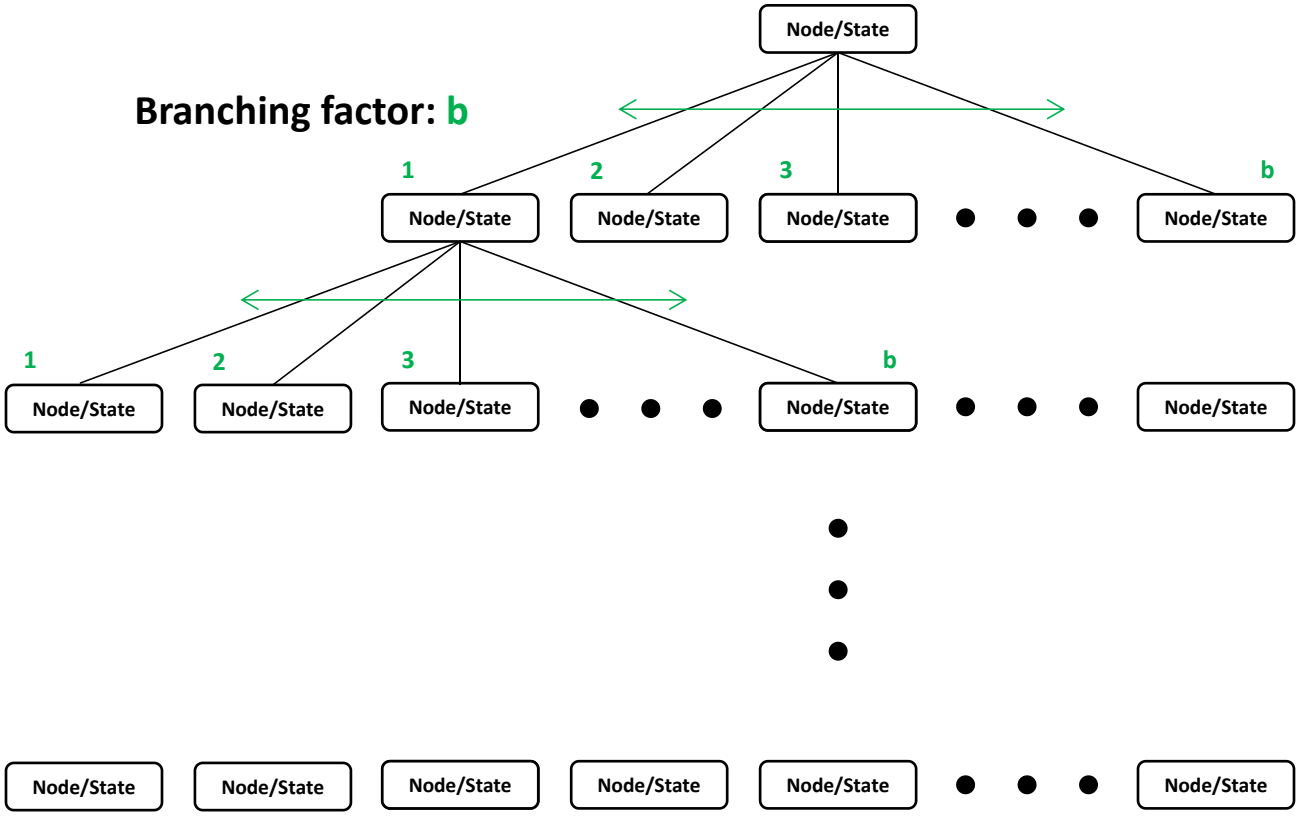
```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
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



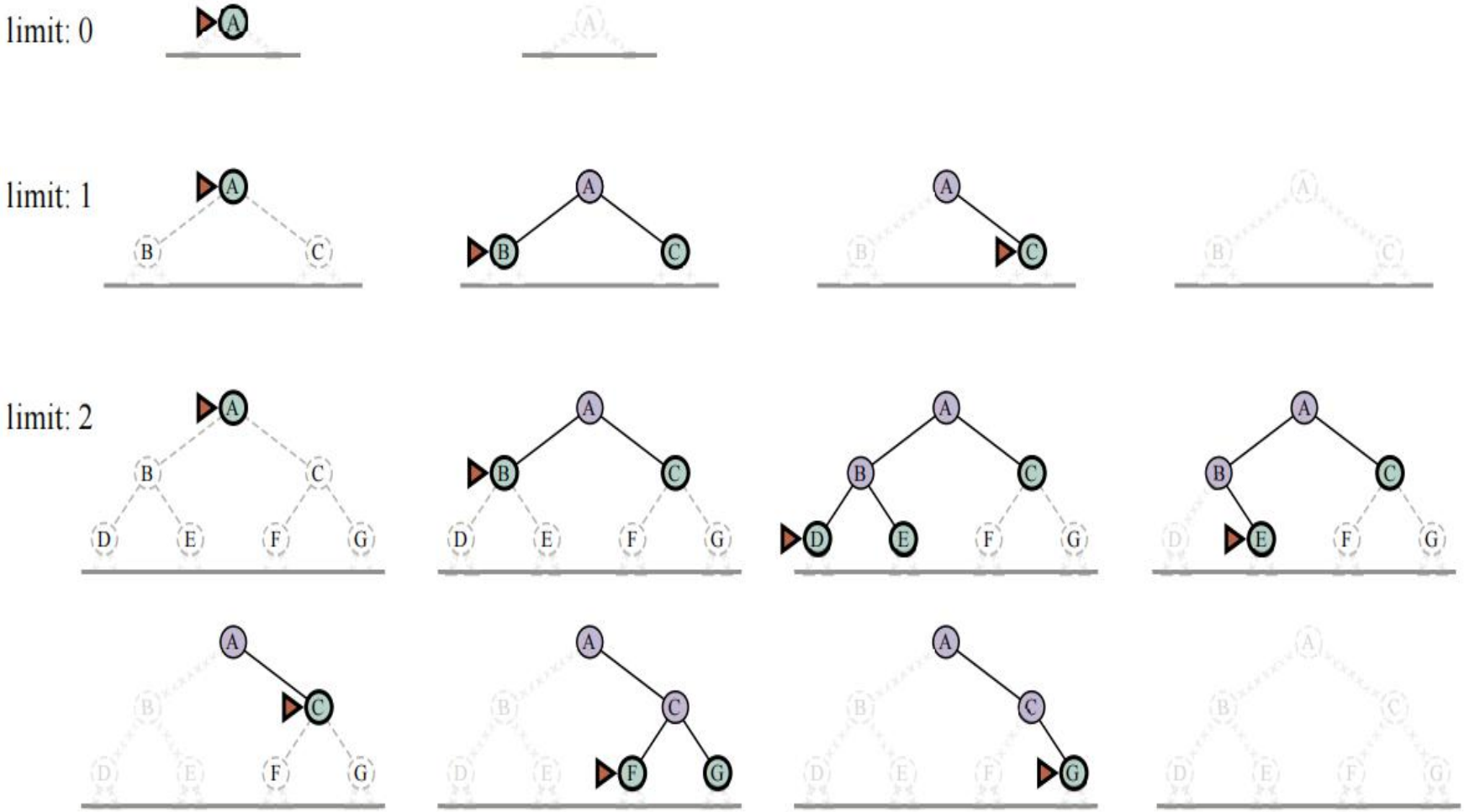
Tree depth is an issue!

"Controlled" DFS: Pseudocode

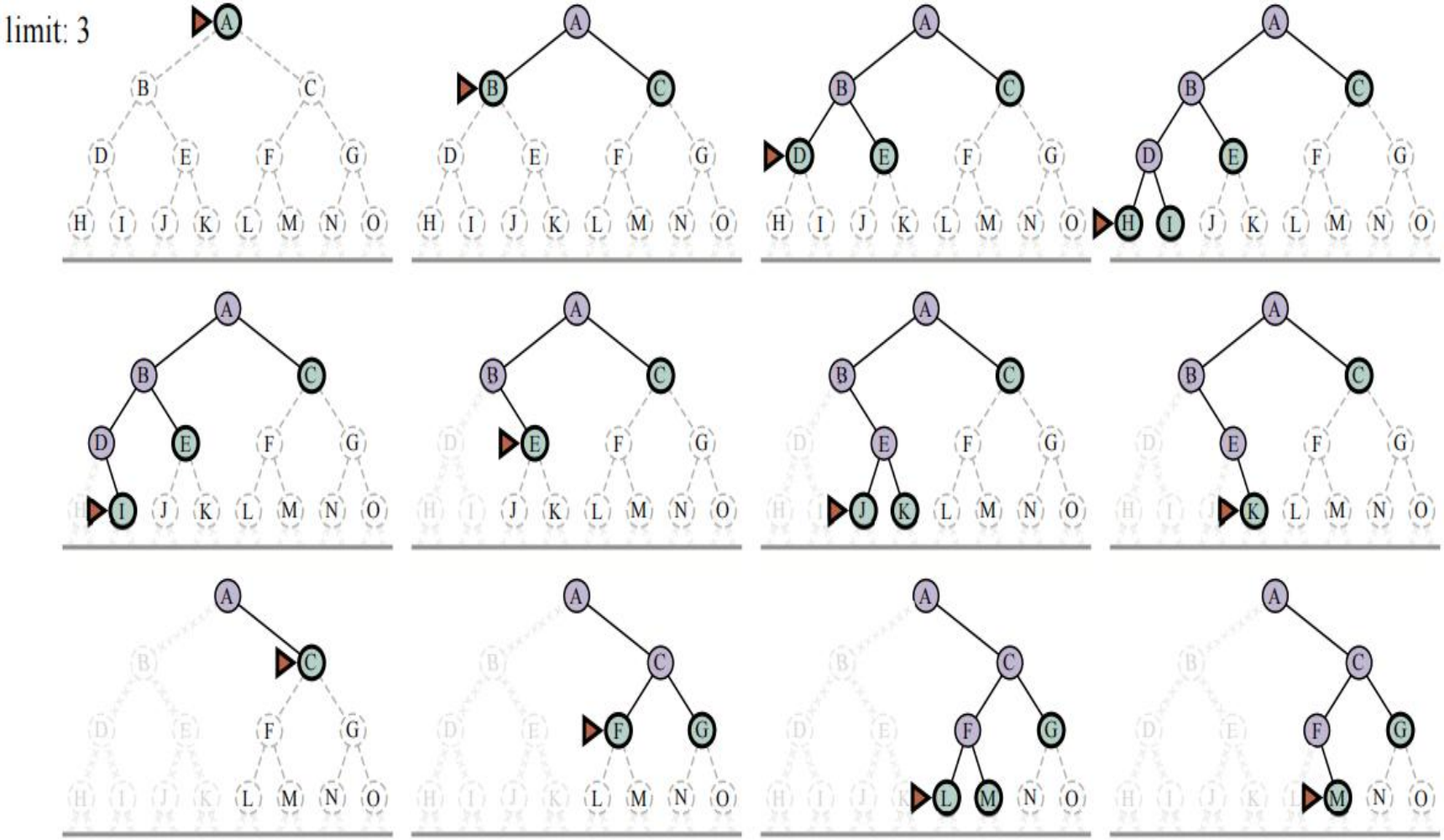
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

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

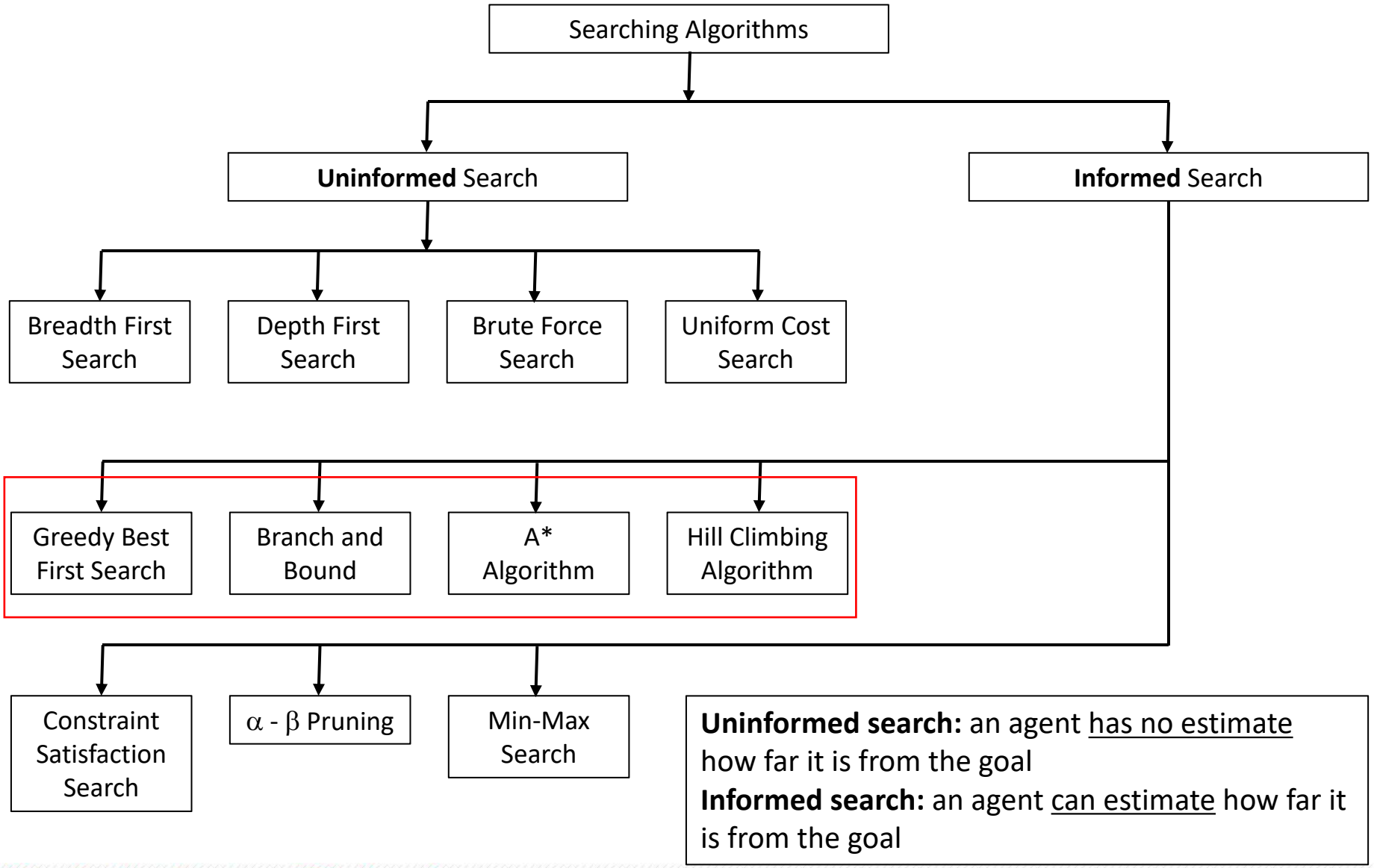


Uninformed Search Algorithms

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.

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

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

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