INTRODUCTION

UNIT - I

INTRODUCTION

- Definition of AI:

  AI is a study which makes the computers to do things which are, at the moment, at the hands of human beings.
- Four different approaches:
  - Thinking Humanly(The Cognitive approach)
  - Acting Humanly(The Turing Test approach)
  - Thinking Rationally(The Laws of Thought approach)
  - Acting Rationally(The Rational Agent approach)

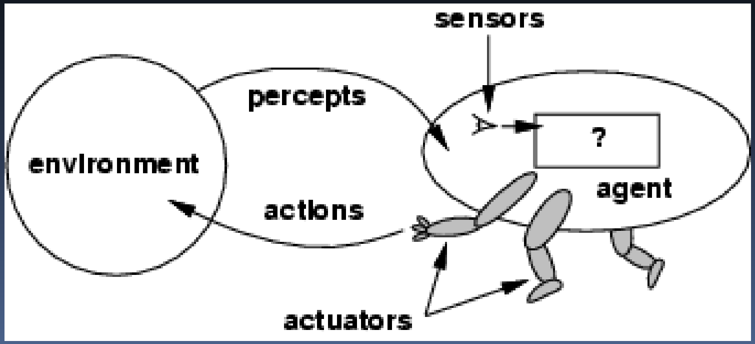
- Foundations of AI:
  - Philosophy (428 BC – Present)
  - Mathematics (c.800 – Present)
  - Economics (1776 – Present)
  - Neuroscience (1861 – Present)
  - Psychology (1879 – Present)
  - Computer engineering (1940 – Present)
  - Control theory and Cybernetics (1948 – Present)
  - Linguistics (1957 – Present)

- History of AI:
  - The gestation of AI (1943 – 1955)
  - The birth of AI (1956)
  - Early enthusiasm, great expectations (1952 – 1969)
  - A dose of reality (1966 – 1973)
  - Knowledge based systems (1969 – 1979)
  - AI becomes an industry (1980 – Present)
  - The return of neural networks (1986 – Present)
  - AI becomes a science (1987 – Present)
  - The emergence of intelligent agents (1995 – Present)

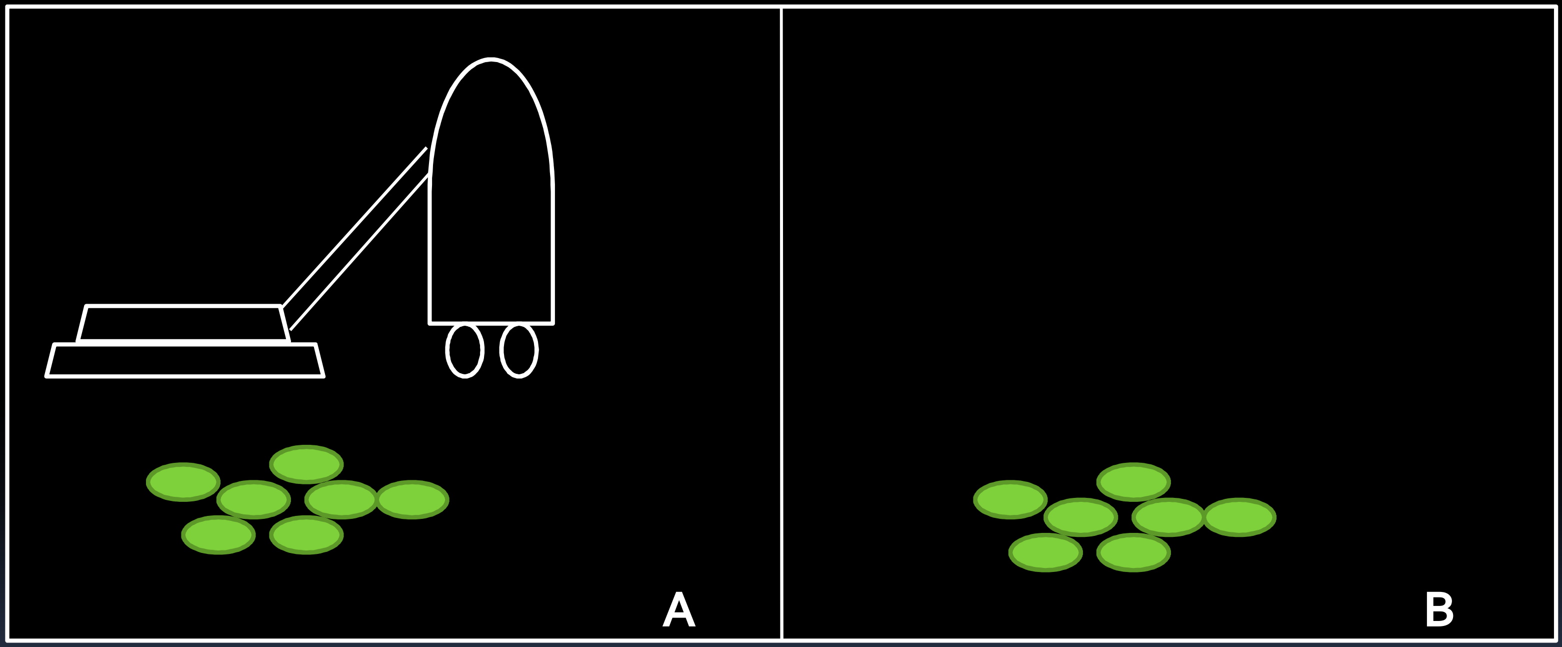
- The state of the art:
  - Autonomous planning
  - Game playing
  - Autonomous control
  - Diagnosis
  - Logistics planning
  - Robotics
  - Language understanding and problem solving

AGENTS

- Agents and Environments:
  - An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**.
  - The term **percept** refers to the agent's perceptual inputs at any given instant.
  - An agent's **percept sequence** is the complete history of everything the agent has ever perceived.
  - *An agent's choice of action at any given instant can depend on the entire percept sequence observed to date.*
  - **Agent function** maps any given percept sequence to an function. It is an abstract mathematical description: the agent program is a concrete implementation, running on the agent architecture.

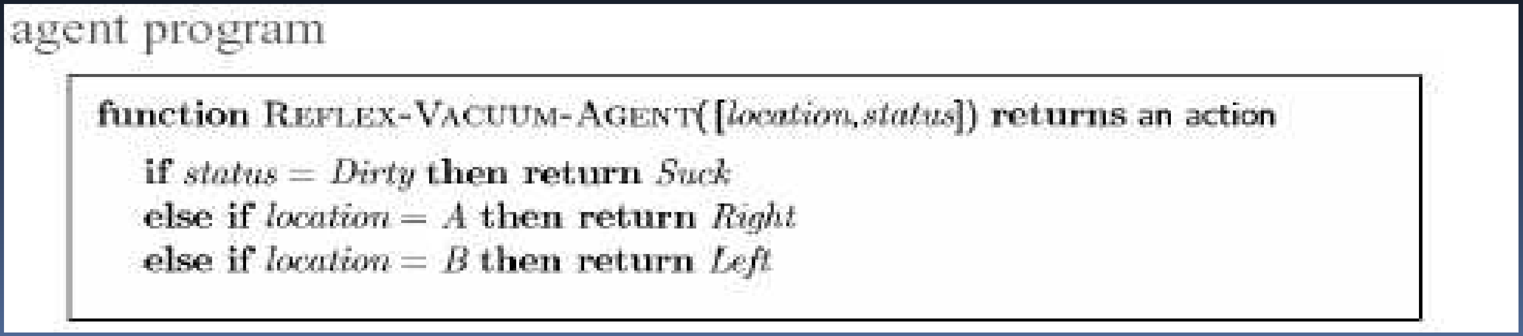


Example: A vacuum cleaner agent with just two locations



Partial tabulation of a simple agent function for the vacuum-cleaner world:

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
:	:
:	:


agent program

```
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
```

- ## Good Behavior:

Rational Agent

A **rational agent is one that does the right thing- conceptually speaking, every entry in** the table for the agent function is filled out correctly. Obviously, doing the right thing is better than doing the wrong thing. The right action is the one that will cause the agent to be most successful.

Performance measures

A performance measure embodies the criterion for success of an agent's behavior. When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives. This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well.

Rationality

What is rational at any given time depends on four things:

- o The performance measure that defines the criterion of success.
- o The agent's prior knowledge of the environment.
- o The actions that the agent can perform.
- o The agent's percept sequence to date.

This leads to a **definition of a 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.

Omniscience, learning, and autonomy

An **omniscient agent knows the *actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.***

Doing actions in order to modify future percepts-sometimes called **information gathering-is an important part of rationality.**

Our definition requires a rational agent not only to gather information, but also to **learn** as much as possible from what it perceives.

To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts, we say that the agent lacks autonomy. A rational agent should be **autonomous-it should learn what it can to compensate for partial or incorrect prior knowledge.**

The Nature of Environments

- **Task environments**

  We must think about **task environments, which are essentially the "problems" to which rational agents are the "solutions."**

- **Specifying the task environment**

  The rationality of the simple vacuum-cleaner agent, needs specification of

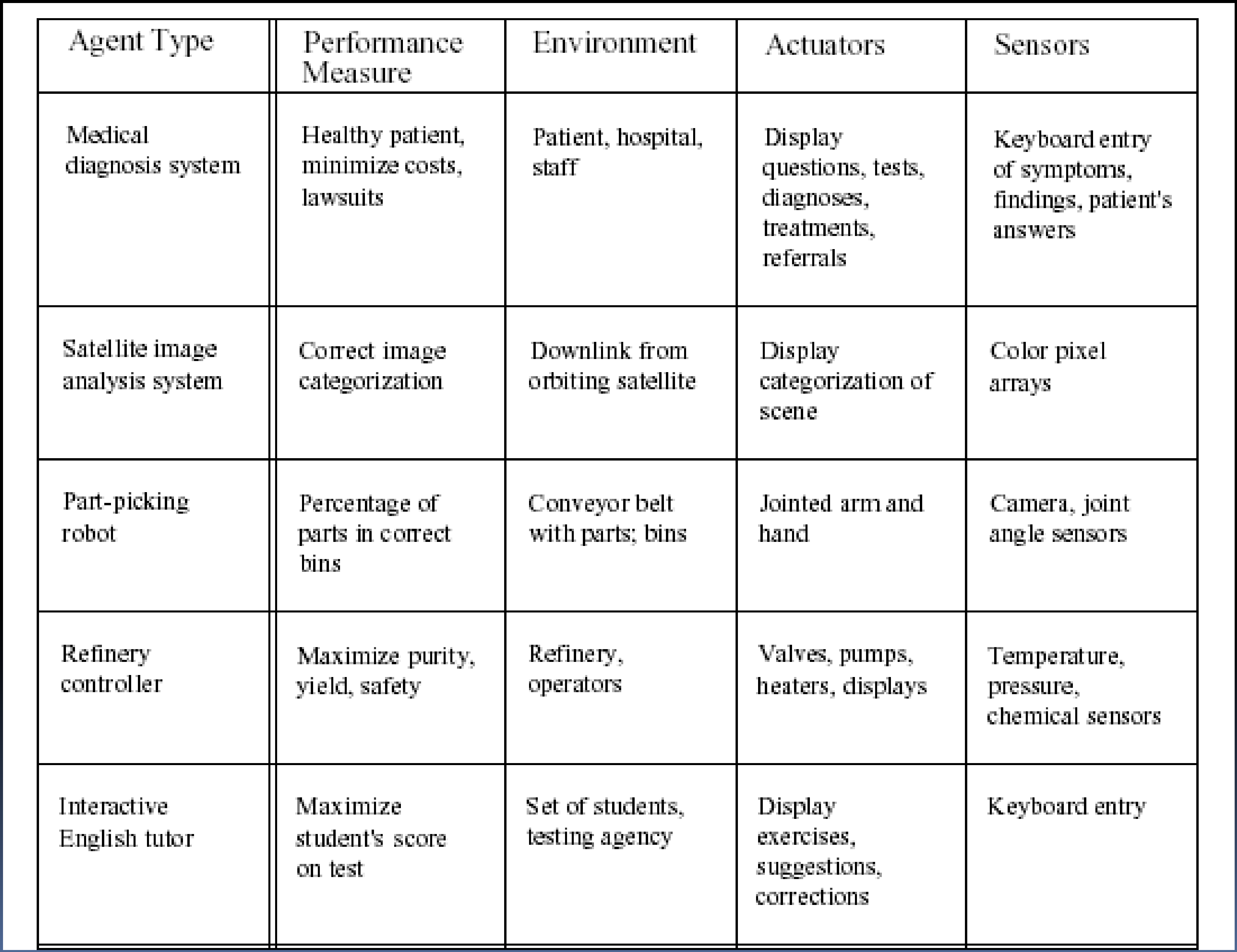
  o the performance measure

  o the environment

  o the agent's actuators and sensors.

- **PEAS**

  All these are grouped together under the heading of the **task environment.** We call this the **PEAS (Performance, Environment, Actuators, Sensors) description.** In designing an agent, the first step must always be to specify the task environment as fully as possible.



Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, minimize costs, lawsuits	Patient, hospital, staff	Display questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display categorization of scene	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Maximize purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Maximize student's score on test	Set of students, testing agency	Display exercises, suggestions, corrections	Keyboard entry

Properties of task environments

o Fully observable vs. partially observable

o Deterministic vs. stochastic

o Episodic vs. sequential

o Static vs. dynamic

o Discrete vs. continuous

o Single agent vs. multiagent

Fully observable vs. partially observable.

If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable. A task environment is effectively fully observable if the sensors detect all aspects that are *relevant* to the choice of action; An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data.

Deterministic vs. stochastic.

If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; otherwise, it is stochastic.

Episodic vs. sequential

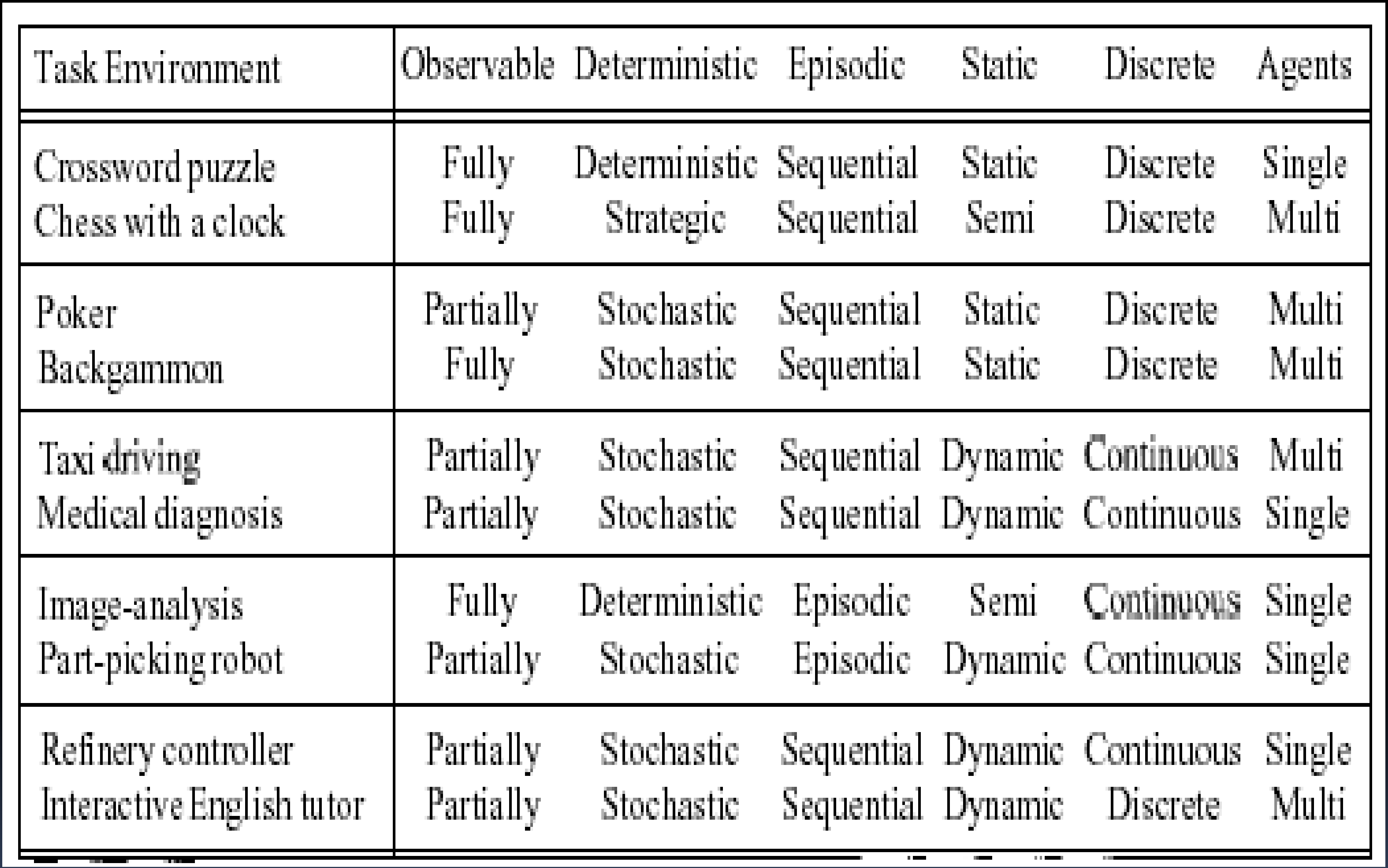
In an **episodic task environment, the agent's experience is divided into atomic episodes.** Each episode consists of the agent perceiving and then performing a single action. Crucially, the next episode does not depend on the actions taken in previous episodes. For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions; In **sequential environments, on the other hand, the current decision** could affect all future decisions. Chess and taxi driving are sequential:

Discrete vs. continuous.

The discrete/continuous distinction can be applied to the *state of the environment, to* the way *time is handled, and to the percepts and actions of the agent. For example, a* discrete-state environment such as a chess game has a finite number of distinct states. Chess also has a discrete set of percepts and actions. Taxi driving is a continuous- state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time. Taxi-driving actions are also continuous (steering angles, etc.).

Single agent vs. multiagent.

An agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two-agent environment. As one might expect, the hardest case is *partially observable, stochastic, sequential, dynamic, continuous, and multiagent.*



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

- The Structure of Agents:

  Agent = Architecture + Program

  - Table Driven Agent
  - Simple Reflex Agents
  - Model Based Reflex Agents
  - Goal Based Reflex Agents
  - Utility Based Reflex Agents

Table-Driven Agent

```
Function TABLE-DRIVEN_AGENT(percept) returns an action
static: percepts, a sequence initially empty
        table, a table of actions, indexed by percept sequence
            append percept to the end of percepts
            action LOOKUP(percepts, table)
return action
```

Drawbacks:

• **Table lookup** of percept-action pairs defining all possible condition-action rules necessary to interact in an environment

• **Problems**

– Too big to generate and to store (Chess has about 10^120 states, for example)

– No knowledge of non-perceptual parts of the current state

– Not adaptive to changes in the environment; requires entire table to be updated if changes occur

– Looping: Can't make actions conditional

• Take a long time to build the table

• No autonomy

• Even with learning, need a long time to learn the table entries

Some Agent Types

• **Table-driven agents**

– use a percept sequence/action table in memory to find the next action. They are implemented by a (large) **lookup table.**

• **Simple reflex agents**

– are based on **condition-action rules, implemented with an appropriate production system. They are stateless devices which do not have memory of past world states.**

• **Agents with memory**

– have **internal state, which is used to keep track of past states of the world.**

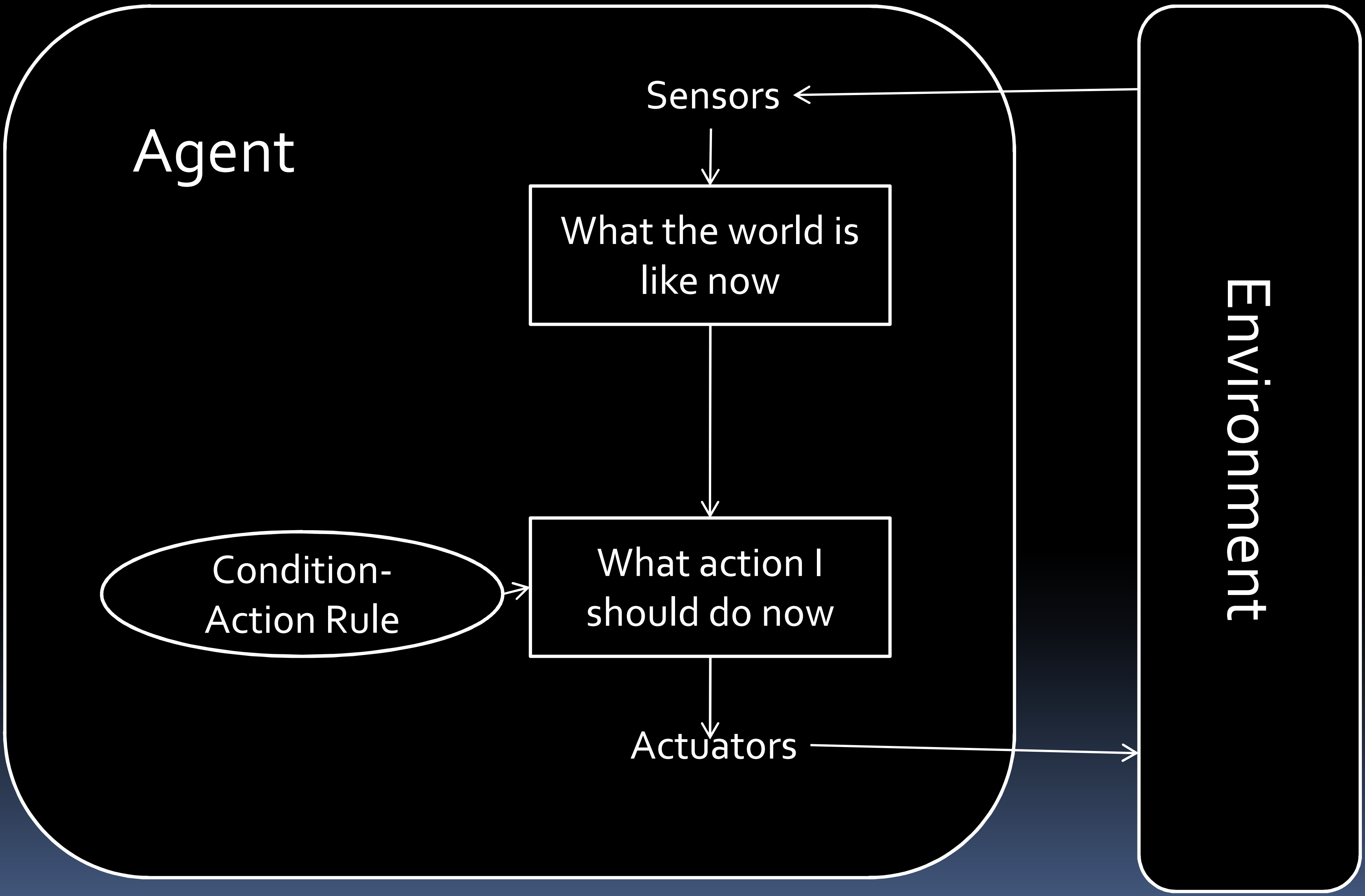
• **Agents with goals**

– are agents that, in addition to state information, have **goal information that describes desirable situations. Agents of this kind take future events into consideration.**

• **Utility-based agents**

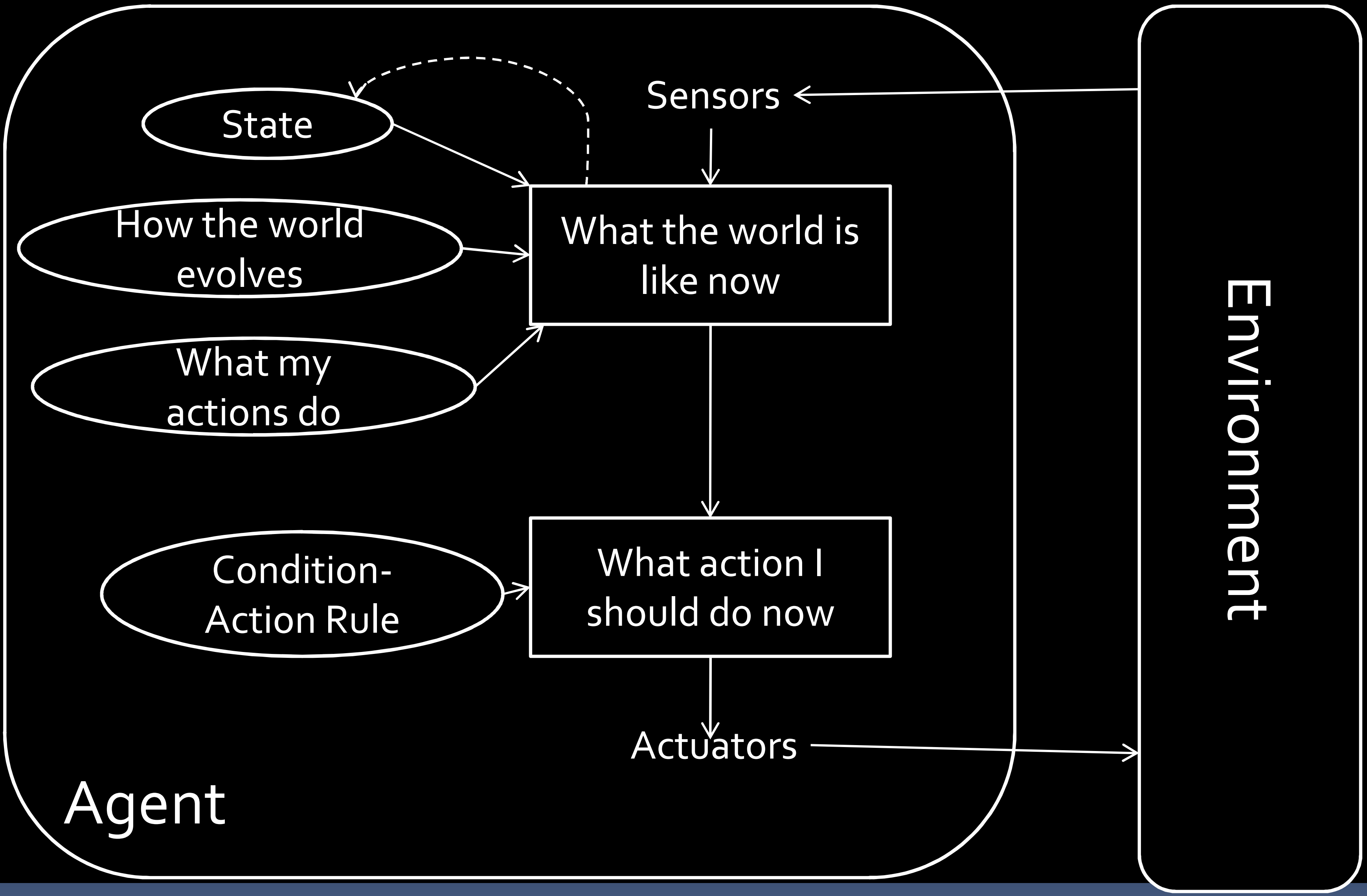
– base their decisions on **classic axiomatic utility theory in order to act rationally.**

SIMPLE-REFLEX AGENT:



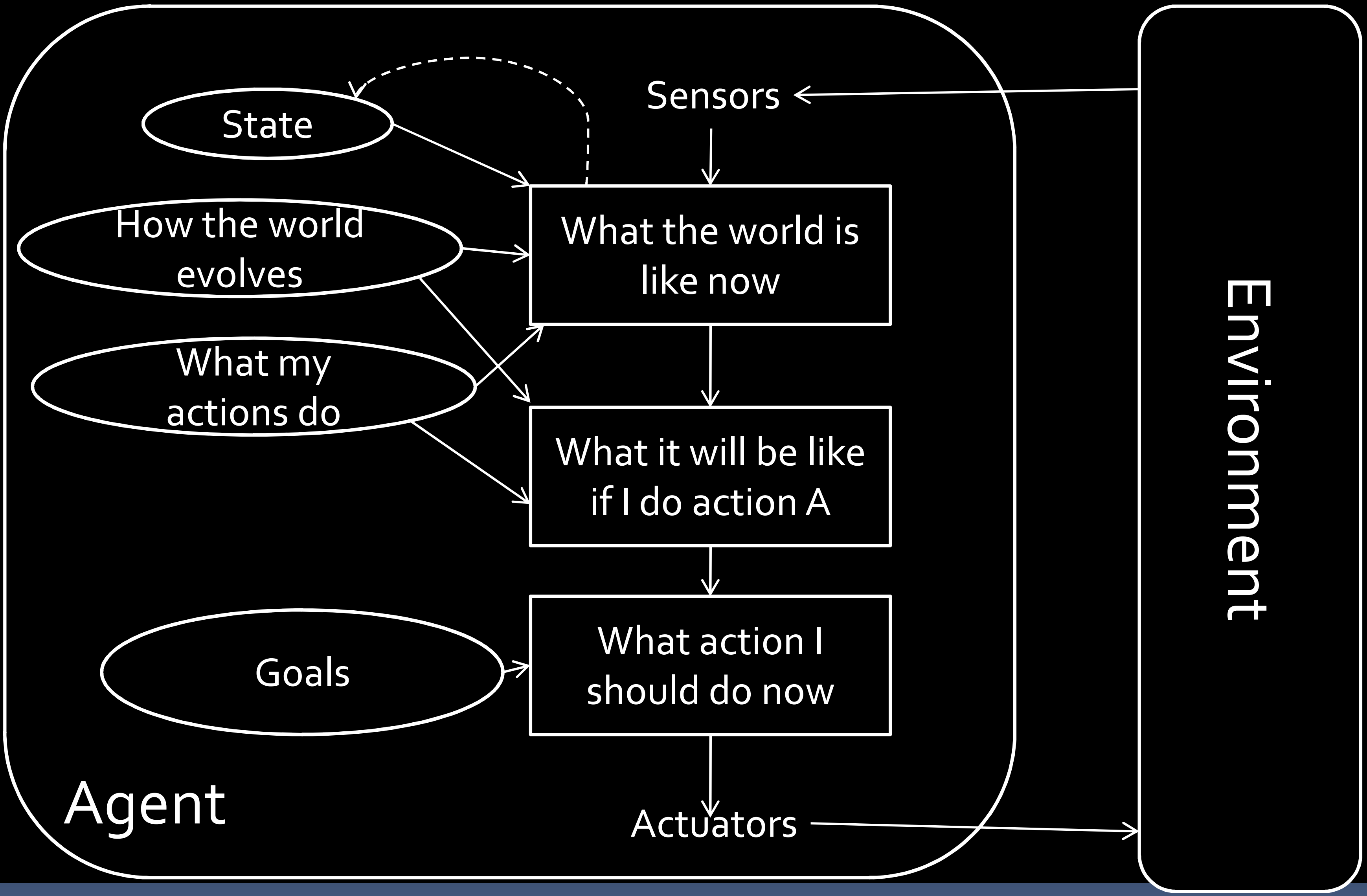
```
function SIMPLE-REFLEX-AGENT(percept) returns an action
static: rules, a set of condition-action rules
    state ← INTERPRET-INPUT(percept)
    rule ← RULE-MATCH(state, rule)
    action ← RULE-ACTION[rule]
return action

function REFLEX-VACUUM-AGENT ([location, status]) return
    an action
    if status == Dirty then return Suck
    else if location == A then return Right
    else if location == B then return Left
```

MODEL-BASED REFLEX AGENT:



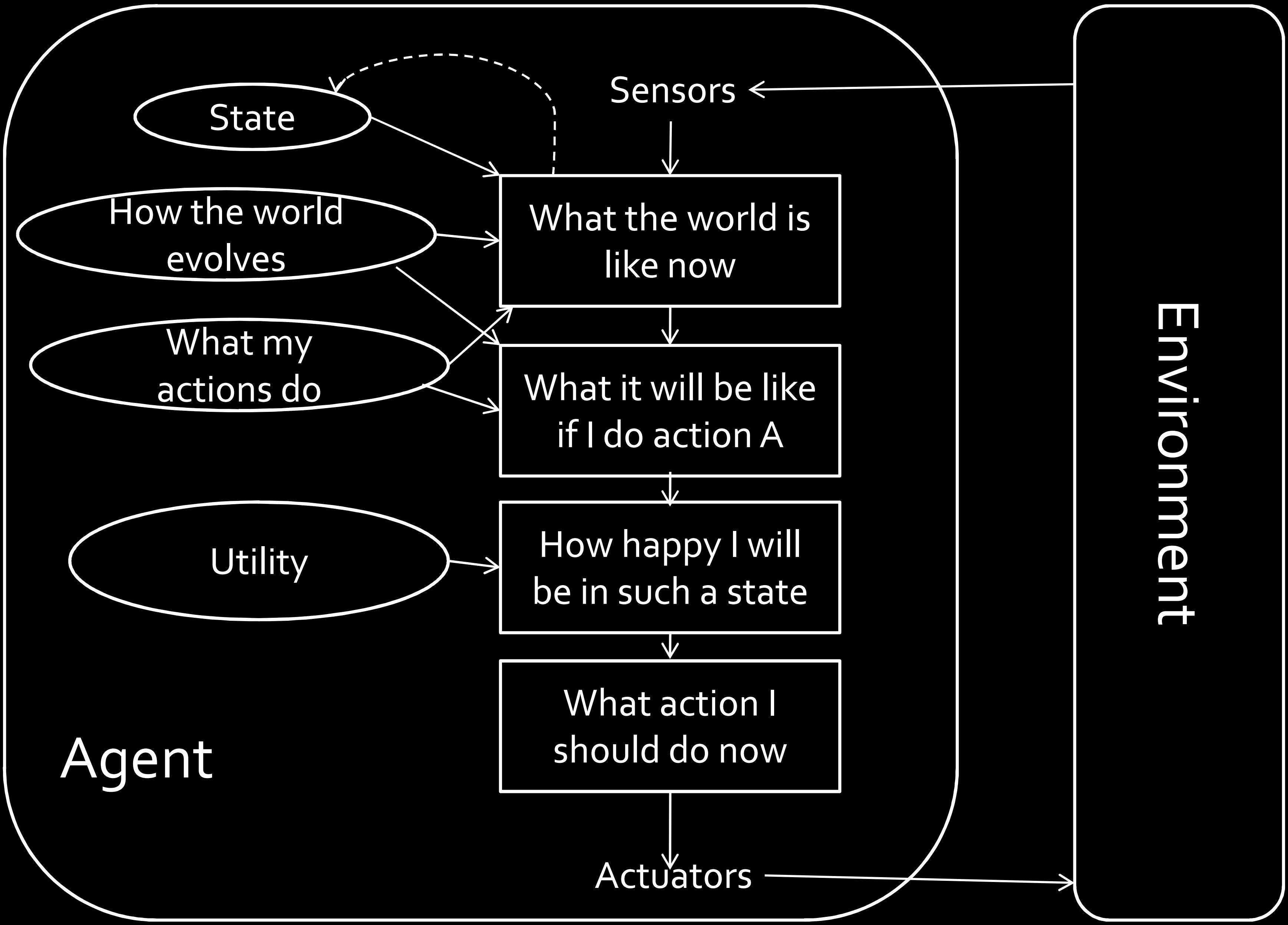
```
function REFLEX-AGENT-WITH-STATE(percept)
    returns an action
static: rules, a set of condition-action rules
        state, a description of the current world state
        action, the most recent action.
    state ← UPDATE-STATE(state, action, percept)
    rule ← RULE-MATCH(state, rule)
    action ← RULE-ACTION[rule]
return action
```

GOAL-BASED AGENT:

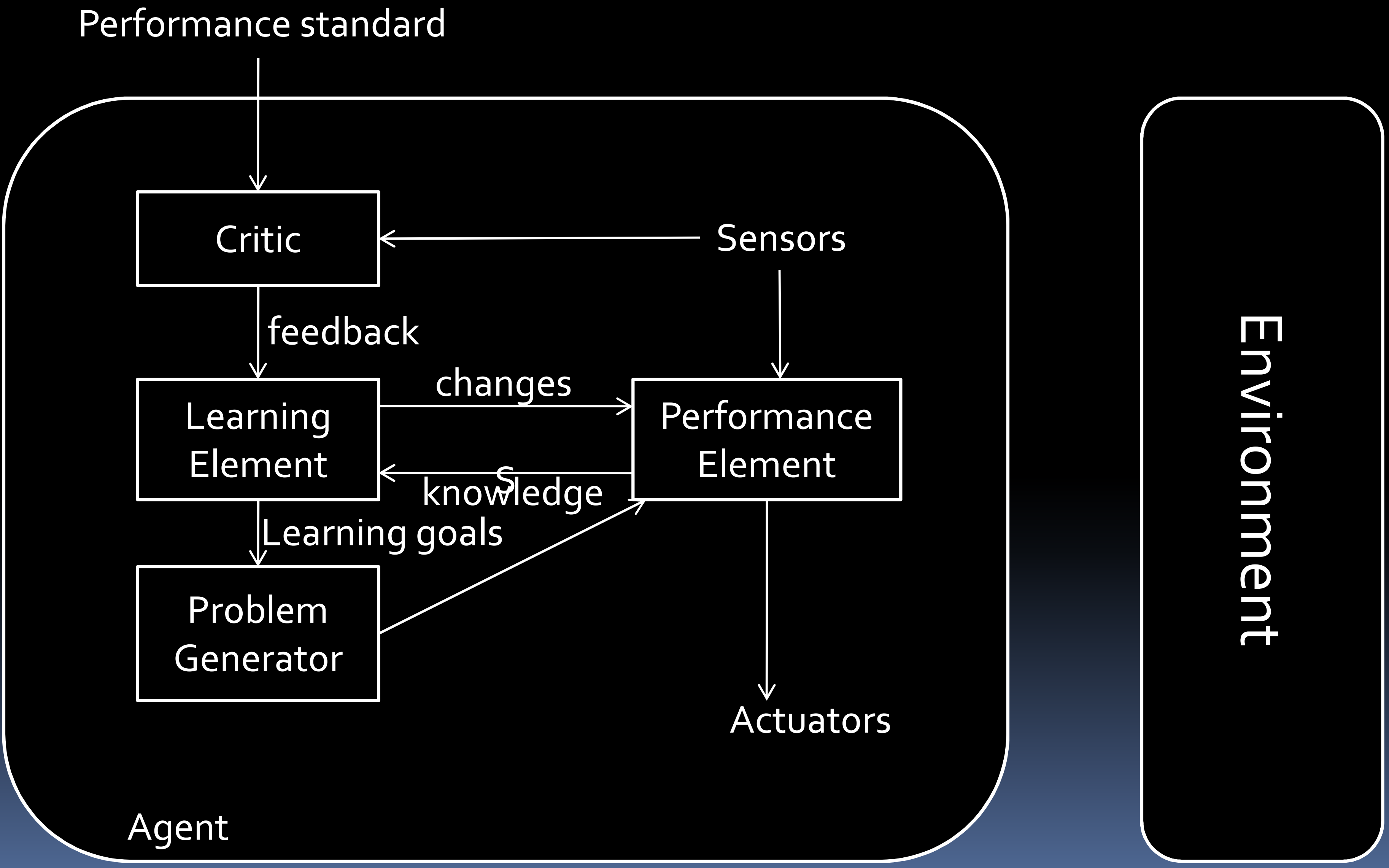


Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

UTILITY-BASED AGENT:


State
Sensors
How the world evolves
What the world is like now
What my actions do
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

LEARNING AGENT



PROBLEM FORMULATION

- **Problem solving agent** is one kind of goal-based agent.
- A well defined problem can be classified by:
  - Initial state
  - Operator or Successor function
  - State space
  - Path
  - Path cost
  - Goal test

- **What is search?**

  Search is the systematic examination of **states** to find path from the **start/root state** to the **goal state.**

  The set of possible states, together with *operators* defining their connectivity constitute the *search space.*

  The output of a search algorithm is a solution, ie., a path from the initial state to a state that satisfies the goal test.
- **Goal formulation** is based on current situation and the agent's performance measures, and it is the first step in problem solving.
- **Problem formulation** is a process of deciding what actions and states to consider, given a goal.

- Search

  An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that leads to the states of known value, and then choosing the best sequence. The process of looking for sequences actions from the current state to reach the goal state is called **search.**

  The **search algorithm** takes a problem as input and returns a solution in the form of action sequence. Once a solution is found, the **execution phase** consists of carrying out the recommended action.

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
    inputs : percept, a percept
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation
    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH( problem)
    action ← FIRST(seq);
    seq ← REST(seq)
return action
```

- Well – defined problems and solutions:

  A **problem can be formally defined by four components:**
  - The **initial state** that the agent starts in . The initial state for our agent of example problem is described by *In(Arad)*
  - A **Successor Function** returns the possible actions available to the agent. Given a state x,SUCCESSOR-FN(x) returns a set of {action, successor} ordered pairs where each action is one of the legal actions in state x,and each successor is a state that can be reached from x by applying the action.

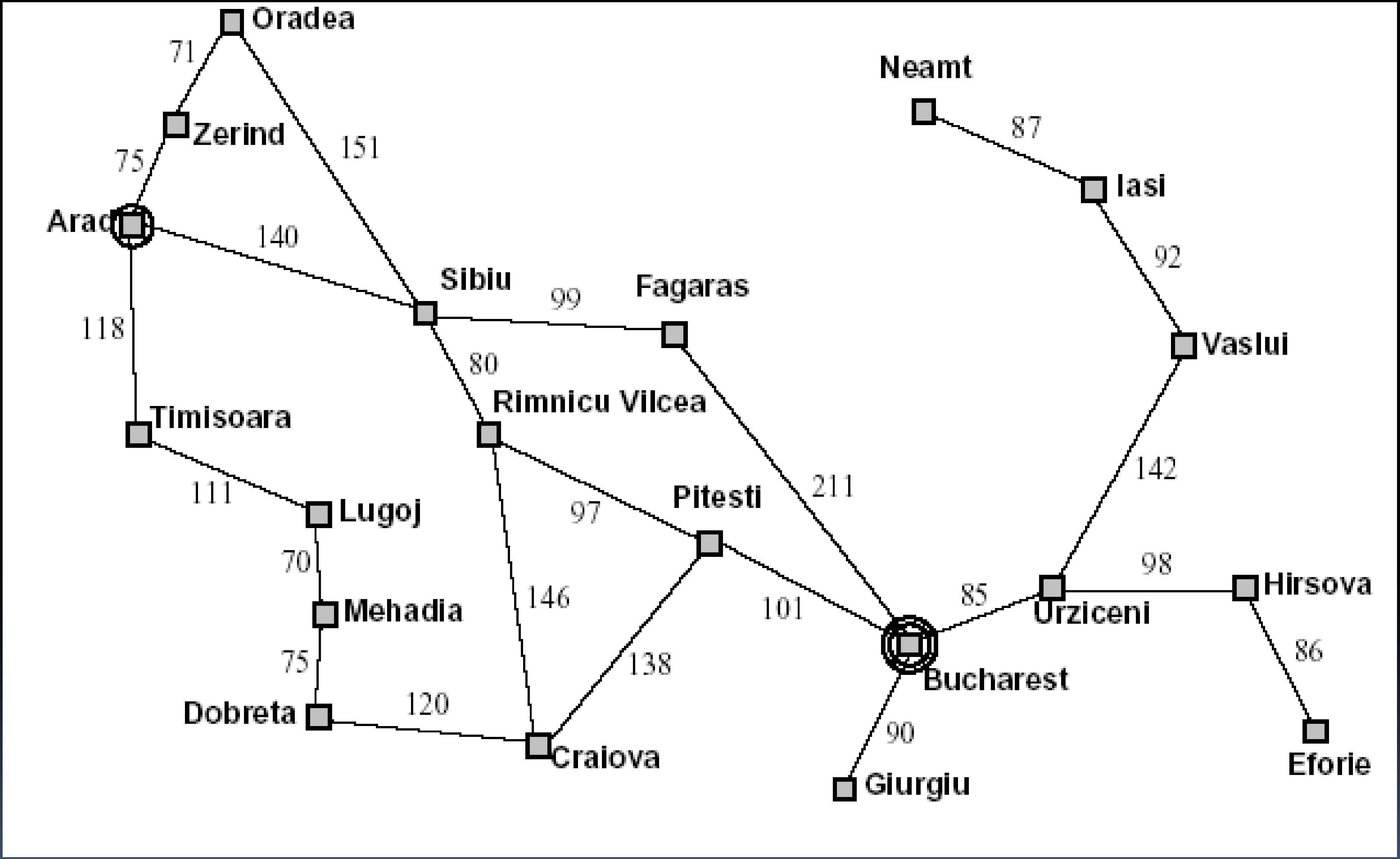
    For example, from the state In(Arad),the successor function for the Romania problem would return {
    [Go(Sibiu),In(Sibiu)],[Go(Timisoara),In(Timisoara)],[Go(Zerind),In(Zerind)] }

    State Space : The set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions.

    A **path** in the state space is a sequence of states connected by a sequence of actions.
  - The **goal test** determines whether the given state is a goal state.
  - A **path cost function** assigns numeric cost to each action. For the Romania problem the cost of path might be its length in kilometers.

    The **step cost** of taking action a to go from state x to state y is denoted by c(x,a,y).
- A **solution to the problem is a path from the initial state to a goal state.**
- An **optimal solution has the lowest path cost among all solutions.**

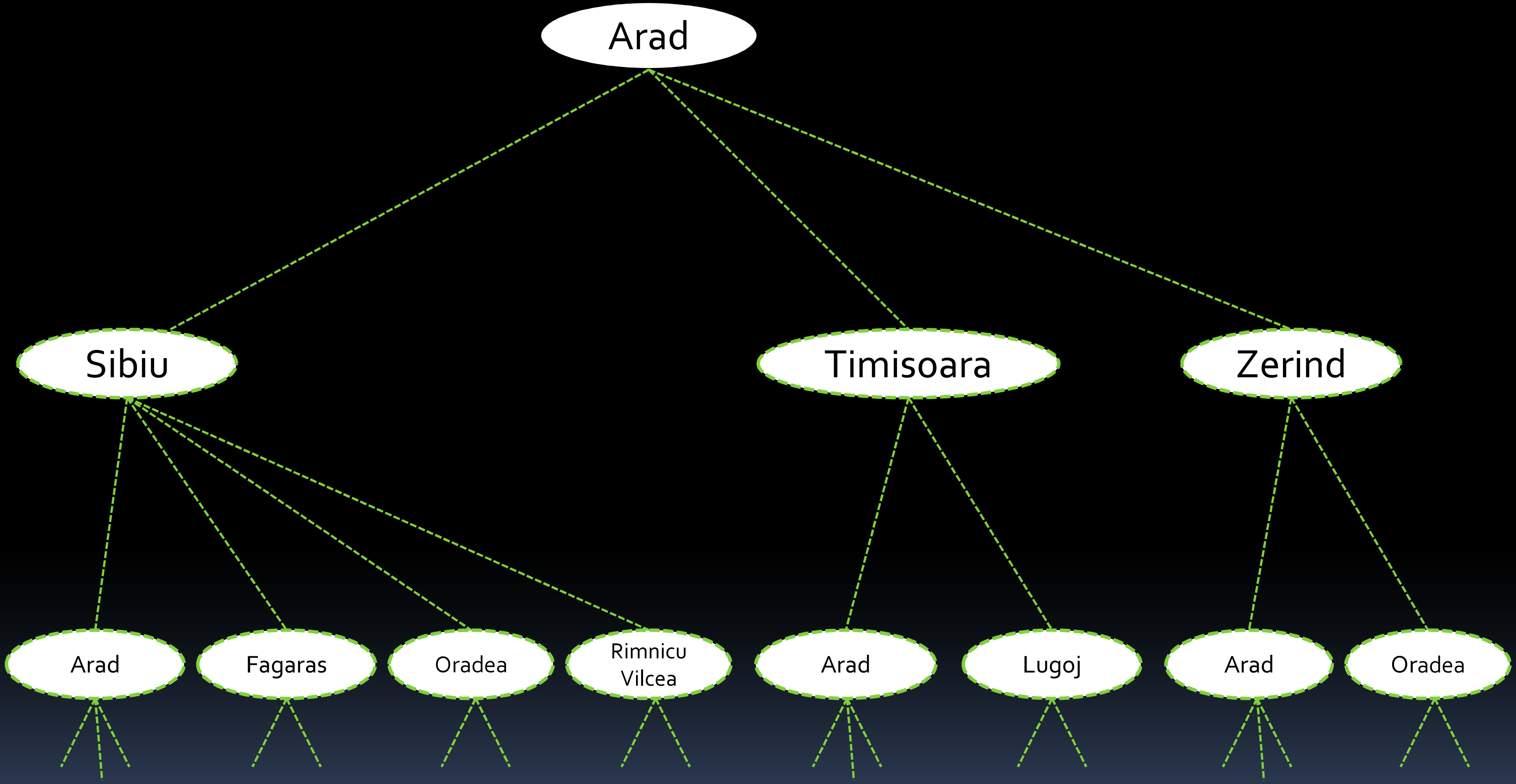
A Simple road map of Romania

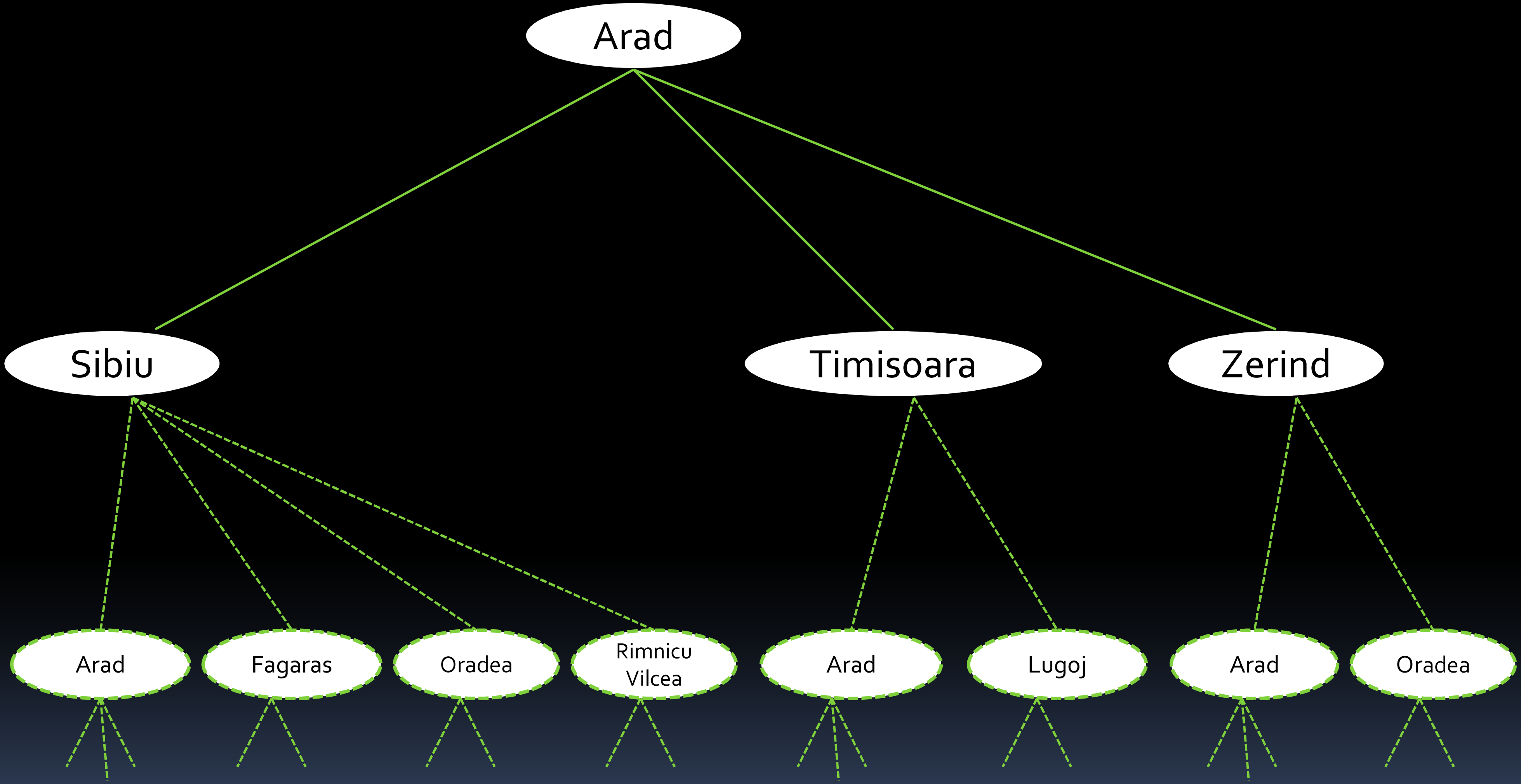


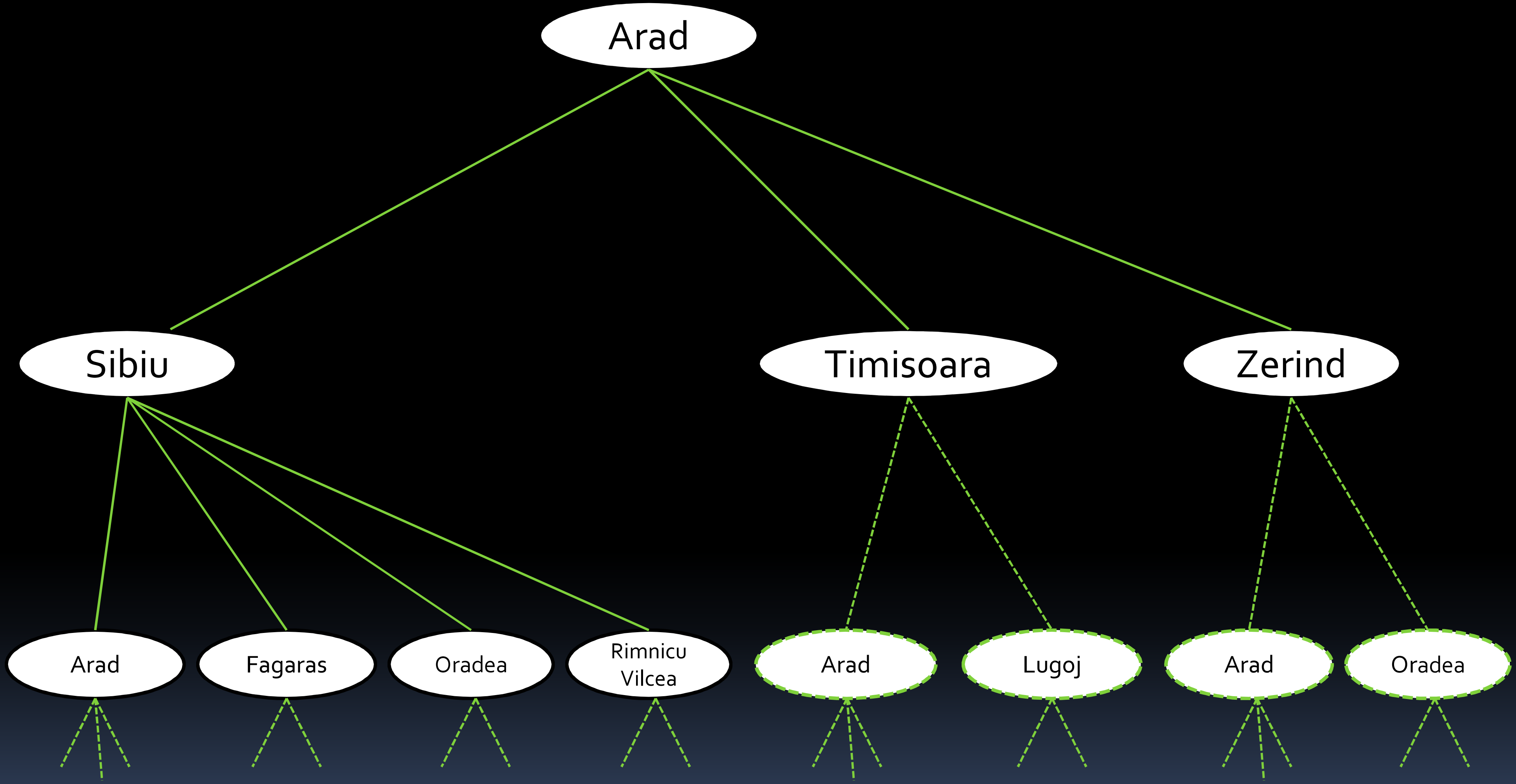
- Example Problems:
  - A **toy problem** can be easily used by different researches to compare the performance of algorithms.
    - Vacuum world
    - 8-puzzle
    - 8-queens
    - Sliding block puzzles
  - A **real world problem** is one whose solutions people actually care about.
    - Route finding problem
    - Touring problem
    - TSP problem
    - VLSI layout
    - Robot navigation
    - Automatic assembly sequencing
    - Internet searching

- Searching for solutions:
  - A **search tree** is generated by the initial state and the successor function that together define the state space.
  - In general, we may have **search graph** rather than a **search tree** when the same state can be reached from multiple paths.
  - **Search strategy** determines the choice of which state to expand.

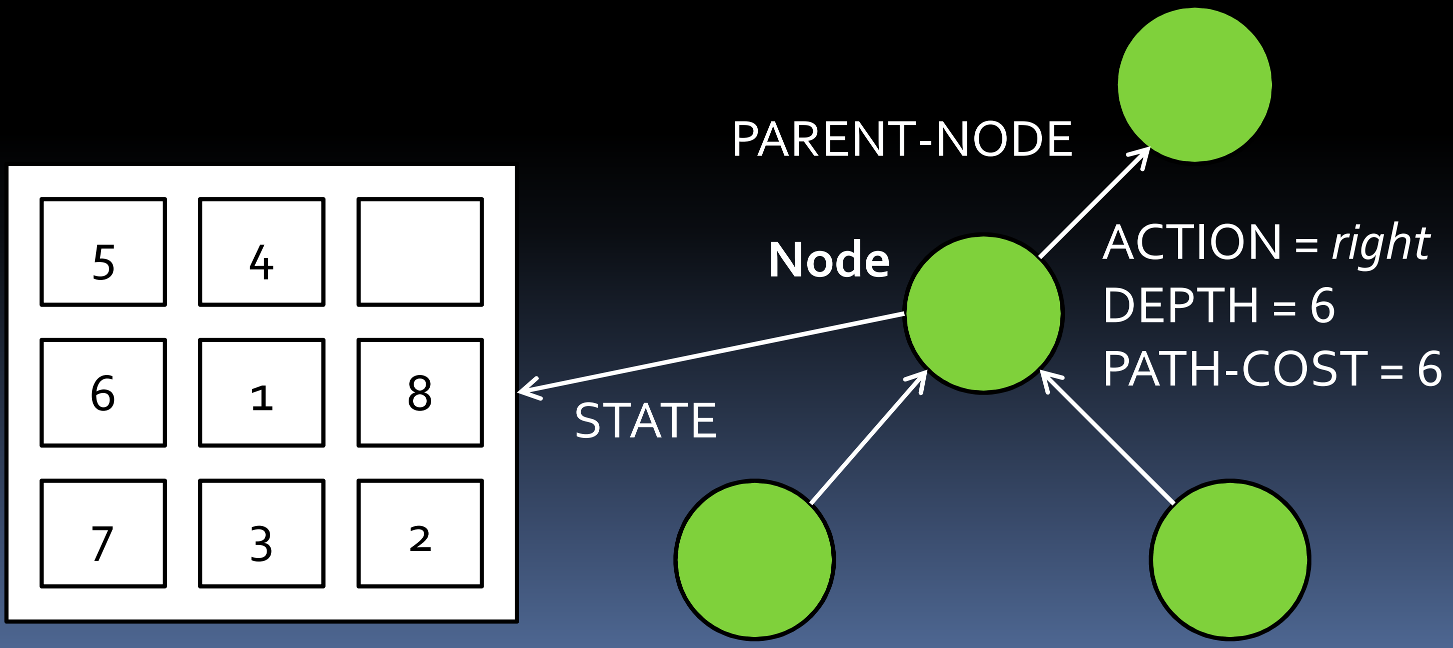
```
function Tree-Search( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state
    then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
end
```


Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea


Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea


Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

- A **state** is a (representation of) a physical configuration.
- A **node** is a data structure constituting part of a search tree includes parent, children, depth, path cost g(x).
- States do not have parents, children, depth, or path cost.



```
function Tree-Search( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if EMPTY?(fringe) then return failure
        node ← REMOVE-FIRST(fringe)
        if GOAL-TEST[problem] applied to STATE[node] succeeds                    then return
    SOLUTION(node)
        fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each <action, result> in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new Node
        STATE[s] ← result
        PARENT-NODE[s] ← node
        ACTION[s] ← action
        PATH-COST[s] ← PATH-COST[node]+STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
return successors
```

- A strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - Completeness – does it always find a solution if one exists?
  - Optimality – does it always find a least-cost solution?
  - Time complexity – number of nodes generated/expanded
  - Space complexity – maximum number of nodes in memory

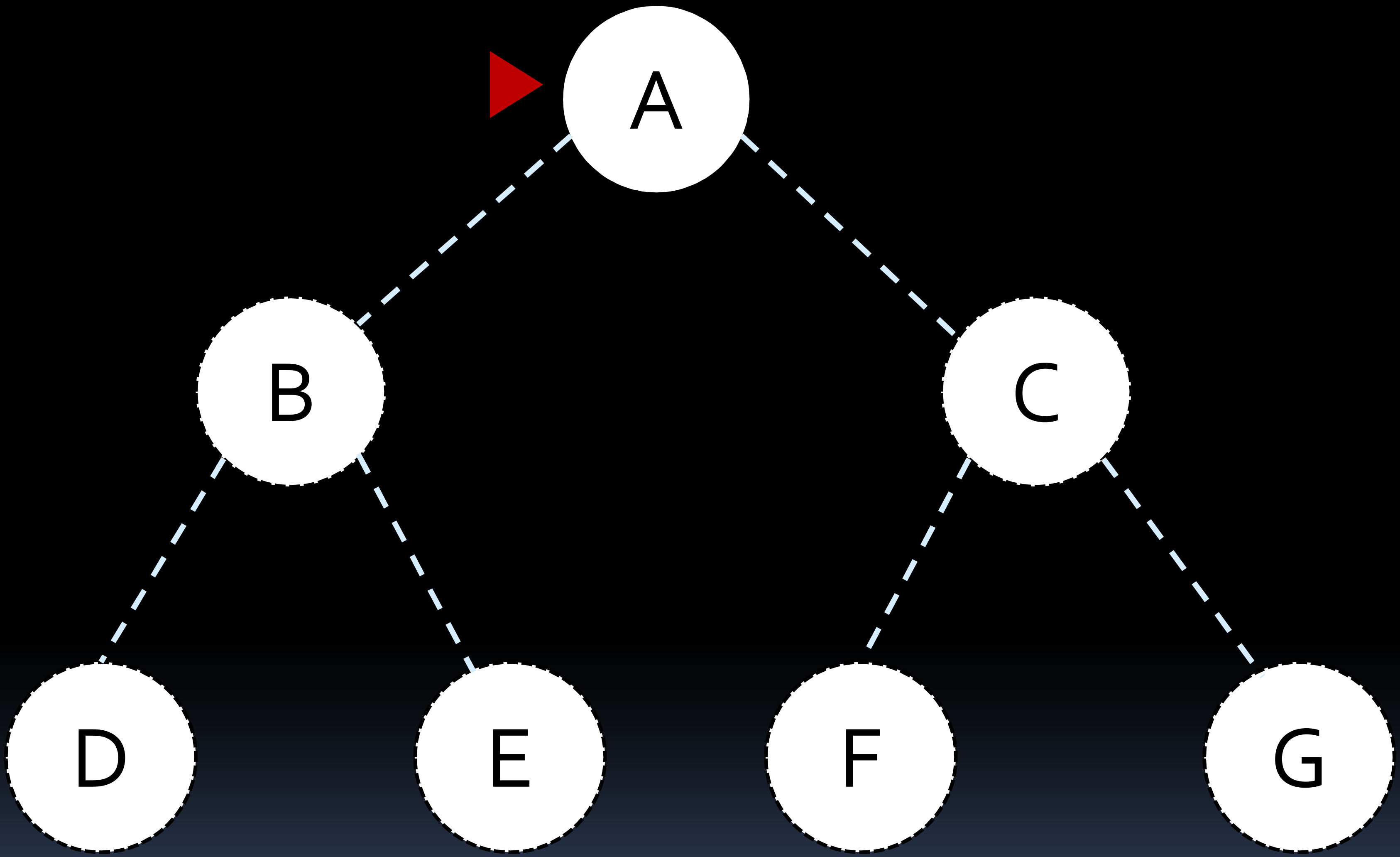
  Time and space complexity are measured in terms of:
  - b – maximum branching factor of the search tree
  - d – depth of the least-cost solution
  - C* - path cost of the least-cost solution
  - m – maximum depth of the state space (may be ∞)

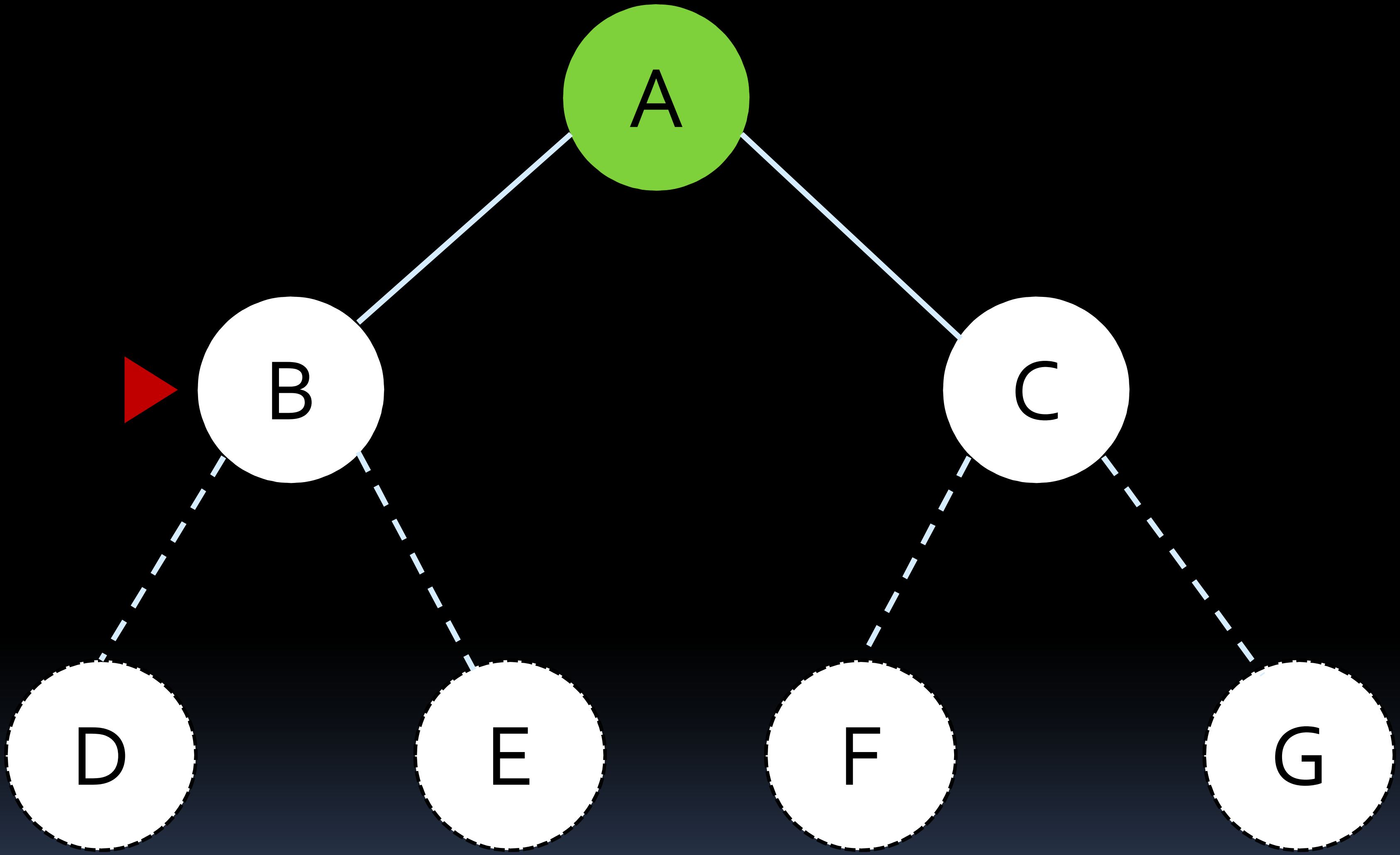
UNINFORMED SEARCH STRATEGIES

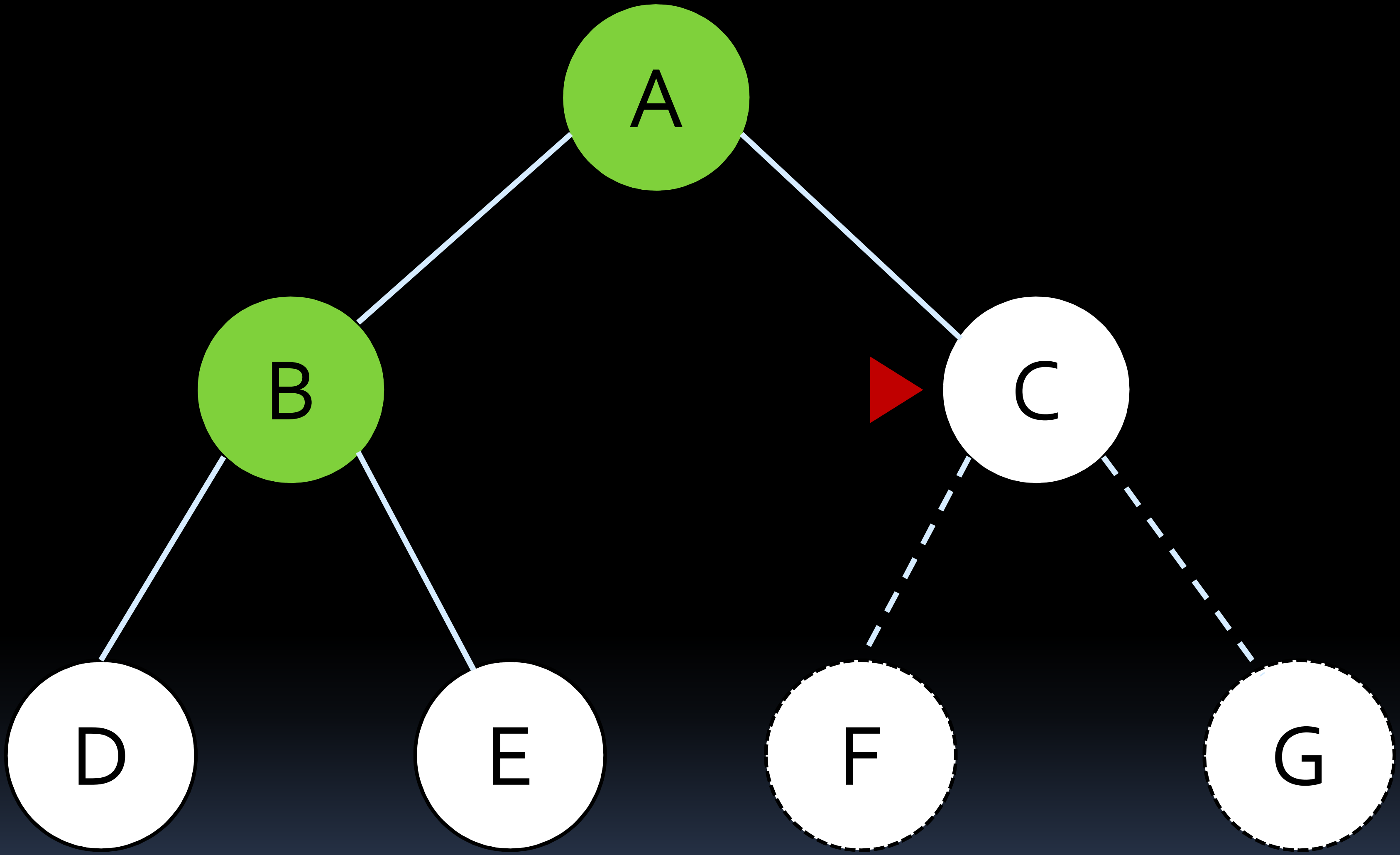
- It is otherwise known as blind search.
- It uses only the information available in the problem definition.
- They just generate successors and distinguish a goal state from a non-goal state, ie., all search strategies are distinguished by the order in which nodes are expanded.
  - Breadth-First Search
  - Uniform-Cost Search
  - Depth-First Search
  - Depth-Limited Search
  - Iterative Deepening Depth-First Search
  - Bi-directional Search

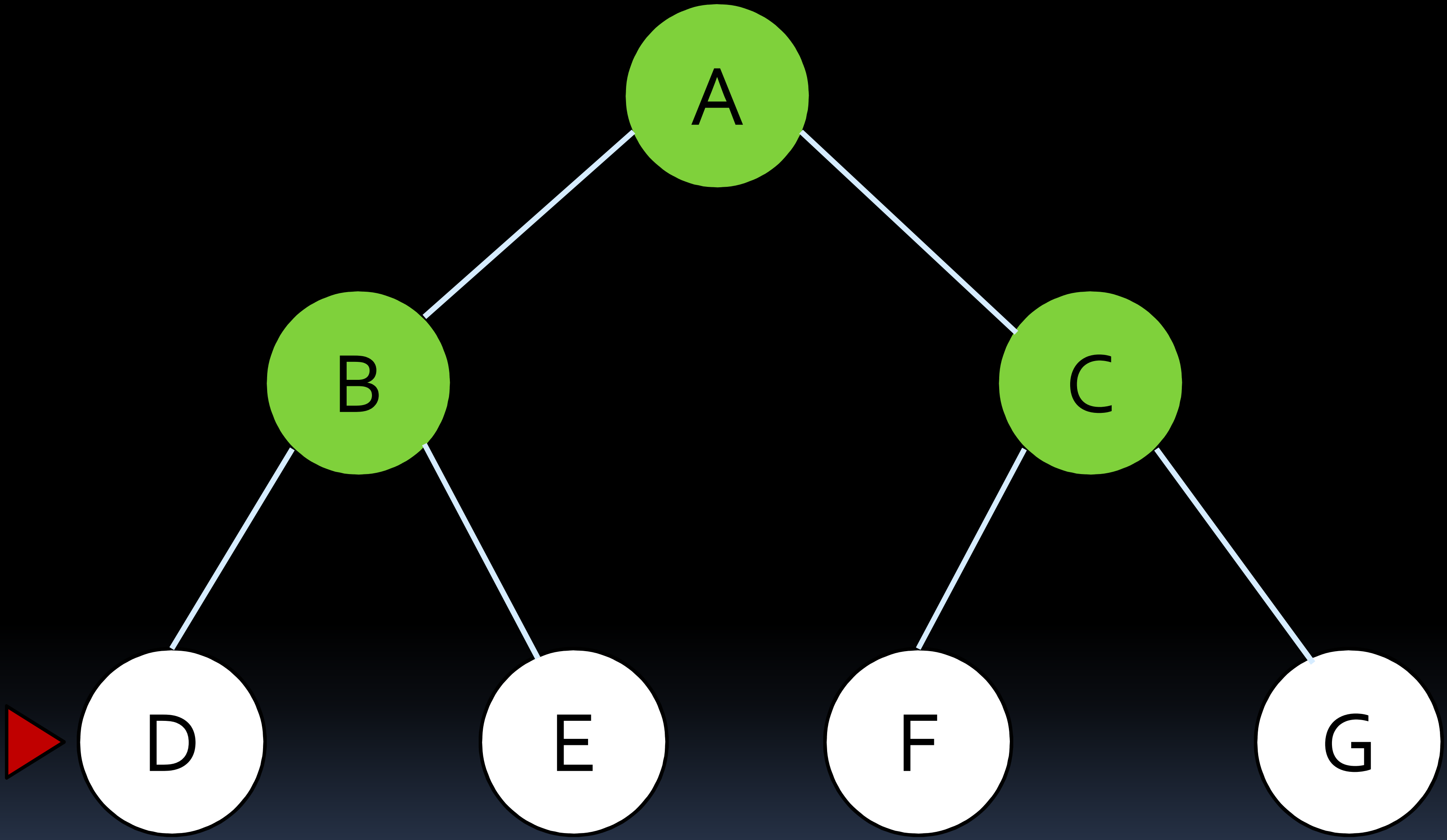
Breadth First Search

- Expands shallowest unexpanded node.
- It can be implemented by calling TREE-SEARCH with an empty fringe, ie., FIFO queue (i.e., new successors go at end)


A
B
C
D
E
F
G


A
B
C
D
E
F
G


A
B
C
D
E
F
G

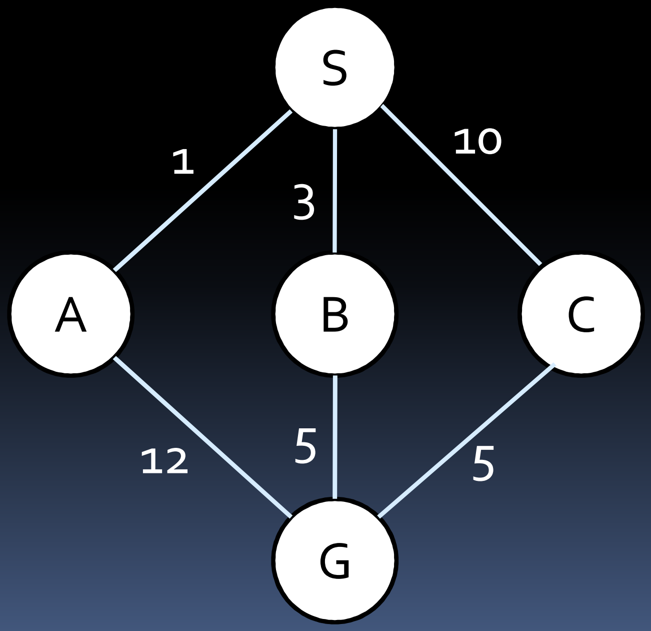

A
B
C
D
E
F
G

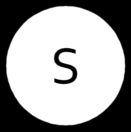
Properties of breadth first search:

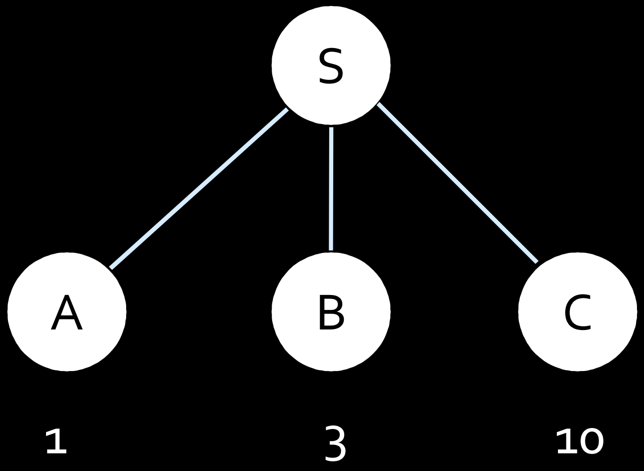
- Completeness? Yes (if b is finite)
- Optimality? No, unless step costs are constant
- Time complexity? $1+b+b^2+b^3+\ldots+b^d+b(b^d-1) = O(b^{d+1})$
- Space complexity? $O(b^{d+1})$ (keeps every node in memory)

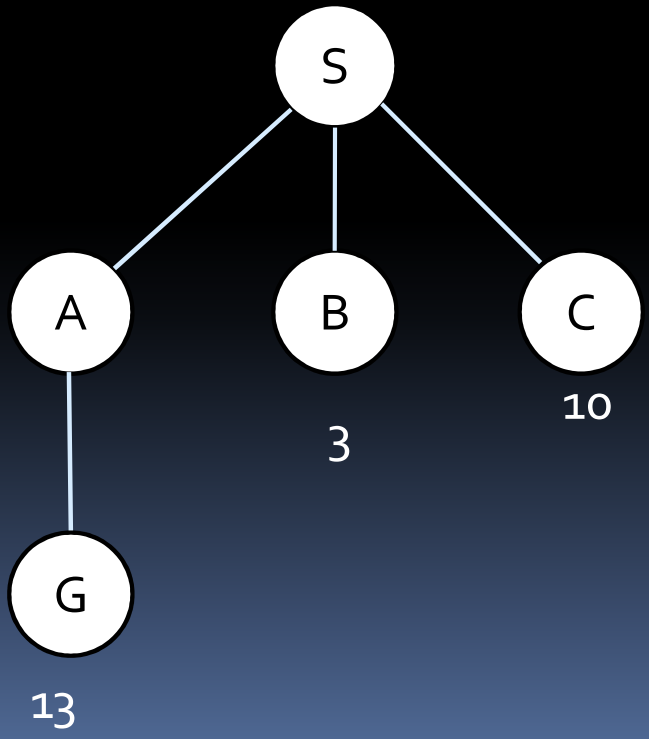
Uniform Cost Search

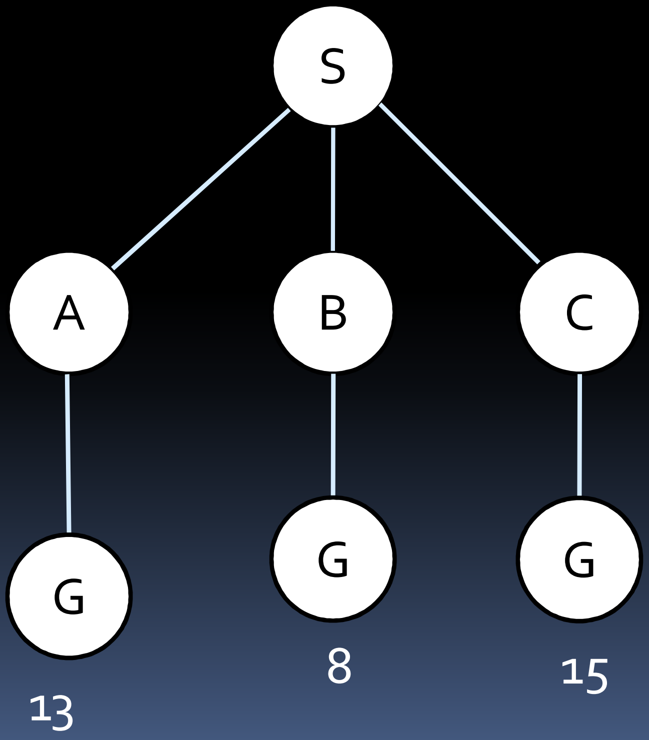
- Expands least cost unexpanded node.
- It is equivalent to breadth first search if all step costs are equal.
- Implementation: fringe = queue ordered by path cost, lower first.




S


S
A
B
C
1
3
10


S
A
B
C
G
3
10
13

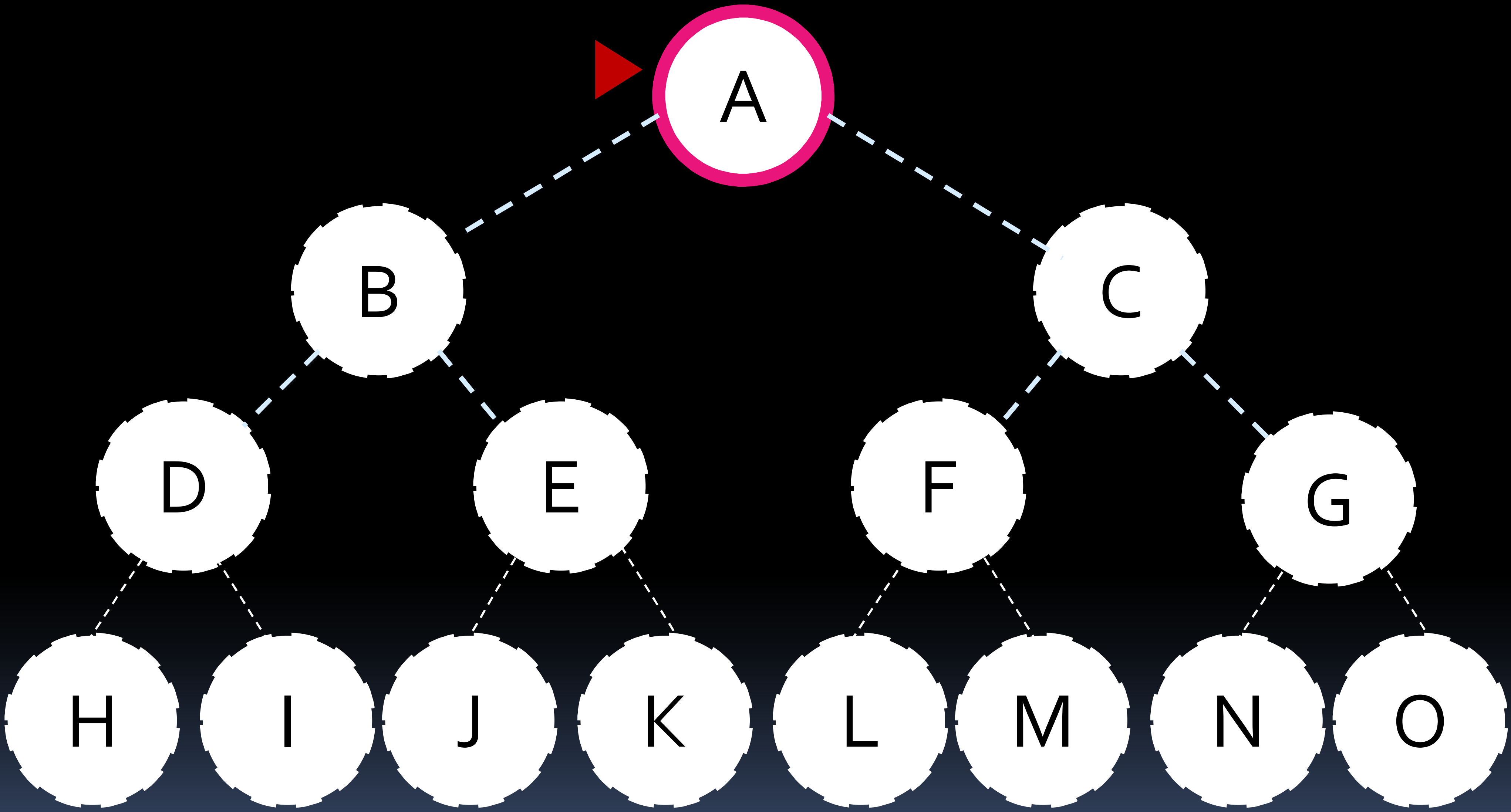

S
A
B
C
G
G
G
13
8
15

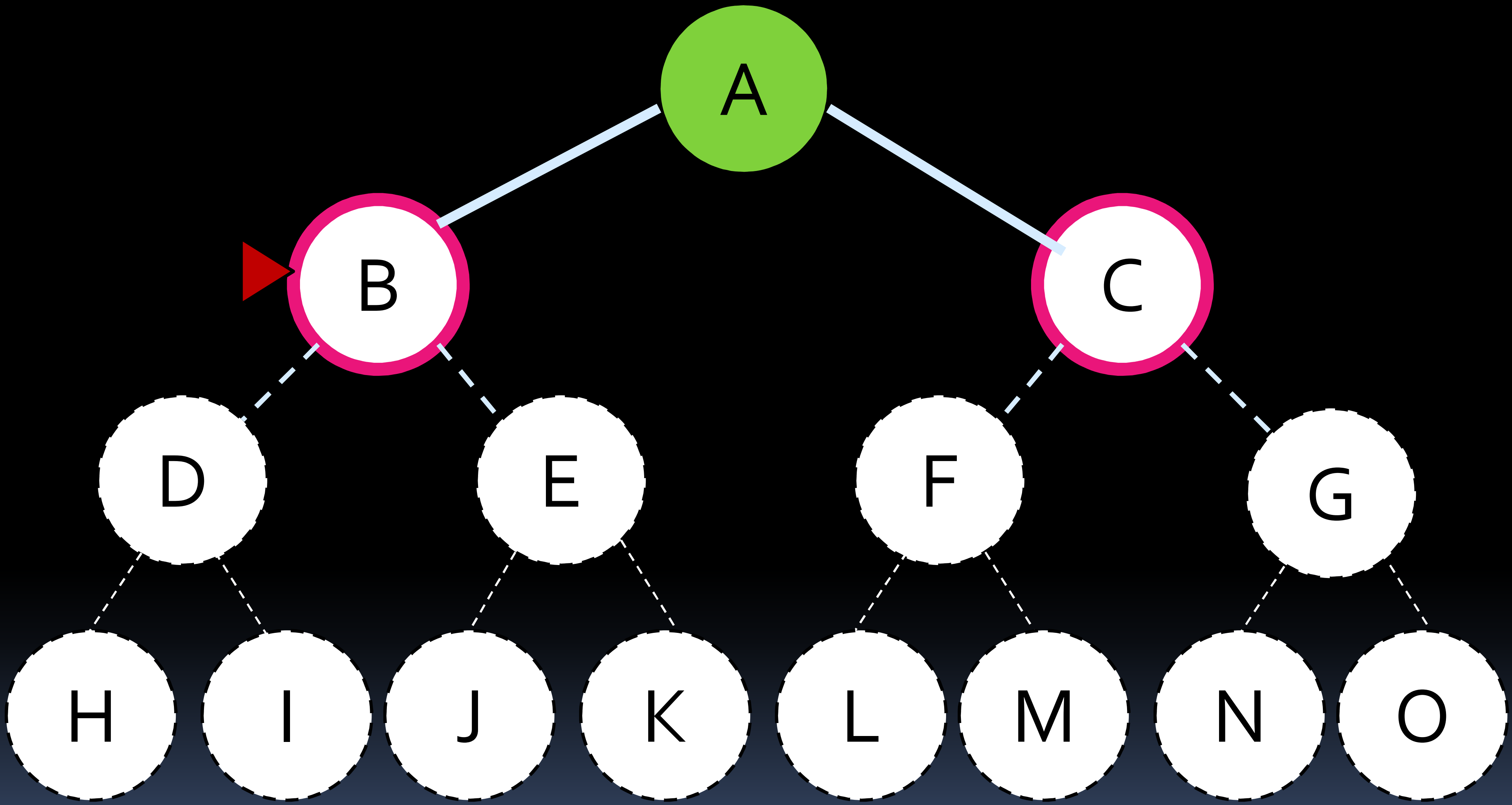
Properties of uniform cost search:

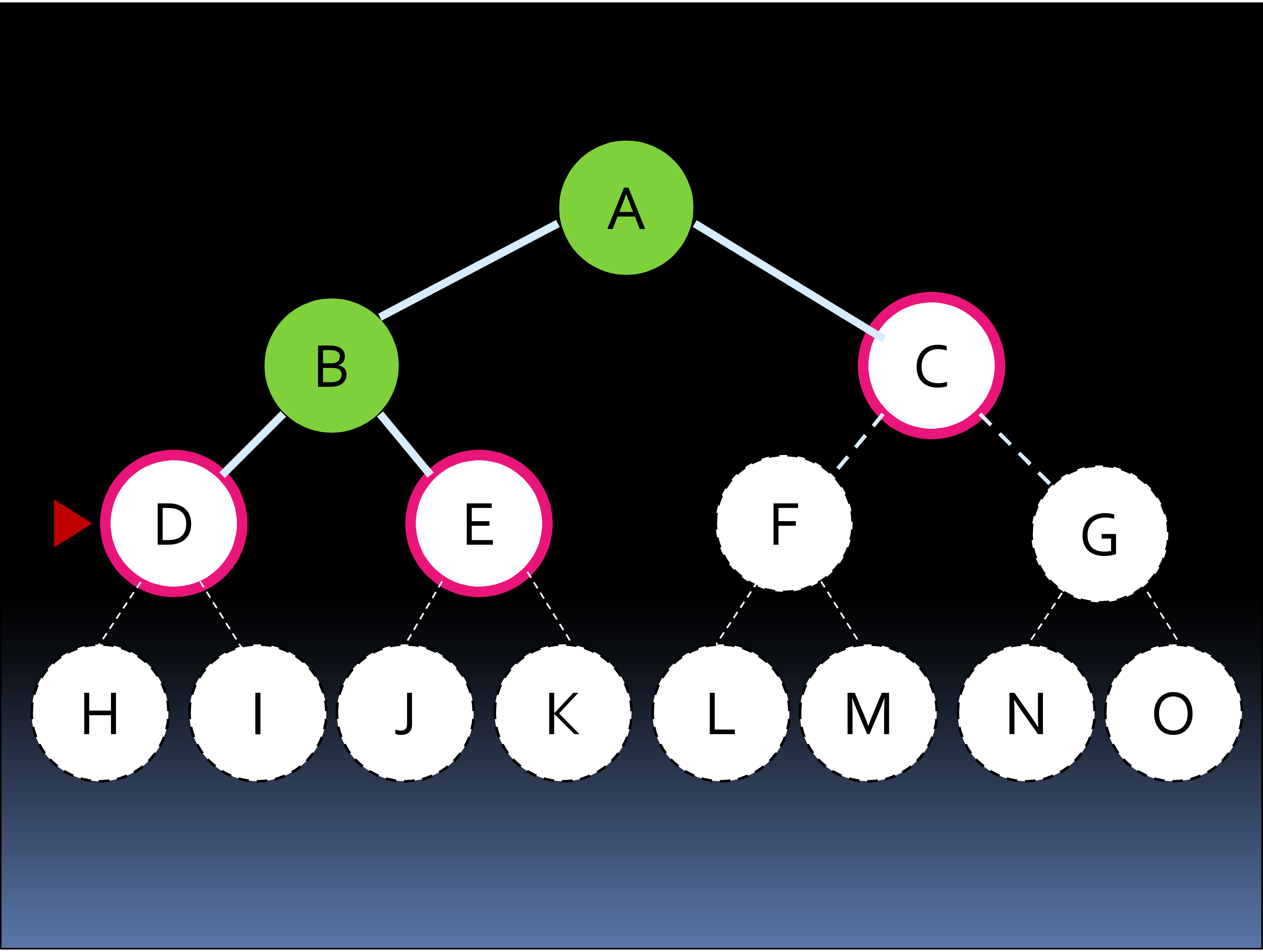
- Completeness? Yes (if step cost ≥ `)
- Optimality? No, unless step costs are constant
- Time complexity? $1+b+b^2+b^3+\ldots+b^d+b(b^d-1) = O(b^{d+1})$
- Space complexity? $O(b^{d+1})$ (keeps every node in memory)

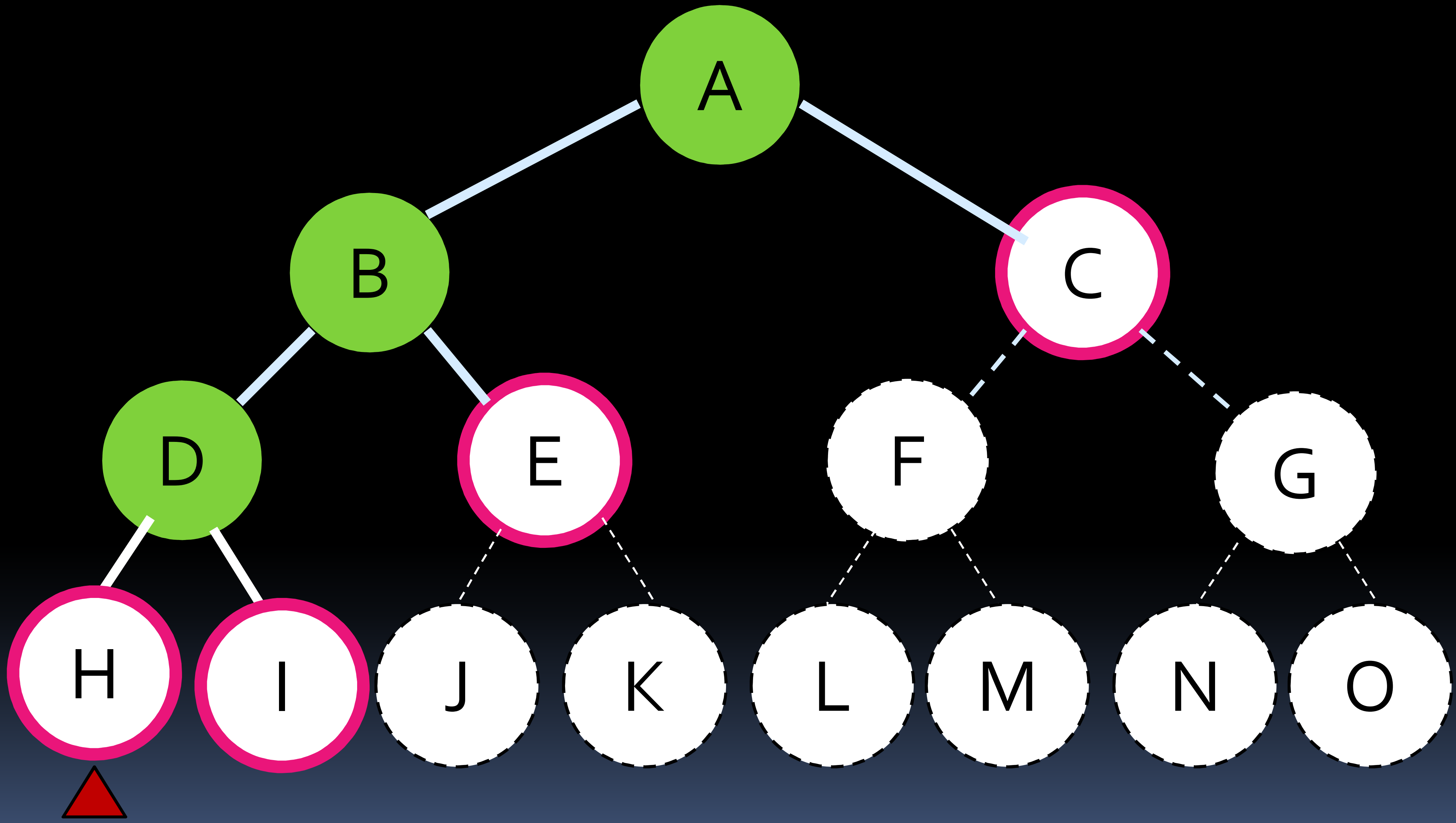
Depth First Search

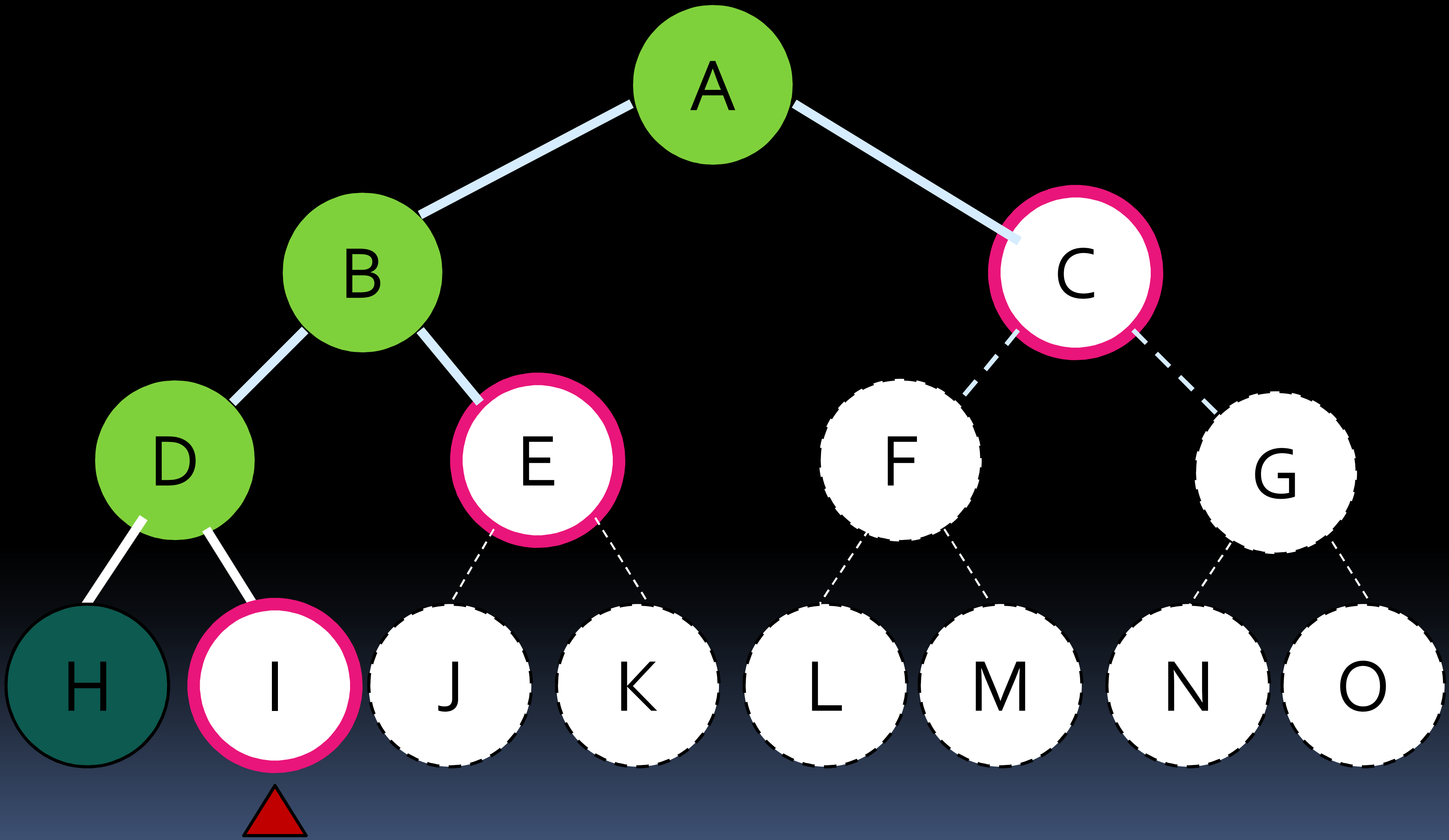
- Expands the deepest node in the current fringe.
- Implementation: fringe = LIFO queue, i.e., put successors at front.

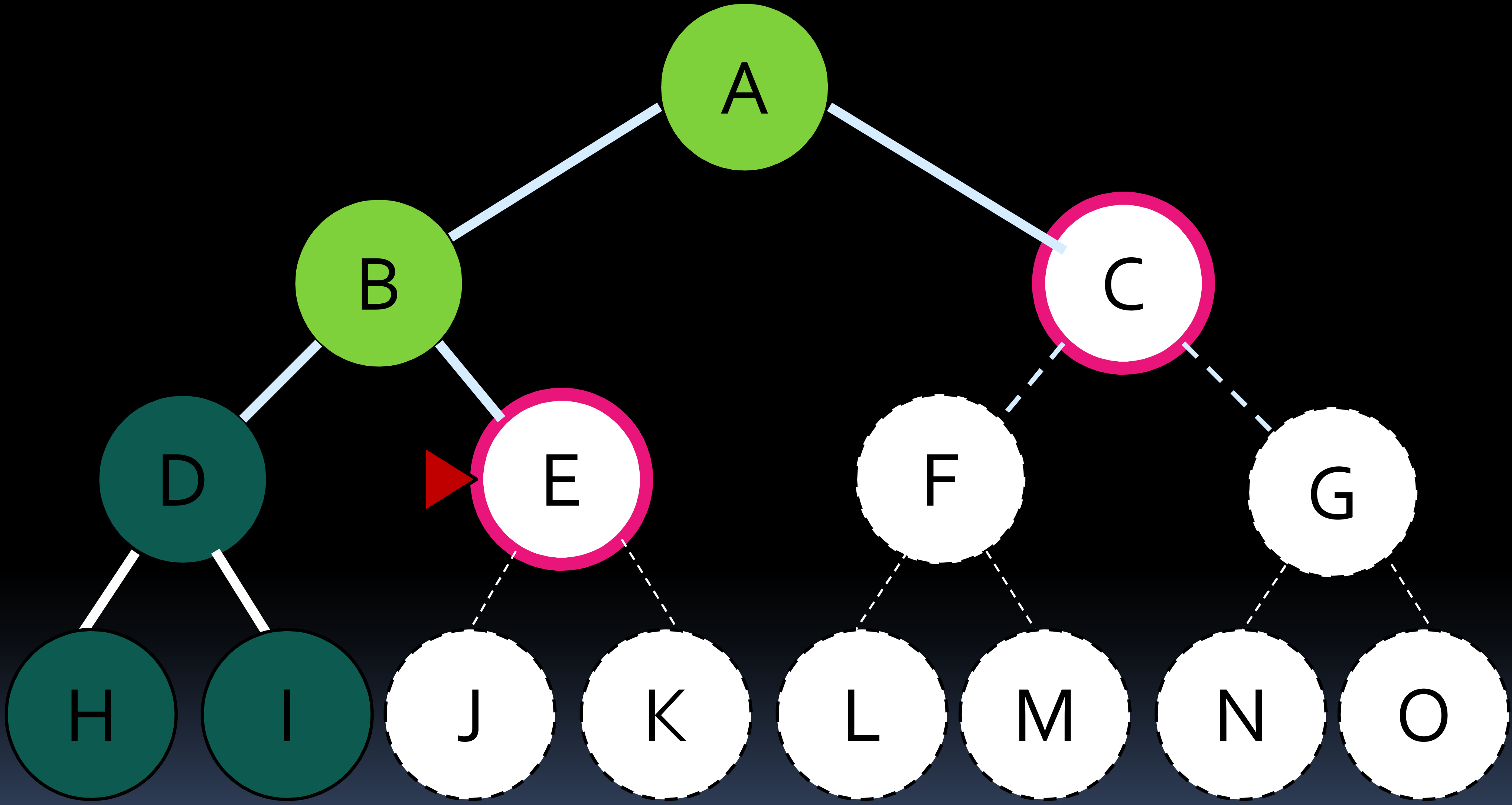

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

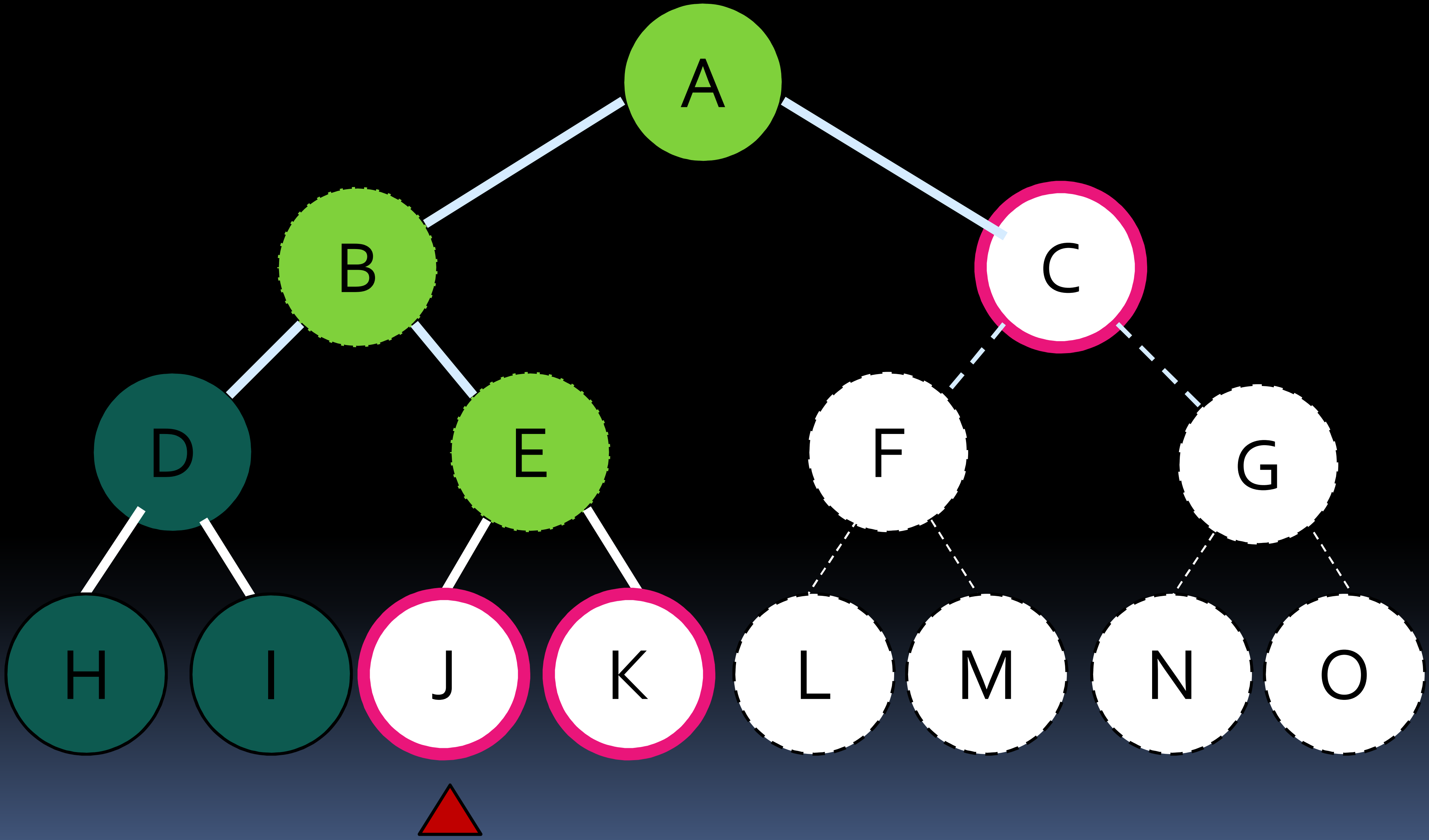

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

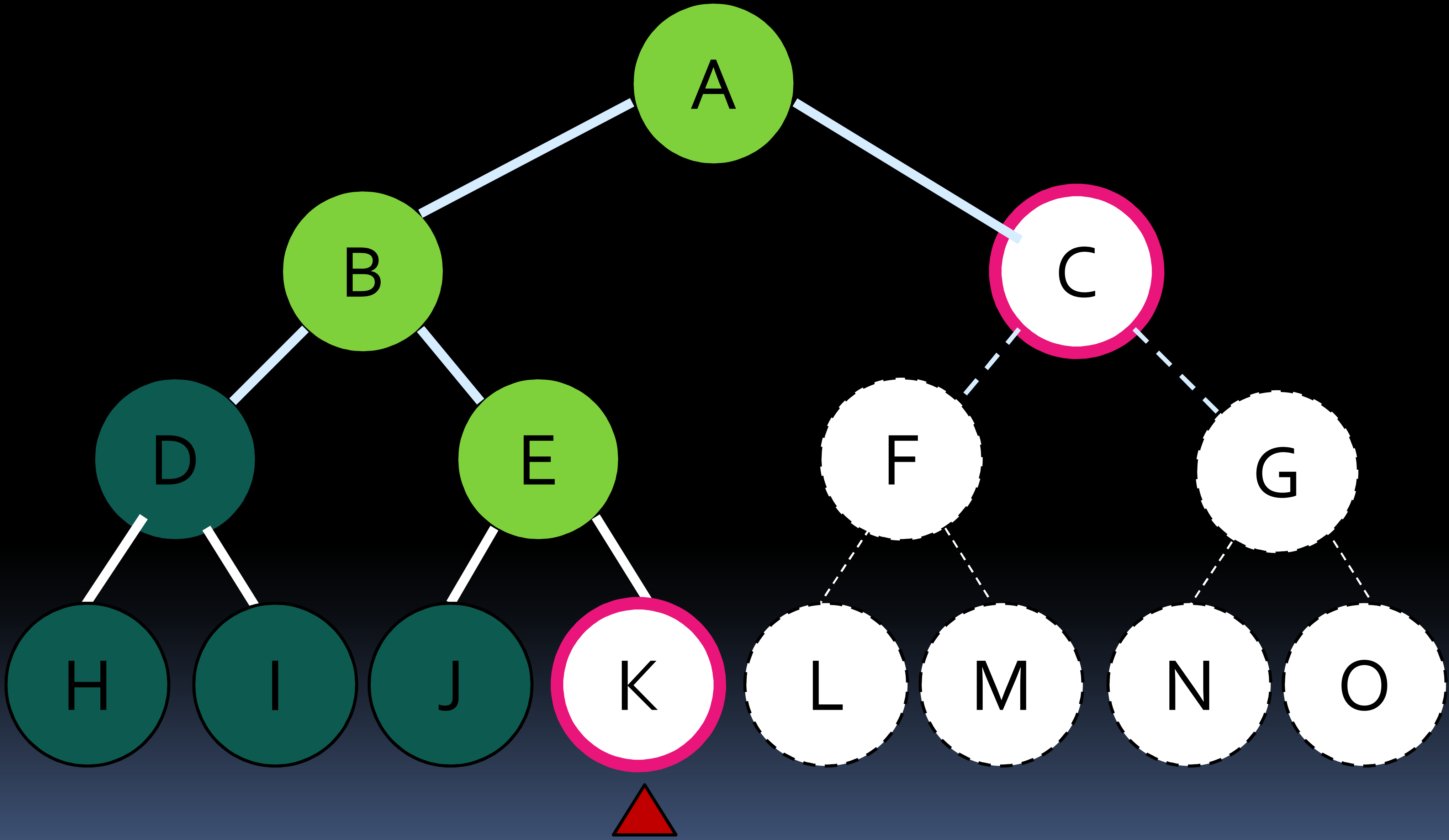

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

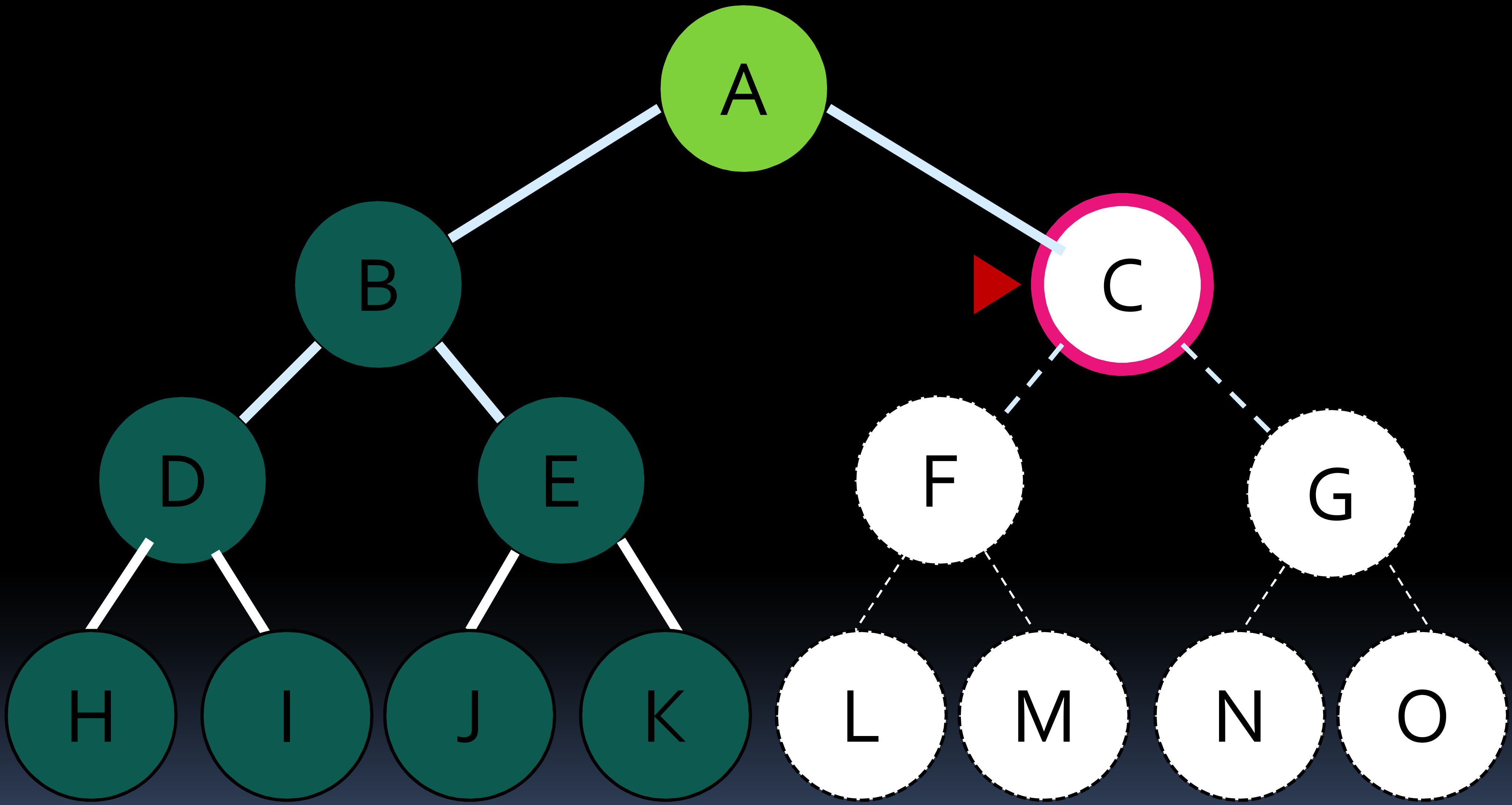

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

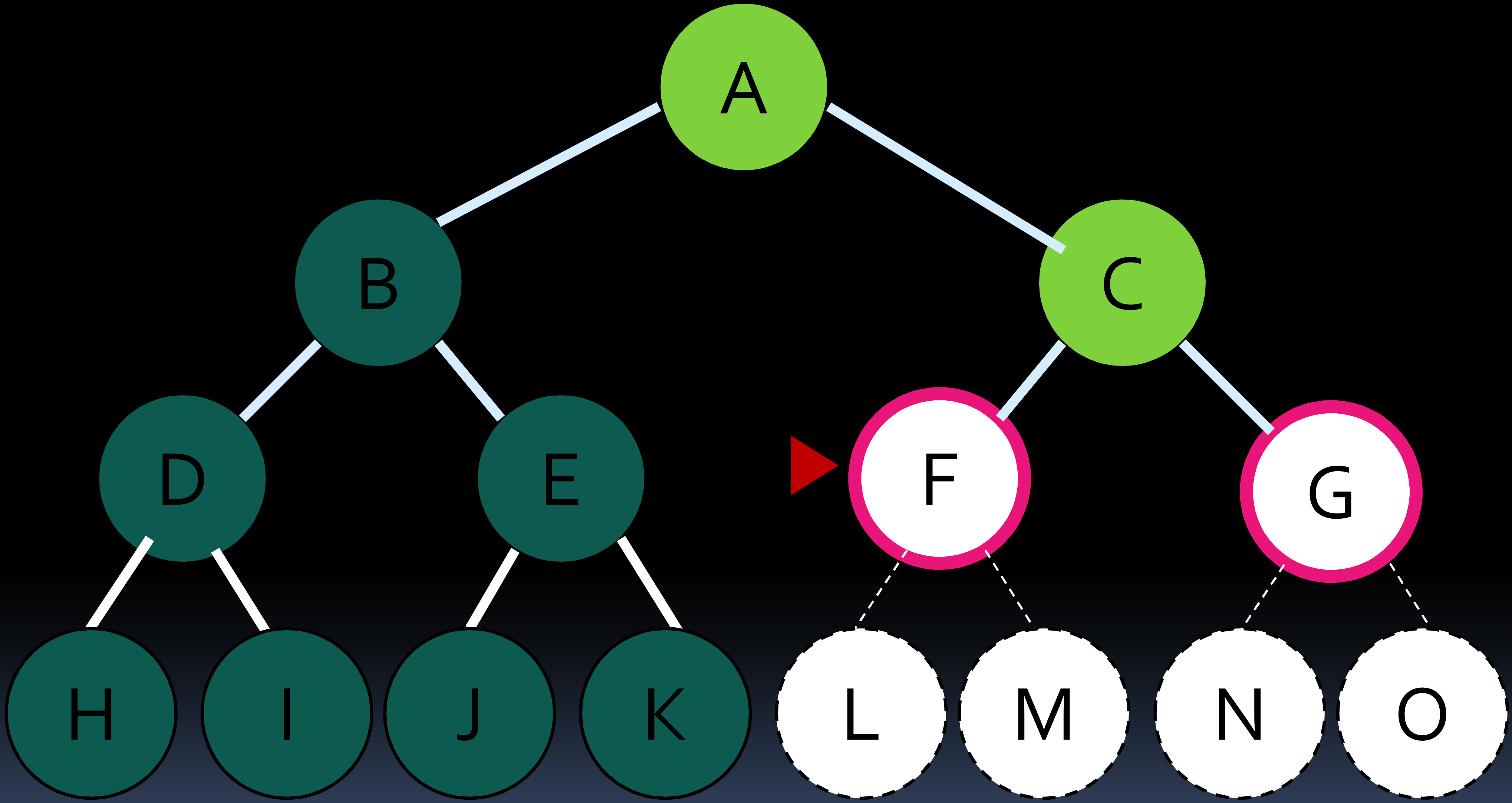

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

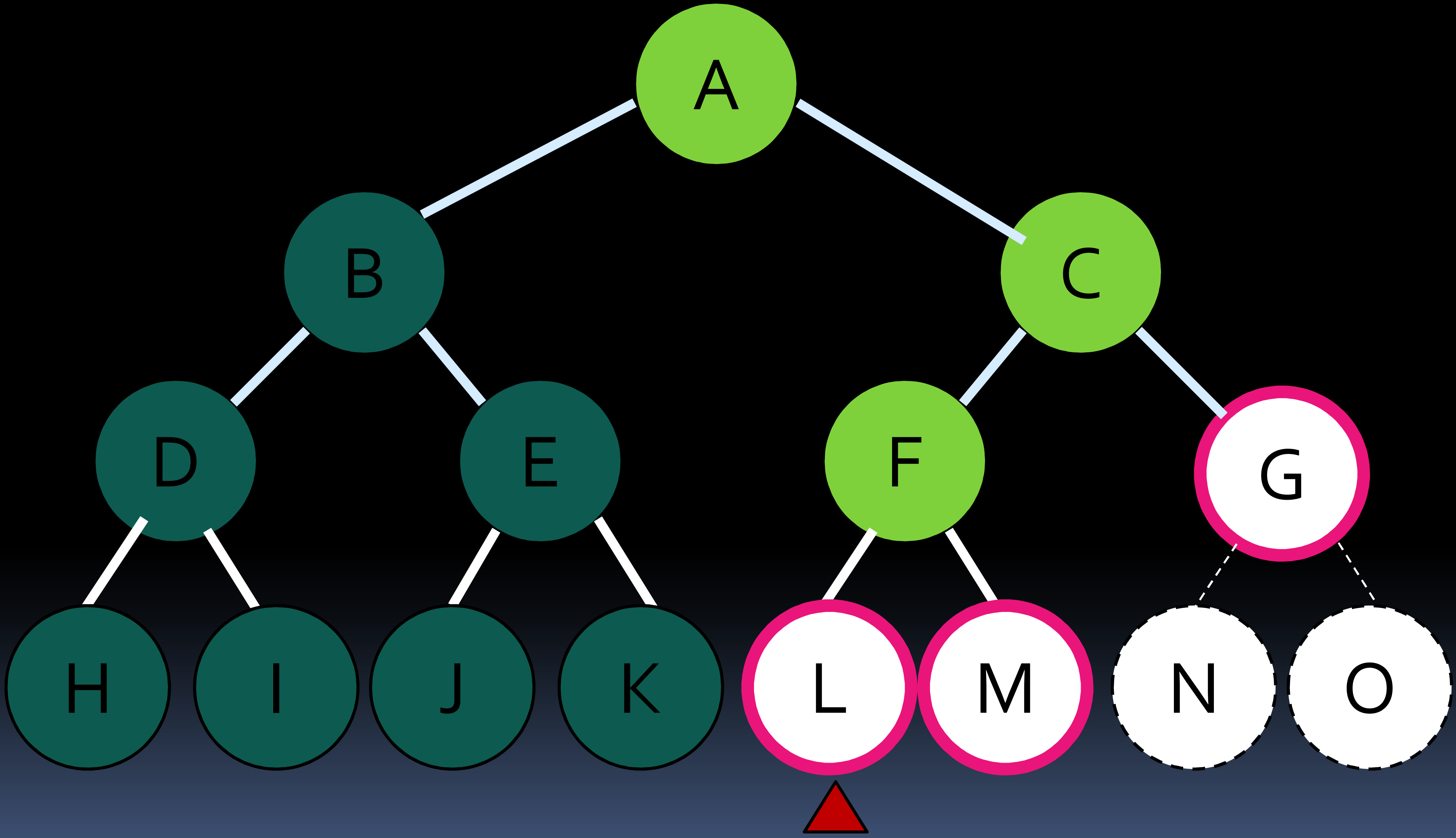

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

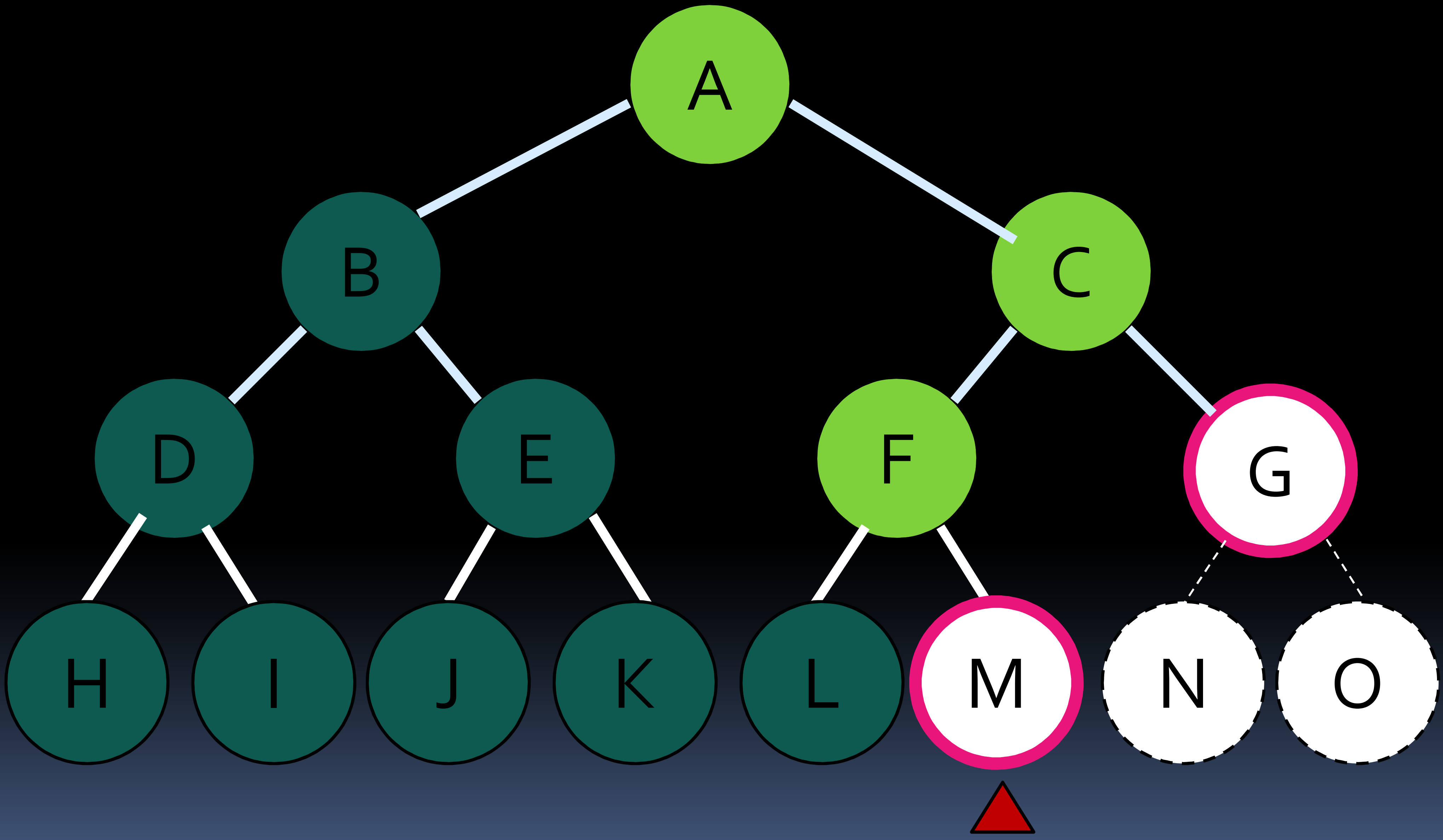

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

- Properties of Depth-First Search:
  - Complete? No: fails in infinite-depth spaces, spaces with loops.

    Modify to avoid repeated states along path => complete in infinite spaces
  - Time? O(b^m): terrible if *m* is much larger than *d* but if solutions are dense, may be much faster than breadth-first.
  - Space? O(bm), i.e., linear space.
  - Optimal? No

Depth-Limited Search

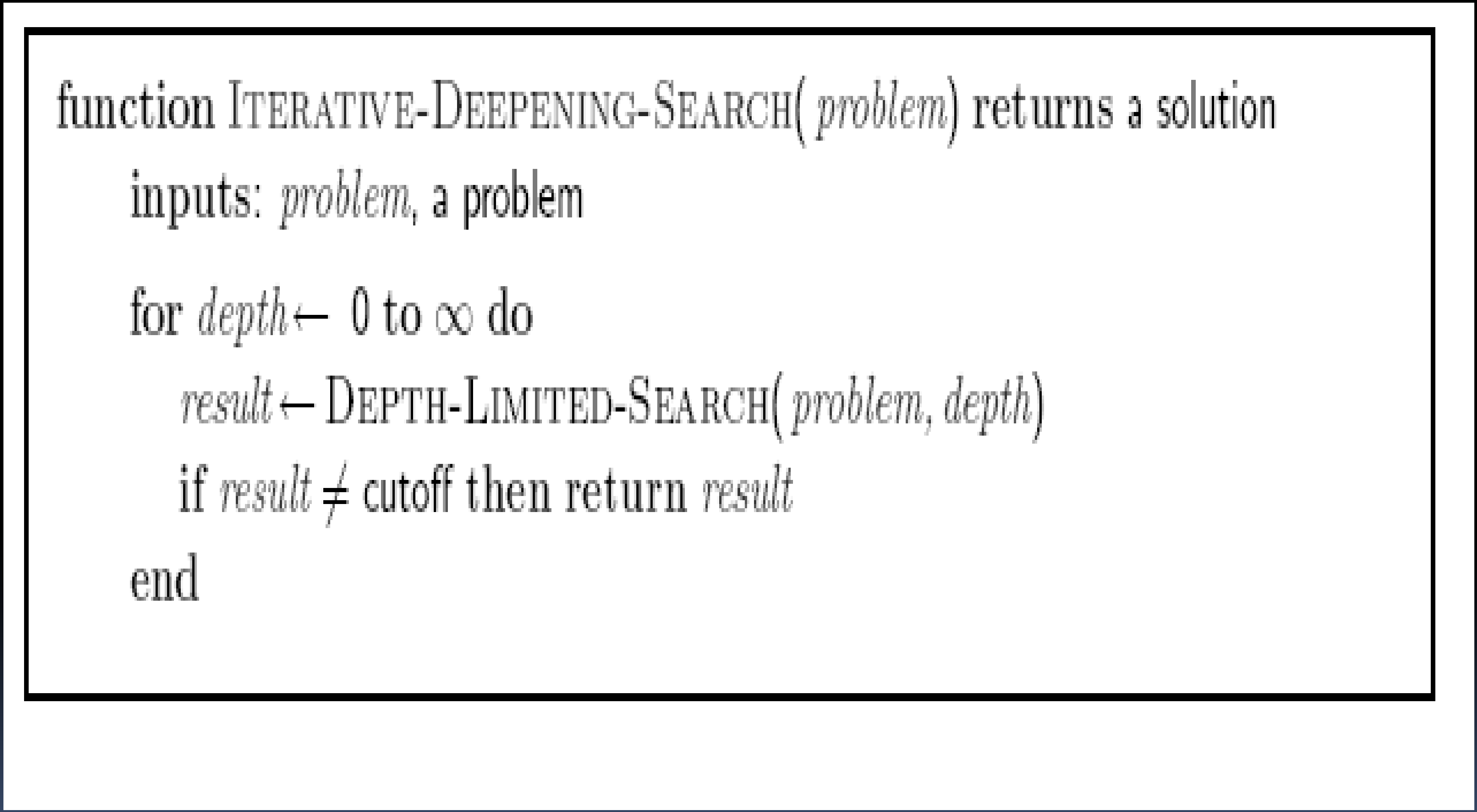
- Depth-first search with depth limit *l*, returns *cutoff* if any path is cut off by depth limit.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln
                                              /fail /cutoff
        RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]),
                                                    problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln /fail
                                                           /cutoff
        cutoff-occurred? ← false
        if GOAL-TEST(problem,STATE[node]) then return node
        else if DEPTH[node] = limit then return cutoff
        else for each successor in EXPAND(node, problem) do
                result ← RECURSIVE-DLS(successor, problem, limit)
                if result = cutoff then cutoff-occurred? ← true
                else if result ≠ failure then return result
        if cutoff-occurred? then return cutoff else return failure
```

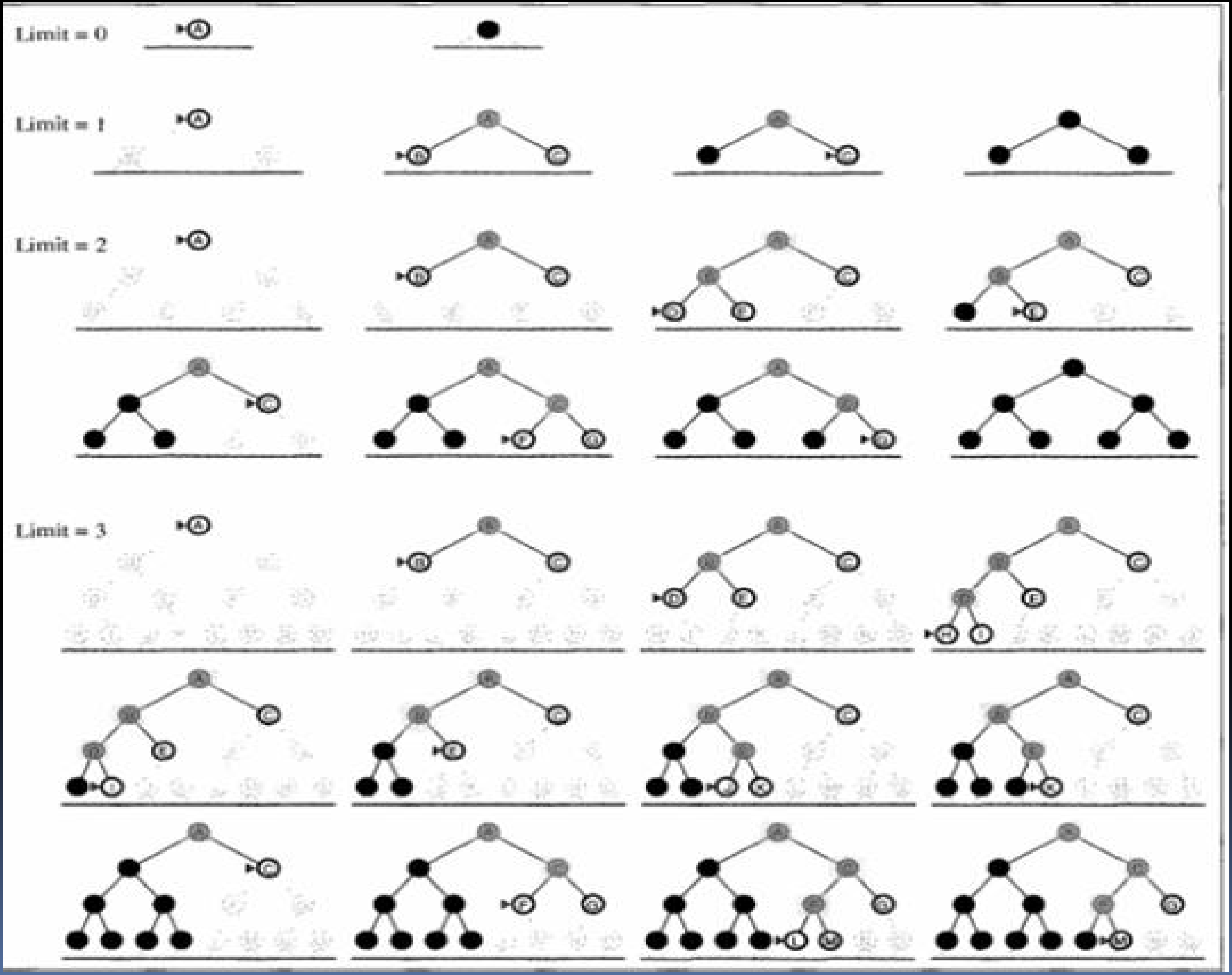
- Properties of Depth-First Search:
  - Complete? No
  - Time? O(b^{l})
  - Space? O(bl)
  - Optimal? No

Iterative Deepening Depth-First Search

- Otherwise called Iterative deepening search is a general strategy often used in combination with depth-first search, that finds the better depth limit.
- It is done by gradually increasing the limit – first 0, then 1, then 2 and so on – until goal is found.
- It combines the benefits of depth-first search and breadth-first search.



```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
    end
```


Limit = 0
Limit = 1
Limit = 2
Limit = 3

- Properties of Iterative deepening search:
  - Complete? Yes
  - Time? $(d + 1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
  - Space? $O(bd)$
  - Optimal? No, unless step costs are constant

    Can be modified to explore uniform-cost tree.

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$N(\text{IDS}) = 50 + 100 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$

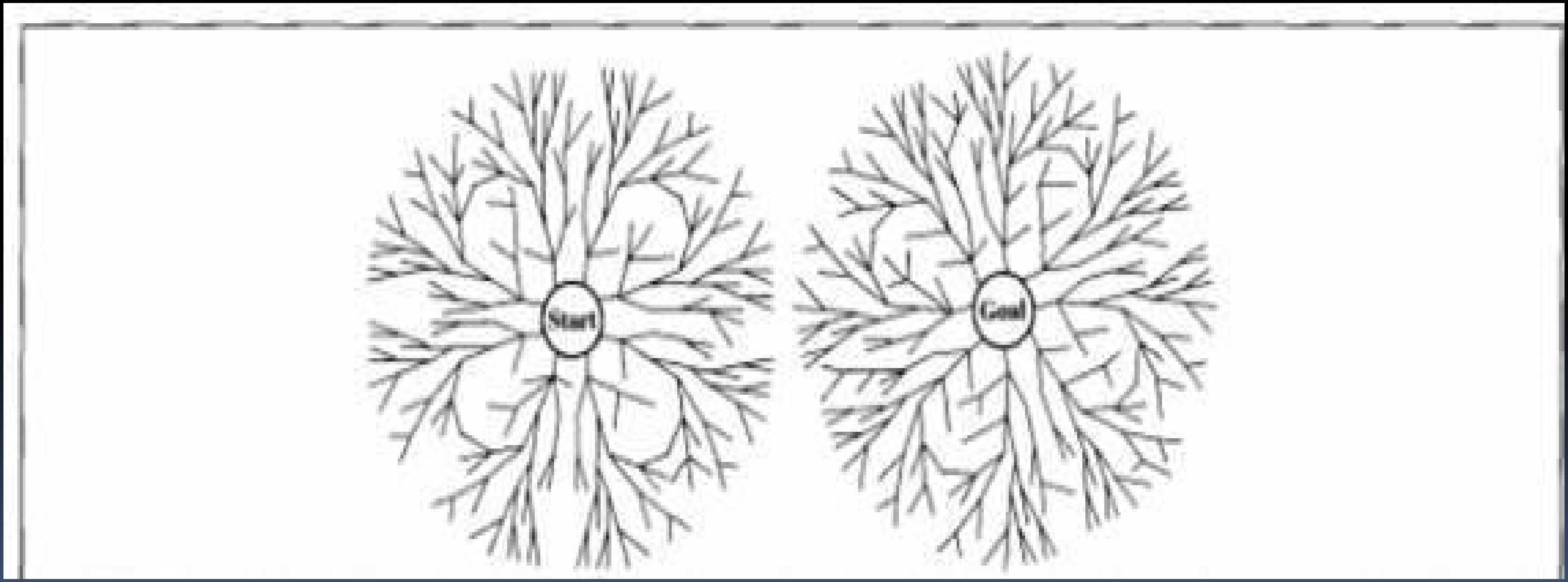
$N(\text{BFS}) = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 + 999{,}990 = 1{,}111{,}100$

IDS does better because other nodes at depth *d* are not expanded

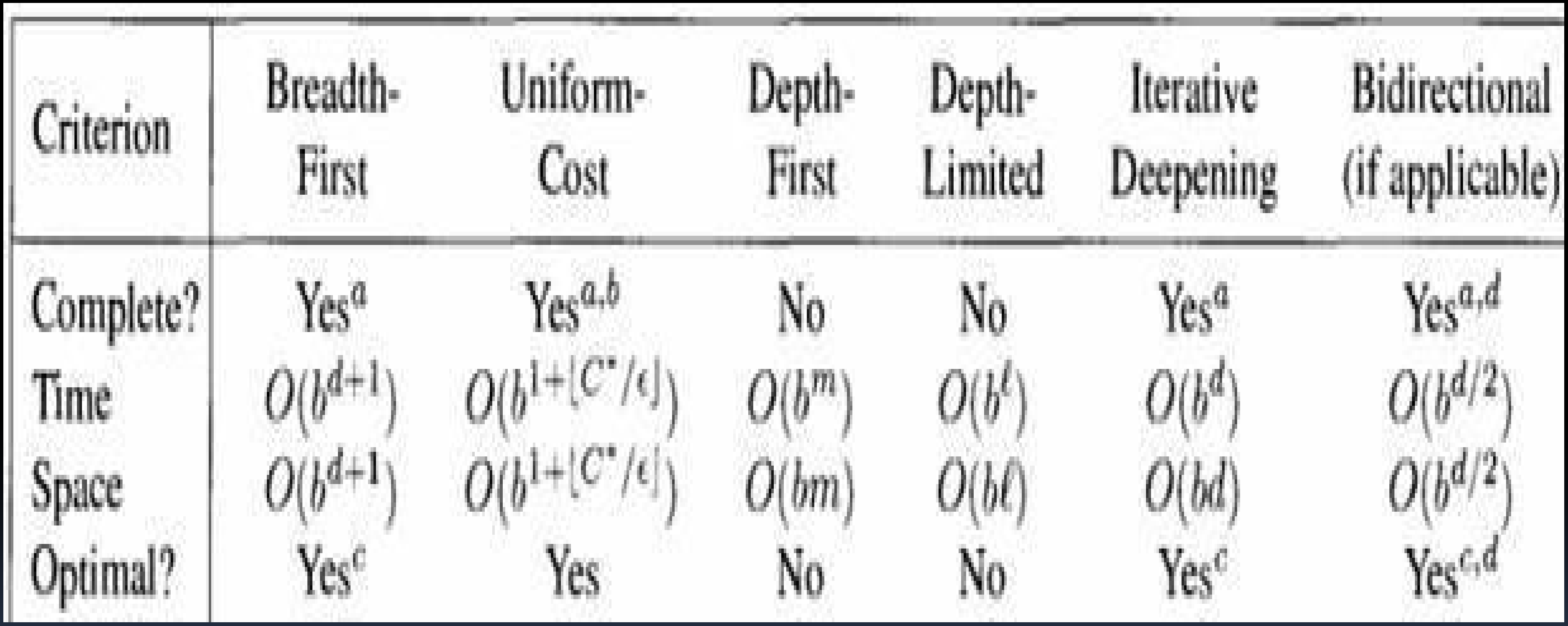
BFS can be modified to apply goal test when a node is generated.

Bidirectional Search

- Runs two simultaneous searches – one forward from the initial state and the other backward from the goal.



Comparing Uninformed Search Strategies

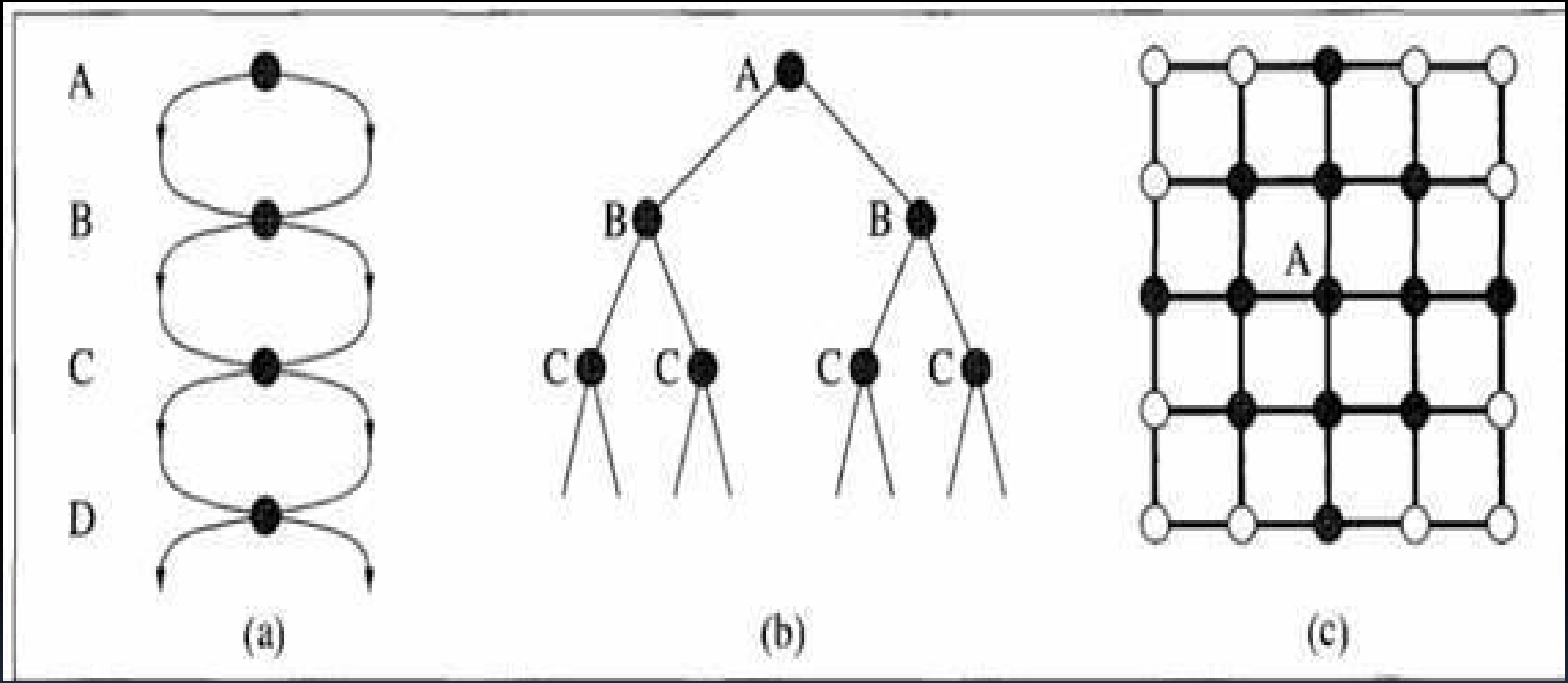


Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Avoiding Repeated States

- In searching, time is wasted by expanding states that have already been encountered and expanded before.
- For some problems repeated states are unavoidable.
- The search trees for these problems are infinite.
- If we prune some of the repeated states, we can cut the search tree down to finite size.
- Considering search tree upto a fixed depth, eliminating repeated states yields an exponential reduction in search cost.

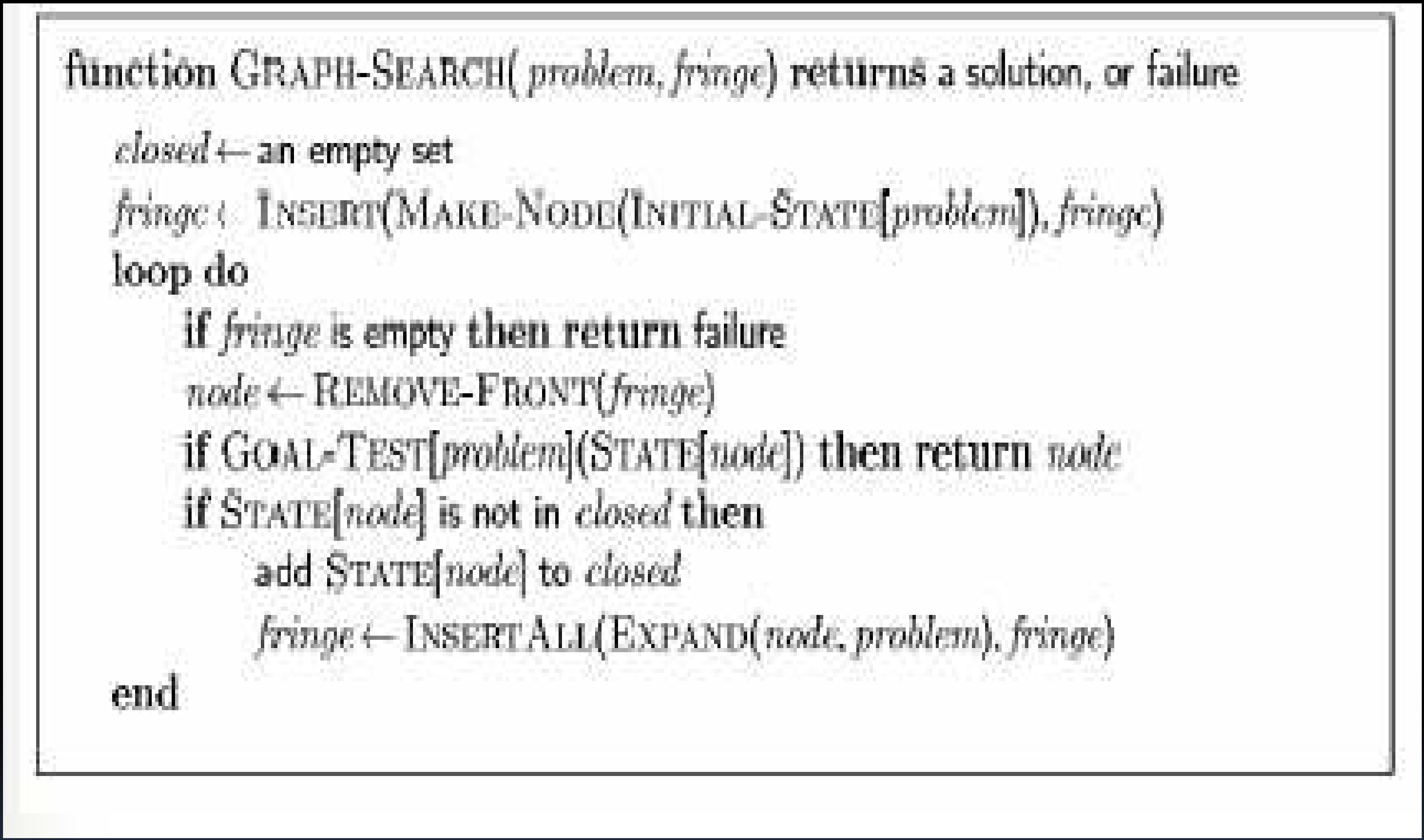
- Repeated states ,can cause a solvable problem to become unsolvable if the algorithm does not detect them.
- Repeated states can be the source of great inefficiency: identical sub trees will be explored many times.


A
B
C
D
(a)
A
B
B
C
C
C
C
(b)
A
(c)

(a) A state space in which there are two possible actions leading from A to B, two from B to C, and so on. The state space contains d + 1 states, where d is the maximum depth.

(b) The corresponding search tree, which has 2d branches corresponding to the 2d paths through the space.

(c) A rectangular grid space. States within 2 steps of the initial state (A) are shown in white.



```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

The General Graph search algorithm. The set closed can be implemented with a hash table to allow efficient checking of repeated states.

Searching with Partial Solutions

- Different types of incompleteness lead to three distinct problem types:
  - **Sensorless problems (conformant): If the agent has no sensors at all**
  - **Contingency problem: if the environment if partially observable or if action are uncertain (adversarial)**
  - **Exploration problems: When the states and actions of the environment are unknown.**

INFORMED (HEURISTICS) SEARCH STRATEGIES

- **Informed search** is one that uses problem-specific knowledge beyond the definition of the problem itself.
- It can find solutions more efficiently than uninformed strategy.
- Best-First Search
  - Greedy Best-First Search
  - A* Search
- Memory bounded Heuristic Search
  - Recursive Best-First Search
  - SMA*

Best – First Search

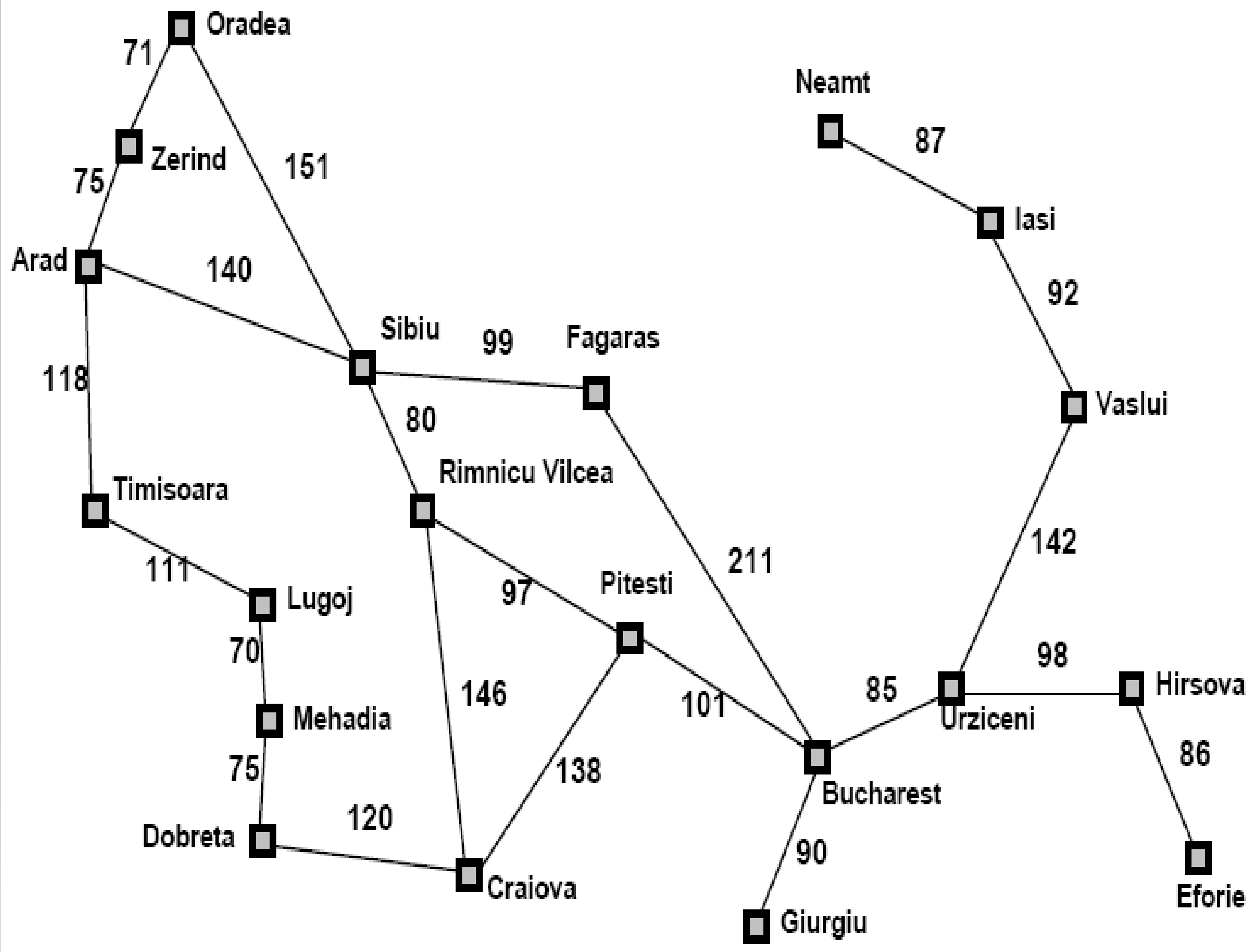
- Use an **evaluation function** *f(n)* for each node.
  - Estimate of "desirability"
- Expand most desirable unexpanded node.
- Implementation: *fringe* is a queue sorted in increasing order of desirability.
- Special cases:
  - Greedy best-First Search
  - A* Search

Heuristic Function

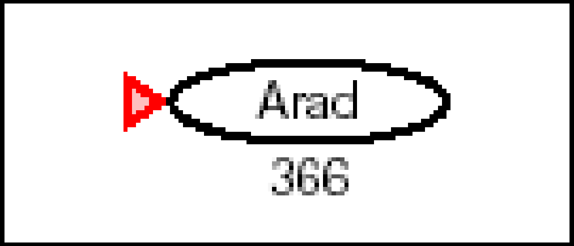
- A **heuristic function** or simply **heuristic** is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search.
- The key component of Best-first search algorithm is a **heuristic function, denoted by h(n):**
- h(n) = estimated cost of the **cheapest path from node n to a goal node.**
- **Straight line distance heuristic (h_{SLD}) is calculated based on some certain experience.**

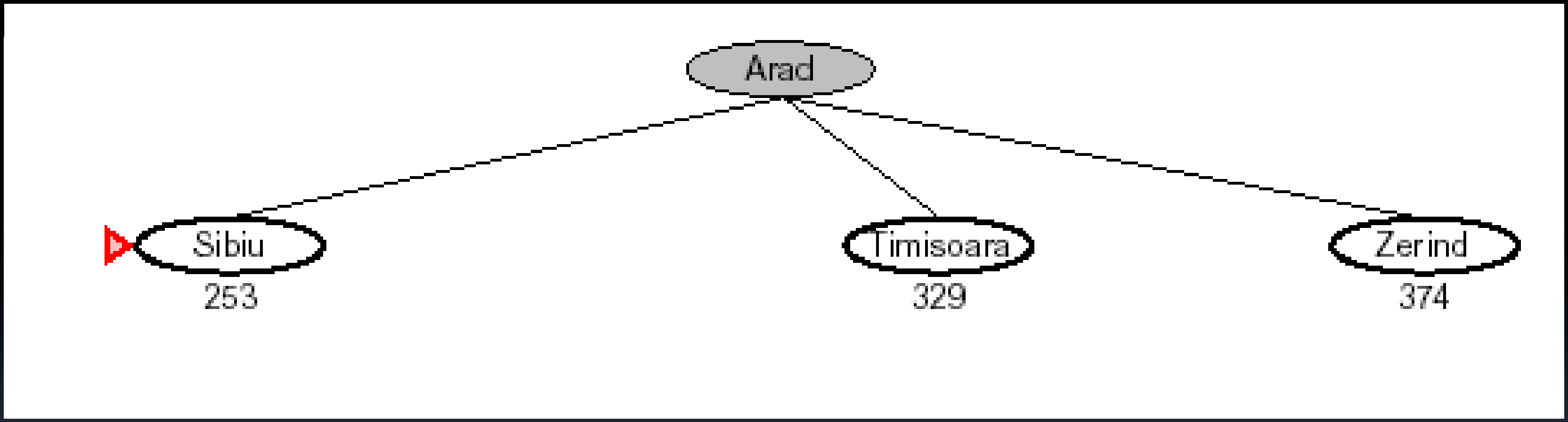
Greedy Best-First Search

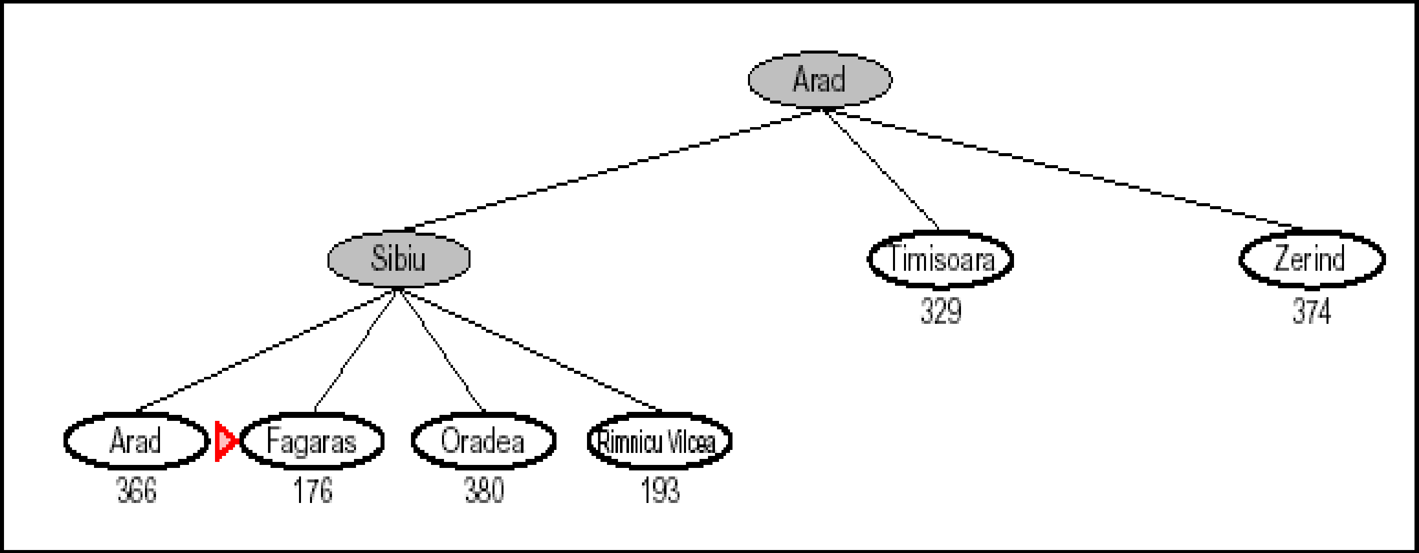
- Evaluation function *h(n)* (heuristic) = estimate of cost from *n* to the closest goal.

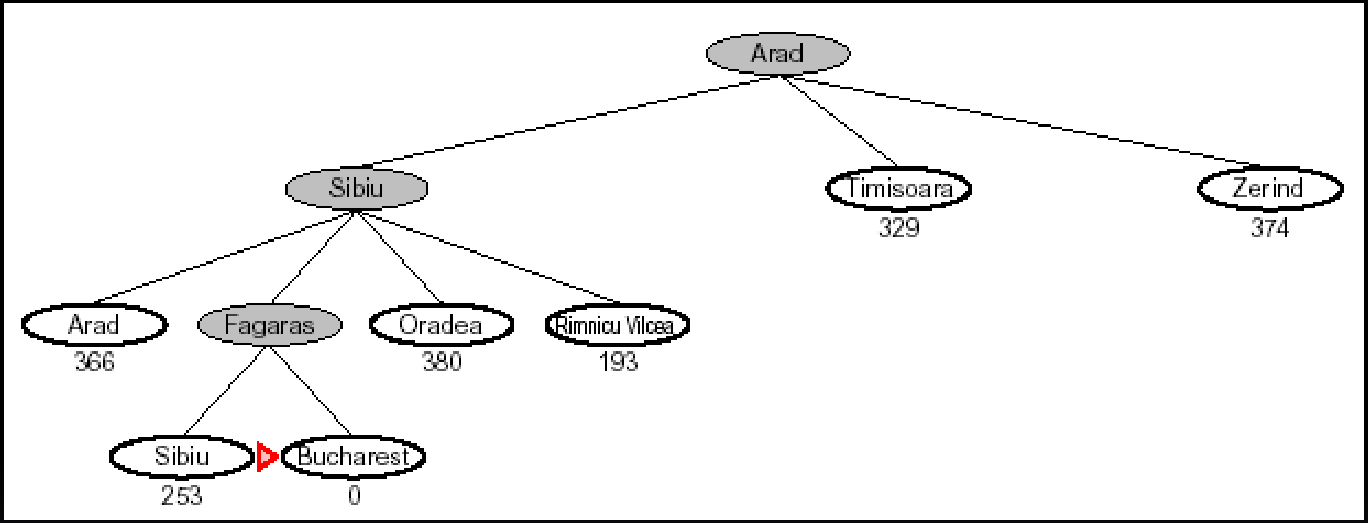


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374


Arad
366


Arad
Sibiu
253
Timisoara
329
Zerind
374


Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
176
Oradea
380
Rimnicu Vilcea
193


Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
Oradea
380
Rimnicu Vilcea
193
Sibiu
253
Bucharest
0

- Properties of greedy search
- o **Complete?? No–can get stuck in loops, e.g.,**
- Iasi ! Neamt ! Iasi ! Neamt !
- Complete in finite space with repeated-state checking
- o **Time?? O(bm), but a good heuristic can give dramatic improvement**
- o **Space?? O(bm)—keeps all nodes in memory**
- o **Optimal?? No**
- Greedy best-first search is not optimal,and it is incomplete.
- The worst-case time and space complexity is O(bm),where m is the maximum depth of the search space.

A* Search

- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost to goal from n
- $f(n)$ = estimated total cost of path through n to goal
- A* search uses an admissible heuristic
- i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n.
- (Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.)
- E.g., $h_{SLD}(n)$ never overestimates the actual road distance
- Theorem: A* search is optimal

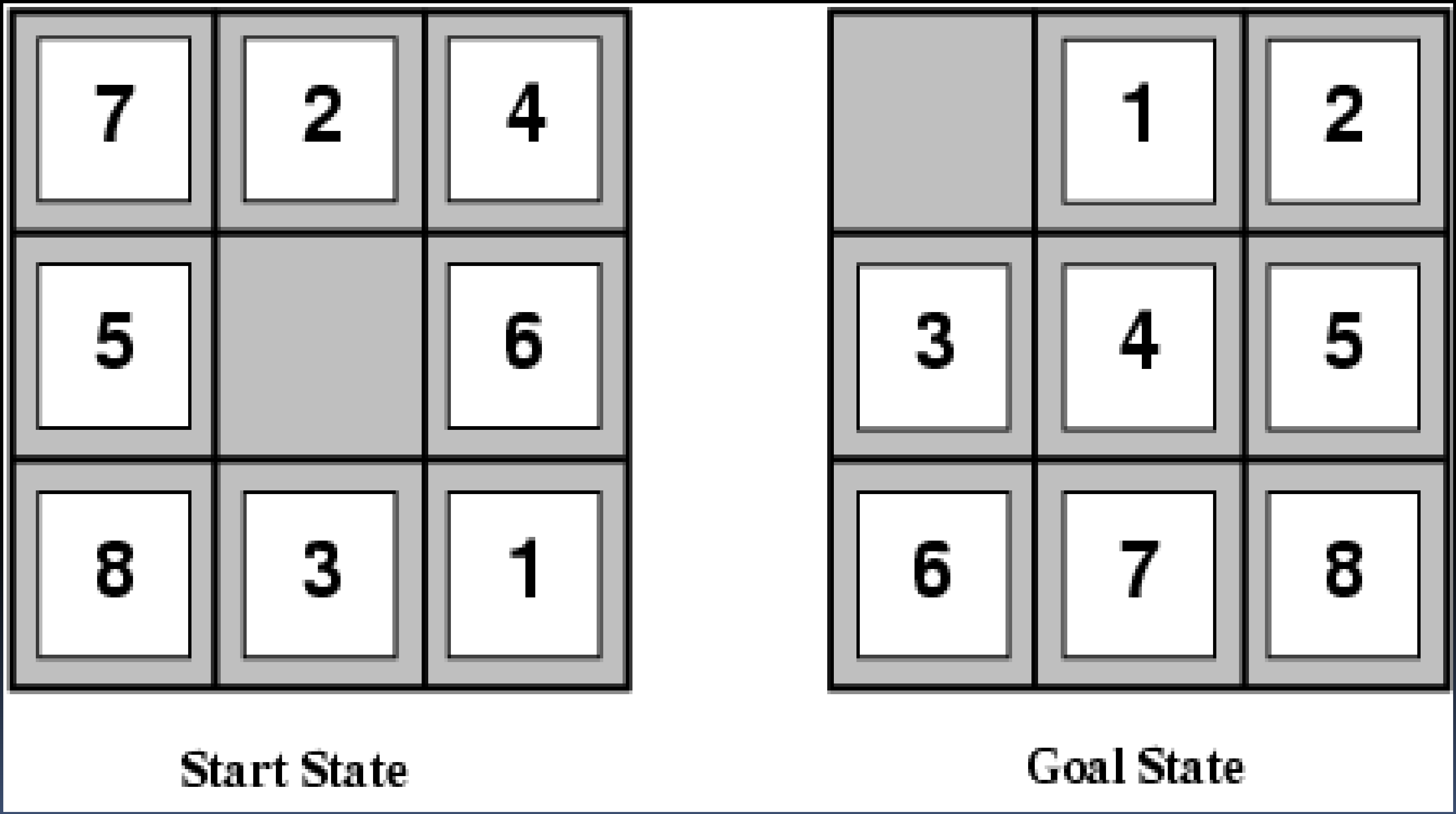
Learning to Search Better

- Meta-level state space
- Object –level state space

HEURISTIC FUNCTIONS

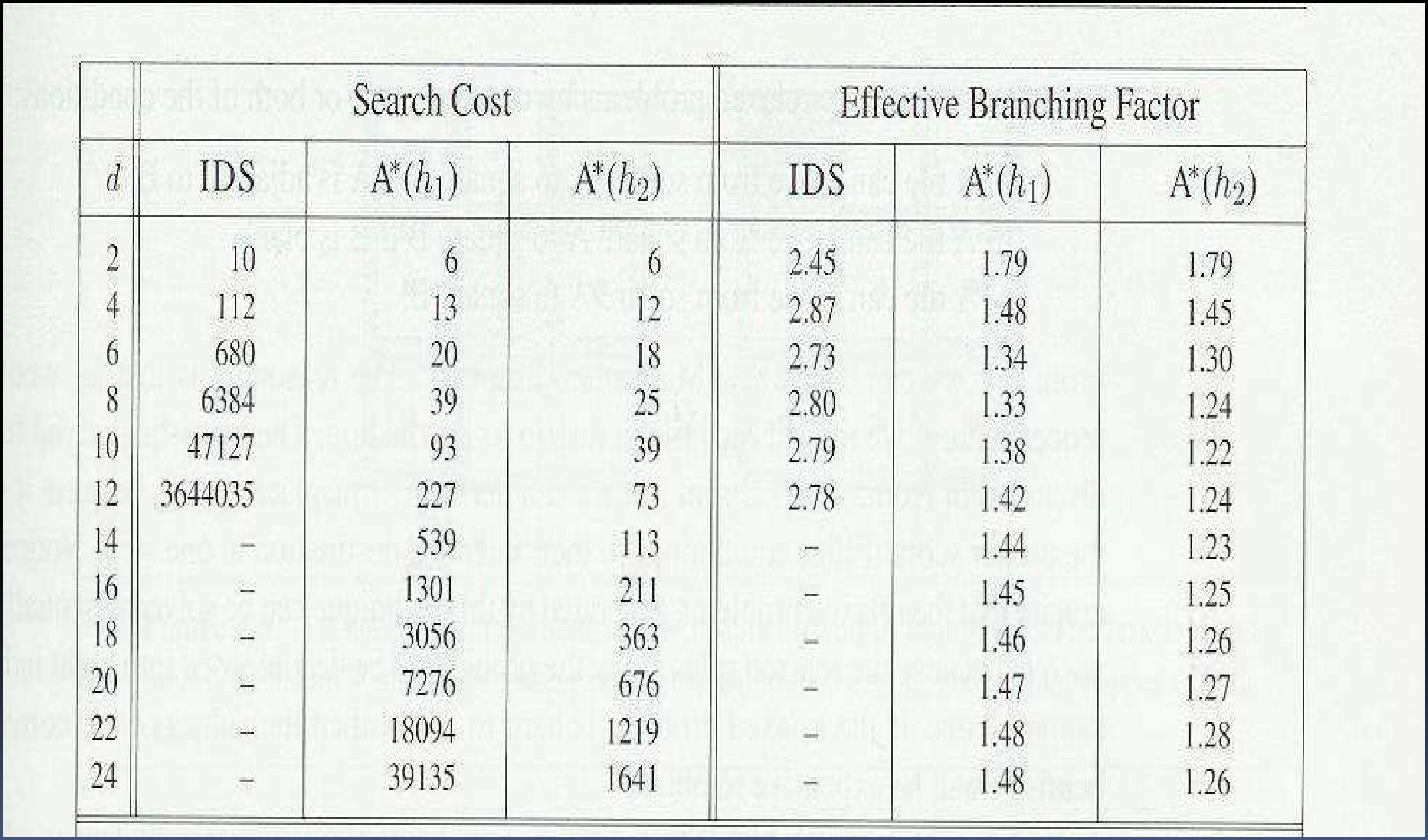
- A **heuristic function or simply a heuristic is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search.**

- 8 – puzzle problem



- Object – sliding the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration.
- Avg. soln. cost for a randomly generated instance is about 22 steps.
- Branching factor is about 3.
- An exhaustive search to depth 22 would look at about 3^{22} approximately 3.1×10^{10} states.
- By keeping track of repeated states, the above states can be reduced by a factor of about 170,000 because there are only $9!/2 = 181,440$ distinct states that are reachable.
- For 15-puzzle problem, two candidates are used:
  - h1 – number of misplaced tiles.
  - h2 – sum of the distance of tiles from their goal positions.

- The effective branching factor, b*.

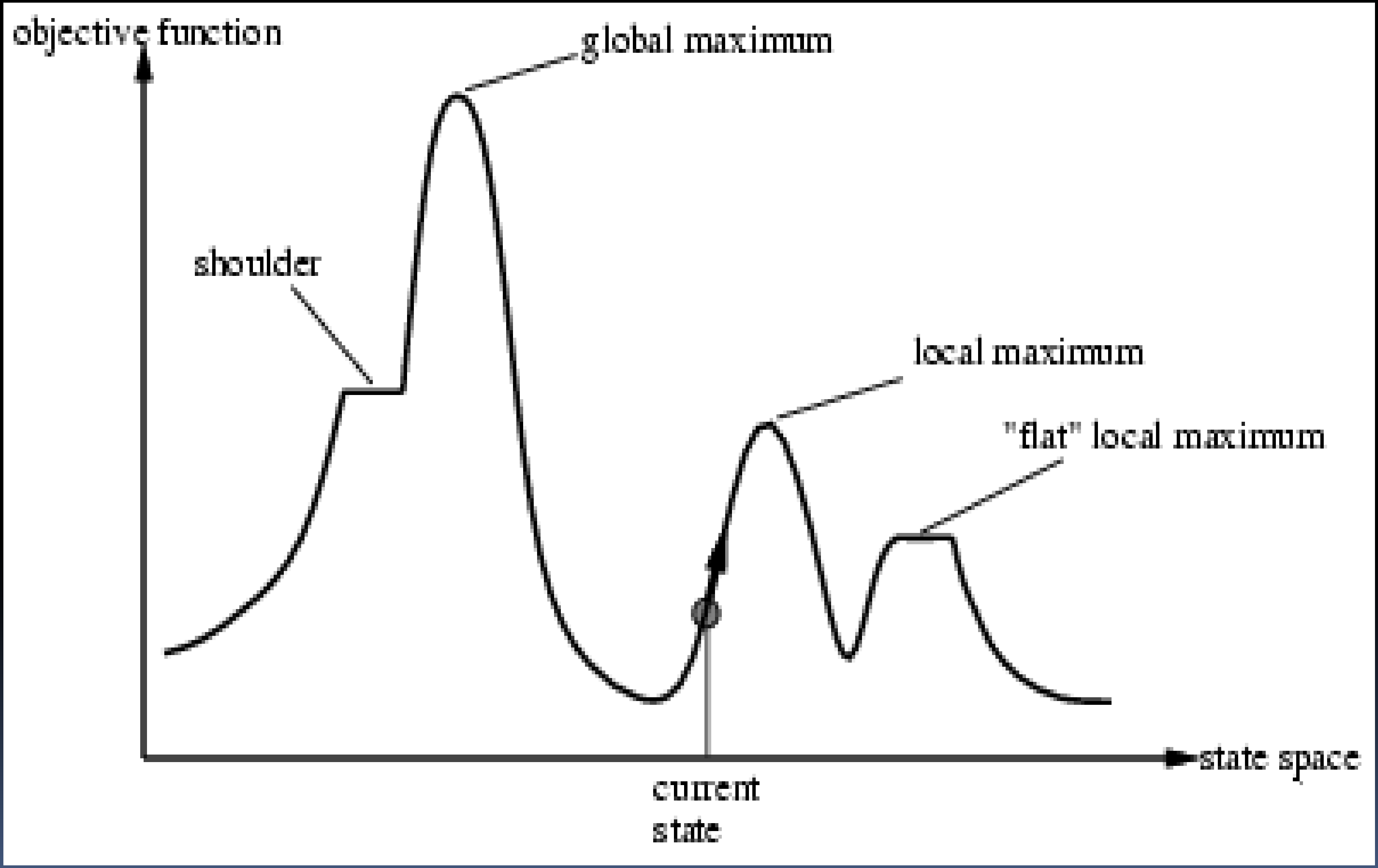


	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

- Learning heuristics from experience
  - Inductive learning algorithm
- Inventing admissible heuristic functions:
  - Relaxed problems
  - Pattern databases
  - Disjoint databases

LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

- **Local search algorithms** operate using a single current state rather than multiple paths and generally move only to neighbors of that state.
- Local search algorithms are useful for solving pure **optimization problems,** in which the aim is to find the best state according to an **objective function.**
- Advantages:
  - They use very little memory.
  - They can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

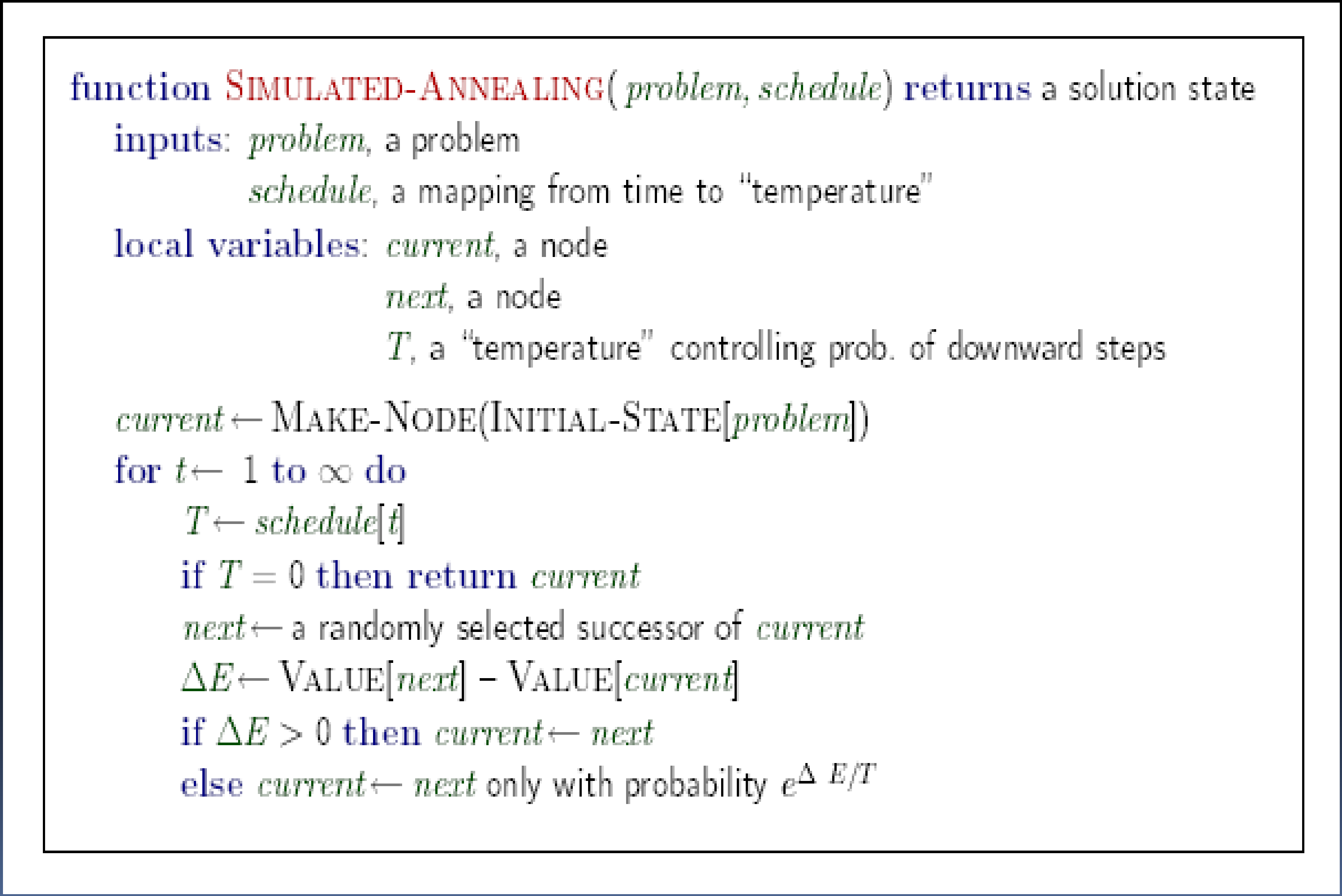

objective function
global maximum
shoulder
local maximum
"flat" local maximum
state space
current state

Hill- Climbing Search

```
function HILL-CLIMBING( problem) return a state
    that is a local maximum
    input: problem, a problem
    local variables: current, a node.
                     neighbor, a node.
    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest valued successor of current
        if VALUE [neighbor] ≤ VALUE[current] then
                 return STATE[current]
        current ← neighbor
```

- Problems with hill-climbing:
  - Local maxima / Foot hills
  - Ridges
  - Plateaux / Shoulder
- Variants of hill-climbing:
  - Stochastic hill climbing
  - First choice hill climbing
  - Random – restart hill climbing

Simulated Annealing Search



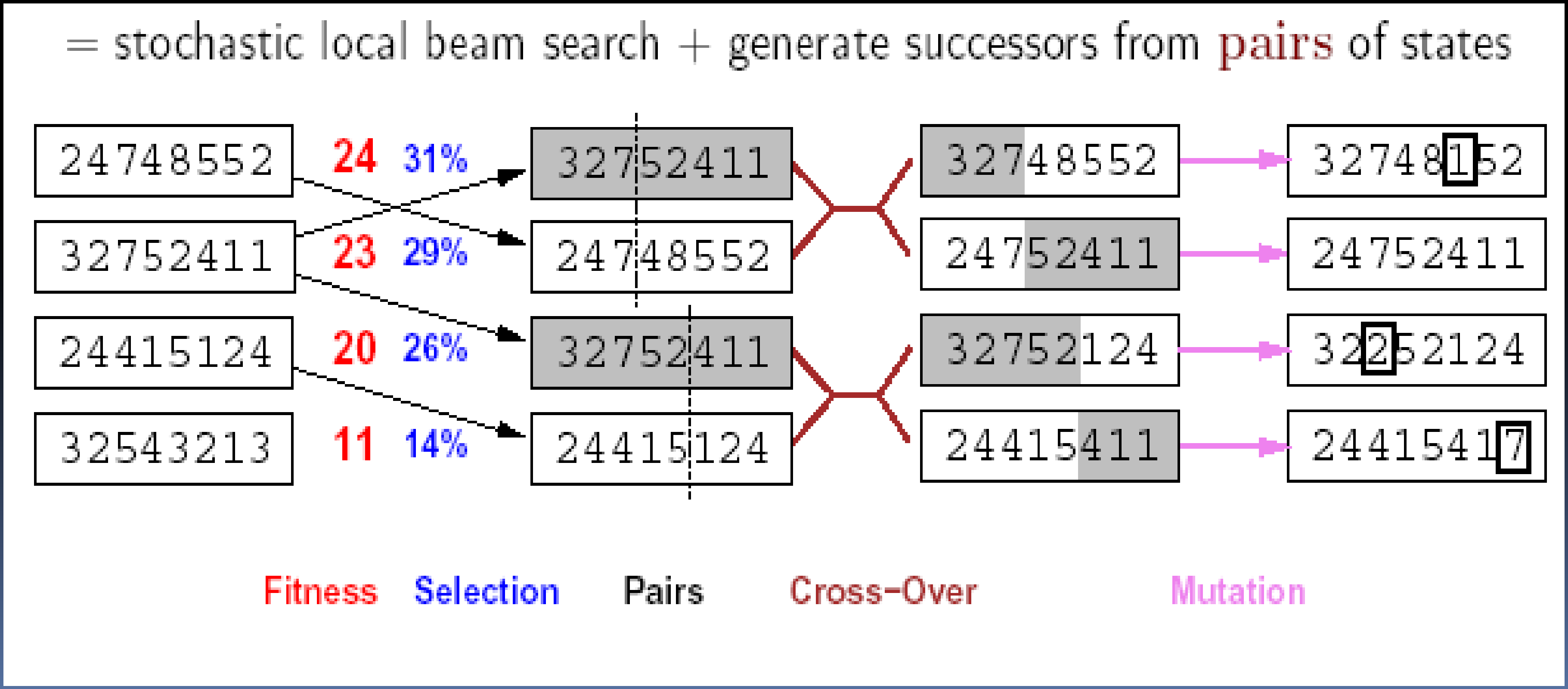
```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     $T$, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for $t$ ← 1 to $\infty$ do
        $T$ ← schedule[$t$]
        if $T = 0$ then return current
        next ← a randomly selected successor of current
        $\Delta E$ ← VALUE[next] – VALUE[current]
        if $\Delta E > 0$ then current ← next
        else current ← next only with probability $e^{\Delta E/T}$
```

Local Beam Search

- Idea: keep k states instead of 1; choose top k of all their successors
- Not the same as k searches run in parallel!
- Searches that nd good states recruit other searches to join them
- Problem: quite often, all k states end up on same local hill
- Idea: choose k successors randomly, biased towards good ones
- Observe the close analogy to natural selection!

Genetic Algorithm

- A **Genetic algorithm** is a variant of stochastic beam search in which successor states are generated by combining two parent states, rather than by modifying a single state.



```
function GENETIC_ALGORITHM( population, FITNESS-FN) return
        an individual
    input: population, a set of individuals
            FITNESS-FN, a function which determines the quality of
                    the individual
    repeat
        new_population ← empty set
        loop for i from 1 to SIZE(population) do
            x ← RANDOM_SELECTION(population, FITNESS_FN)
            y ← RANDOM_SELECTION(population, FITNESS_FN)
            child ← REPRODUCE(x,y)
            if (small random probability) then child  MUTATE(child )
            add child to new_population
        population ← new_population
until some individual is fit enough, or enough time has elapsed
return the best individual  in population, according to FITNESS-FN
```

LOCAL SEARCH IN CONTINUOUS SPACE

- Suppose we want to site three airports in Romania:
  - 6-D state space defined by (x1, y2), (x2, y2), (x3, y3)
  - objective function f(x1, y2, x2, y2, x3, y3) = sum of squared distances from each city to nearest airport

- Discretization methods
- Gradient methods
- Steepest-ascent hill climbing
- Empirical gradient
- Line search
- Newton-Raphson method

ONLINE SEARCH AGENTS AND UNKNOWN ENVIRONMENTS

- Offline search (all algorithms so far)
  - Compute complete solution, ignoring environment carry out action sequence.
- Online search
  - Interleave computation and action
  - Act – Observe - Compute
- Online search agents suits well for the following domains:
  - For dynamic, semi-dynamic, stochastic domains
  - Whenever offline search would yield exponentially many contigencies

- Online search necessary for exploration problem:
  - States and actions unknown to agent
  - Agent uses actions as experiments to determine what to do

  Eg: Robot exploring unknown building

  A new born baby
- The following terms are known by the agent to solve the problem in the given environment:
  - ACTIONS (s)
  - C(s, a, s')
  - GOAL-TEST(s)

- Competitive ratio compares actual cost with cost agent would follow if it knew the search space
- No agent can avoid dead ends in all state spaces
  - Robotics examples: Staircase, ramp, cliff, terrain
- Assume state space is safely explorable - some goal state is always reachable

Online Search Agents

- Interleaving planning and acting hamstrings offline search
  - A* expands arbitrary nodes without waiting for outcome of action Online algorithm can expand only the node it physically occupies Best to explore nodes in physically local order
  - Suggests using depth-first search
  - Next node always a child of the current
- When all actions have been tried, can't just drop state Agent must physically backtrack

- Online Depth-First Search
  - May have arbitrarily bad competitive ratio (wandering past goal) Okay for exploration; bad for minimizing path cost
- Online Iterative-Deepening Search
  - Competitive ratio stays small for state space a uniform tree

Online Local Search

- Hill Climbing Search
  - Also has physical locality in node expansions Is, in fact, already an online search algorithm
  - Local maxima problematic: can't randomly transport agent to new state in effort to escape local maximum.
- Random Walk as alternative
  - Select action at random from current state
  - Will eventually find a goal node in a finite space
  - Can be very slow, esp. if "backward" steps as common as "forward"

- Hill Climbing with Memory instead of randomness
  - Store “current best estimate” of cost to goal at each visited state Starting estimate is just h(s)
  - Augment estimate based on experience in the state space Tends to “flatten out” local minima, allowing progress Employ optimism under uncertainty
  - Untried actions assumed to have least-possible cost Encourage exploration of untried paths

Learning in Online Search

- Rampant ignorance a ripe opportunity for learning Agent learns a ―mapǁ of the environment
- Outcome of each action in each state
- Local search agents improve evaluation function accuracy
- Update estimate of value at each visited state
- Would like to infer higher-level domain model
- Example: ―Upǁ in maze search increases y -coordinate Requires
- Formal way to represent and manipulate such general rules (so far, have hidden rules within the successor function)
- Algorithms that can construct general rules based on observations of the effect of actions

CONSTRAINT SATISFACTION PROBLEMS (CSP)

- Formal Definition:

  A Constraint Satisfaction Problem(or CSP) is defined by a set of variables ,X_1,X_2,....X_n, and a set of constraints C_1,C_2,...,C_m. Each variable X_i has a nonempty domain D, of possible values. Each constraint C_i involves some subset of variables and specifies the allowable combinations of values for that subset.

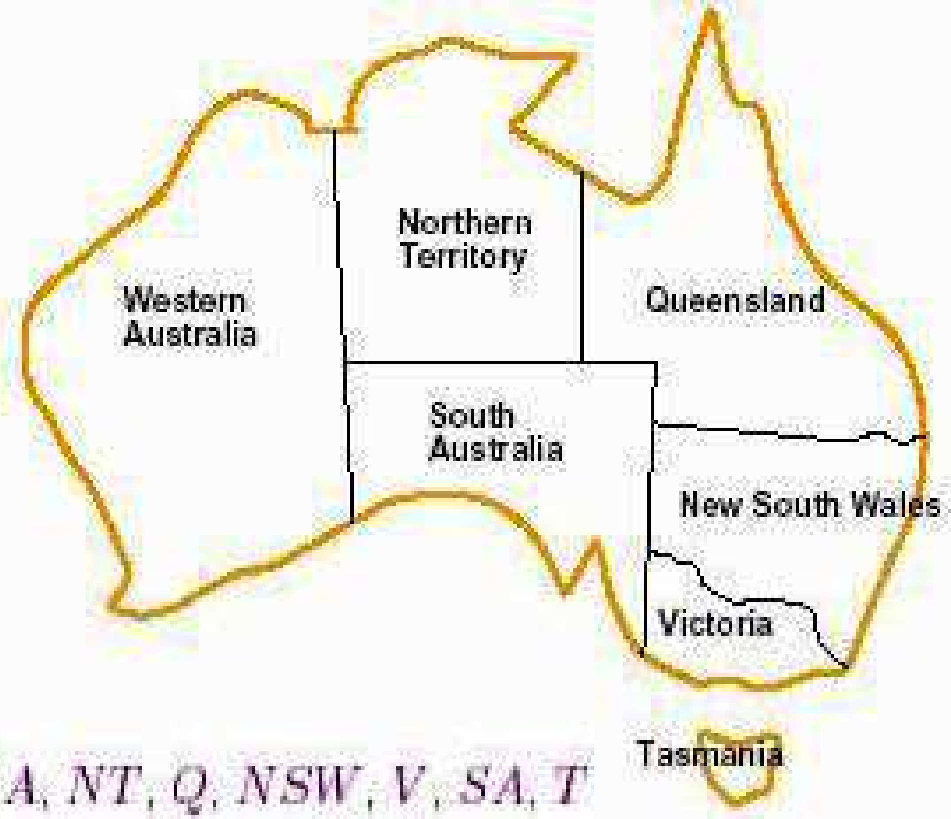
  A State of the problem is defined by an assignment of values to some or all of the variables,{$X_i = v_i$,$X_j = v_j$,...}. An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned,and a solution to a CSP is a complete assignment that satisfies all the constraints.

  Some CSPs also require a solution that maximizes an objective function.

- **Example for Constraint Satisfaction Problem :**
- The map of Australia showing each of its states and territories is given below. We are given the task of coloring each region either red, green, or blue in such a way that the neighboring regions have the same color. To formulate this as CSP ,we define the variable to be the regions :WA,NT,Q,NSW,V,SA, and T. The domain of each variable is the set {red, green, blue}.The constraints require neighboring regions to have distinct colors; for example, the allowable combinations for WA and NT are the pairs

  {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}.
- The constraint can also be represented more succinctly as the inequality WA not = NT,provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions such as

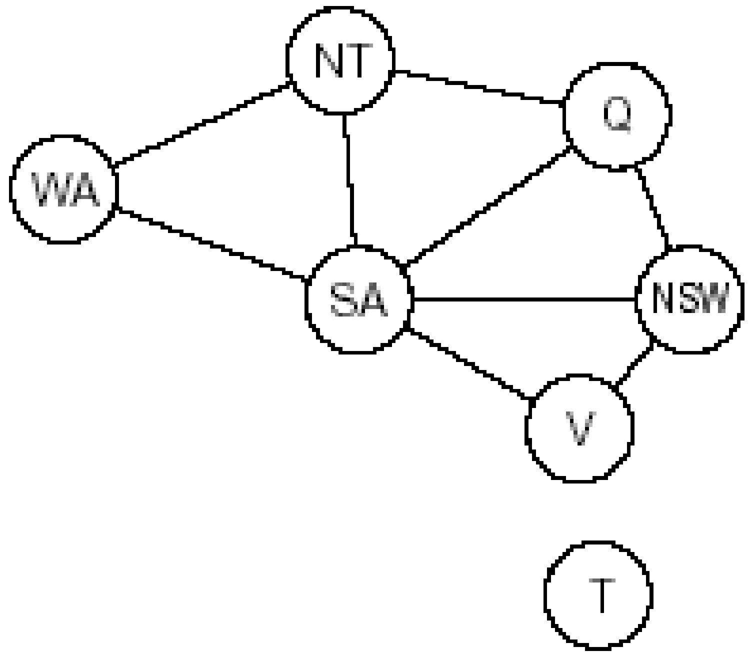
  { WA = red, NT = green, Q = red, NSW = green, V = red , SA = blue, T = red}



Variables WA, NT, Q, NSW, V, SA, T
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

- It is helpful to visualize a CSP as a constraint graph. The nodes of the graph corresponds to variables of the problem and the arcs correspond to constraints.

Constraint graph: nodes are variables, arcs show constraints



- CSP can be viewed as a standard search problem as follows :
  - **Initial state : the empty assignment {},in which all variables are unassigned.**
  - **Successor function : a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.**
  - **Goal test : the current assignment is complete.**
  - **Path cost : a constant cost(E.g.,1) for every step.**
- Every solution must be a complete assignment and therefore appears at depth n if there are n variables.
- Depth first search algorithms are popular for CSPs

- **Varieties of CSPs**

(i) Discrete variables

- **Finite domains**

  The simplest kind of CSP involves variables that are discrete and have finite domains. Map coloring problems are of this kind. The 8-queens problem can also be viewed as finite-domain CSP,where the variables Q1,Q2,…..Q8 are the positions each queen in columns 1,….8 and each variable has the domain {1,2,3,4,5,6,7,8}. If the maximum domain size of any variable in a CSP is d,then the number of possible complete assignments is O(dn) – that is,exponential in the number of variables. Finite domain CSPs include Boolean CSPs,whose variables can be either *true or false.*

- **Infinite domains**

  Discrete variables can also have infinite domains – for example,the set of integers or the set of strings. With infinite domains,it is no longer possible to describe constraints by enumerating all allowed combination of values. Instead a constraint language of algebric inequalities such as

  Startjob1 + 5 <= Startjob3.

(ii) CSPs with continuous domains

CSPs with continuous domains are very common in real world. For example ,in operation research field,the scheduling of experiments on the Hubble Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical,precedence and power constraints. The best known category of continuous-domain CSPs is that of linear programming problems,where the constraints must be linear inequalities forming a *convex region. Linear programming problems can be solved in time polynomial in the number of variables.*

- **Varieties of constraints :**

(i) unary constraints involve a single variable.

Example : SA # green

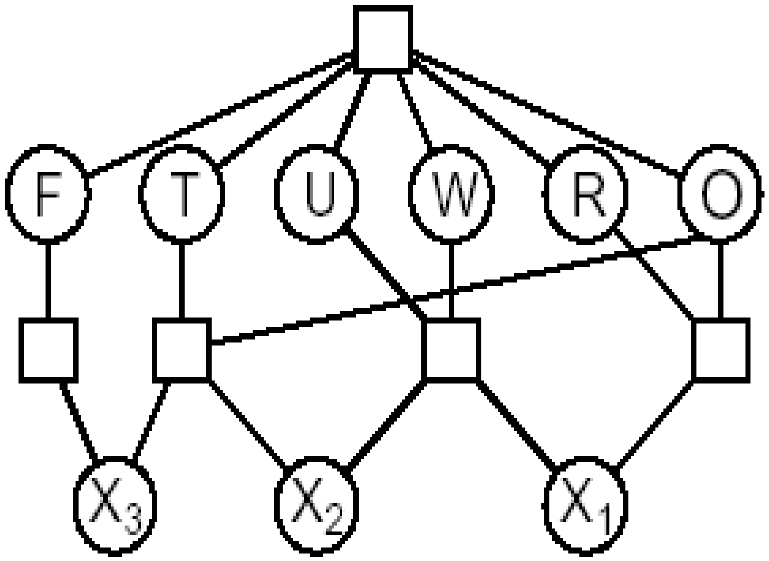
(ii) Binary constraints involve paris of variables.

Example : SA # WA

(iii) Higher order constraints involve 3 or more variables.

Example : cryptarithmetic puzzles.

$$
\begin{array}{r}
T\ W\ O \\
+\ T\ W\ O \\
\hline
F\ O\ U\ R
\end{array}
$$



Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$

Domains: $\{0,1,2,3,4,5,6,7,8,9\}$

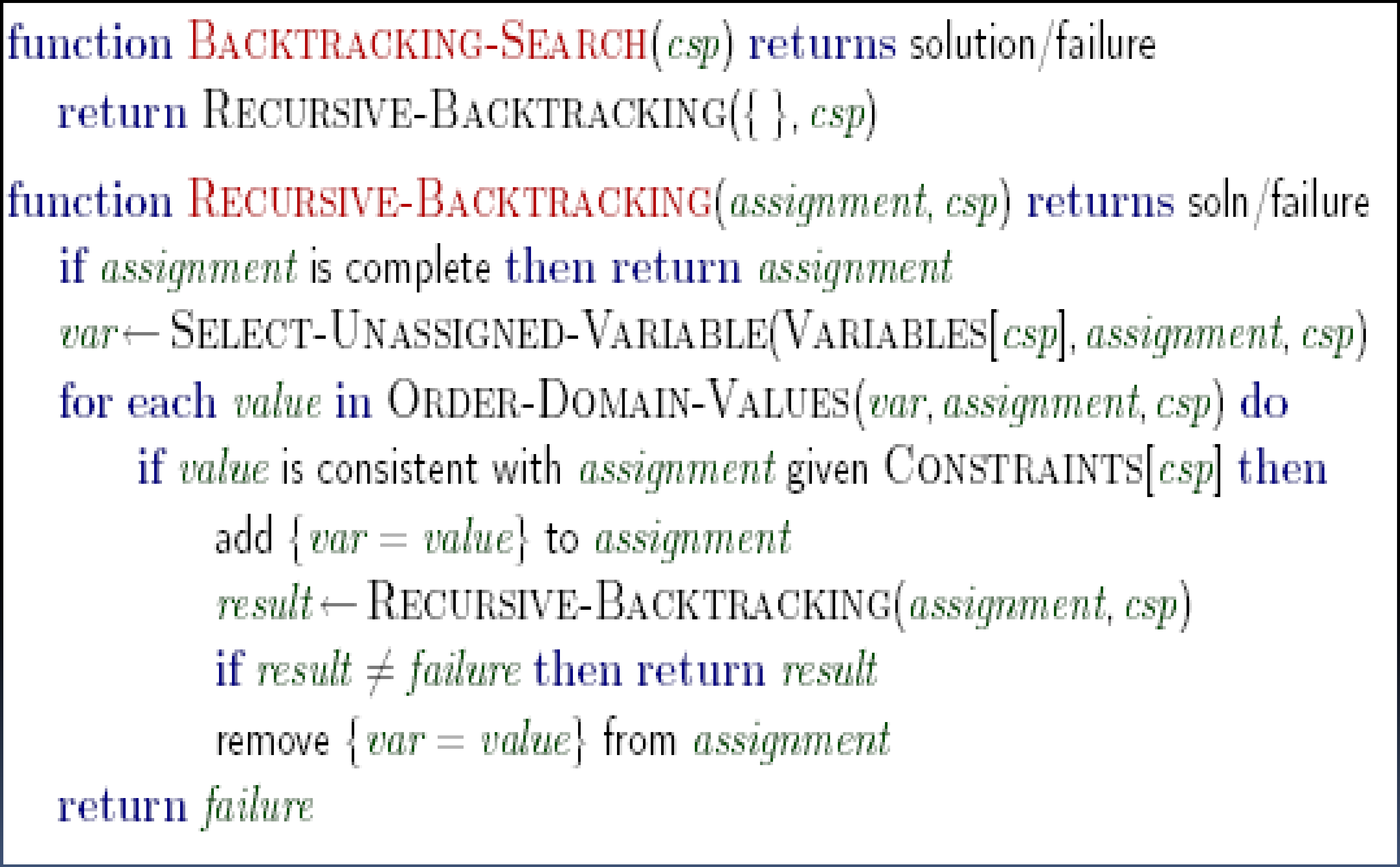
Constraints

alldiff(F,T,U,W,R,O)

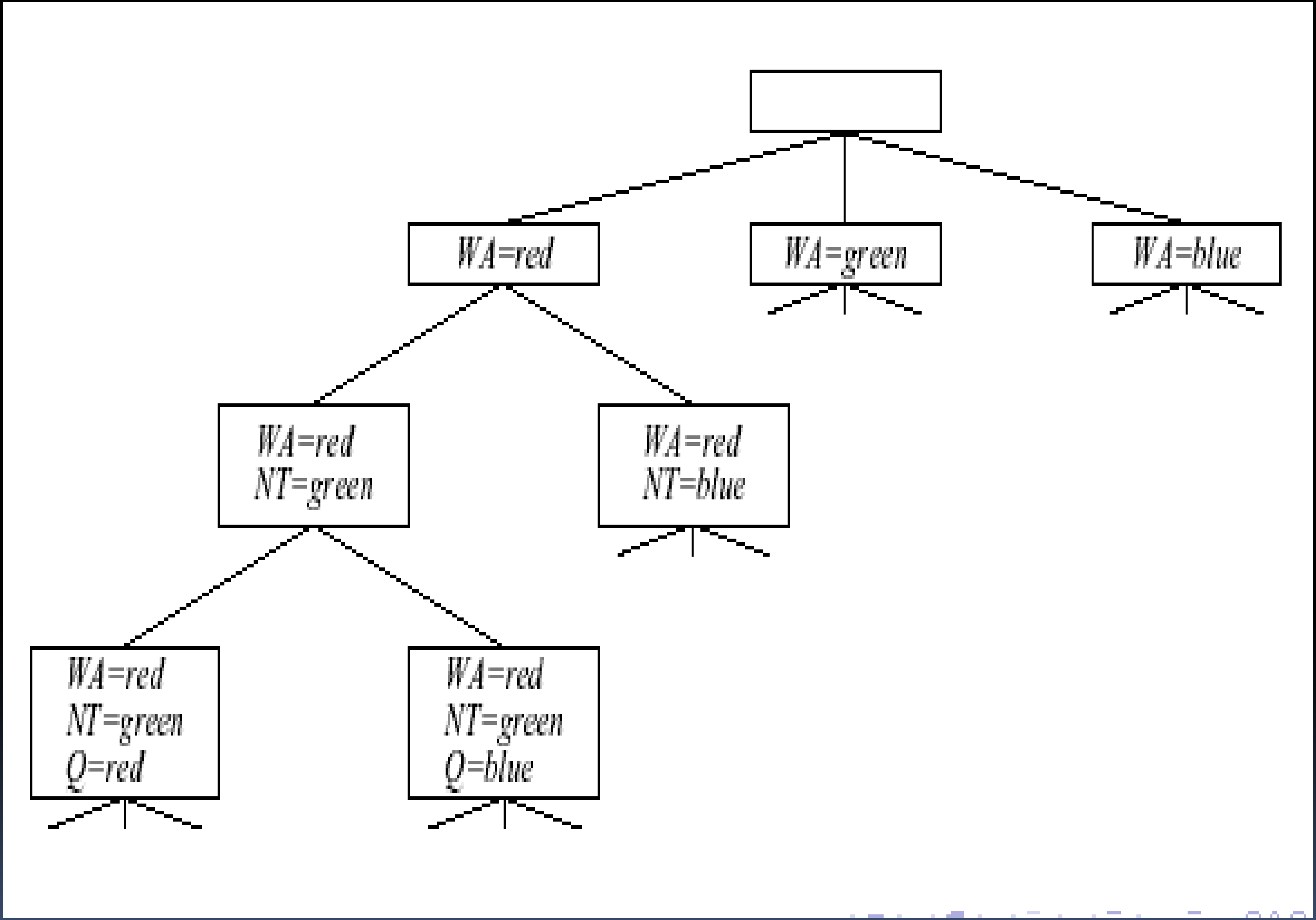
$O + O = R + 10 \cdot X_1$, etc.

Backtracking Search for CSPs

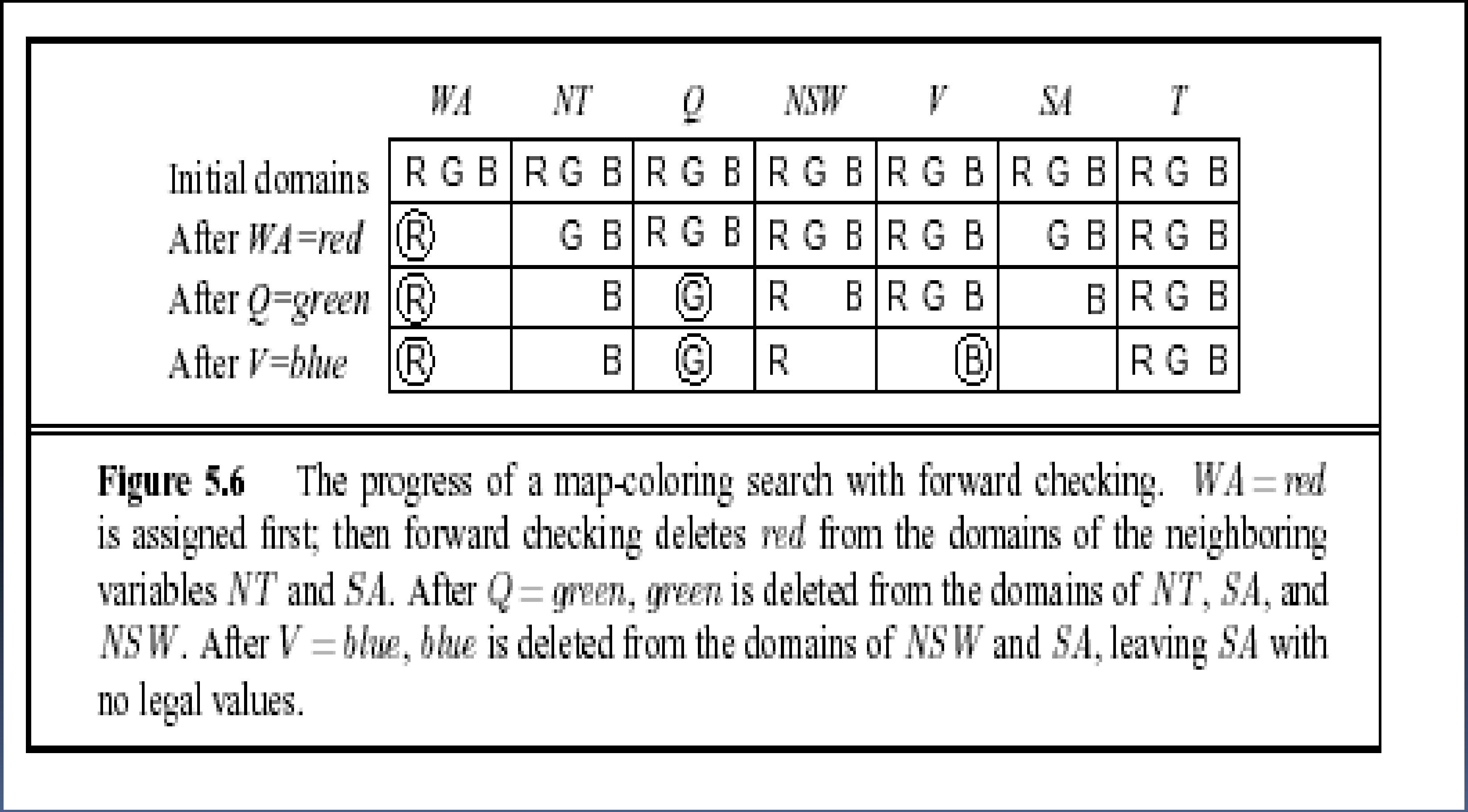
- The term **backtracking search is used for depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.**



```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result $\neq$ failure then return result
            remove {var = value} from assignment
    return failure
```


WA=red
WA=green
WA=blue
WA=red NT=green
WA=red NT=blue
WA=red NT=green Q=red
WA=red NT=green Q=blue

- Variable and Value ordering
- Propagation information through constraints
- Forward checking



	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and SA. After $Q = green$, *green* is deleted from the domains of NT, SA, and NSW. After $V = blue$, *blue* is deleted from the domains of NSW and SA, leaving SA with no legal values.

- Constraint propagation
  - Arc consistency
  - Node consistency
  - Path consistency

- Arc consistency:

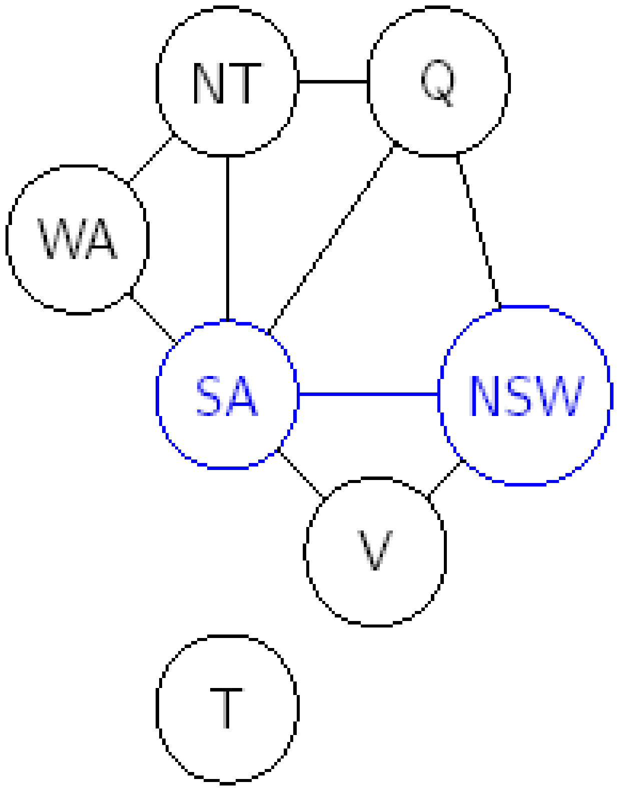
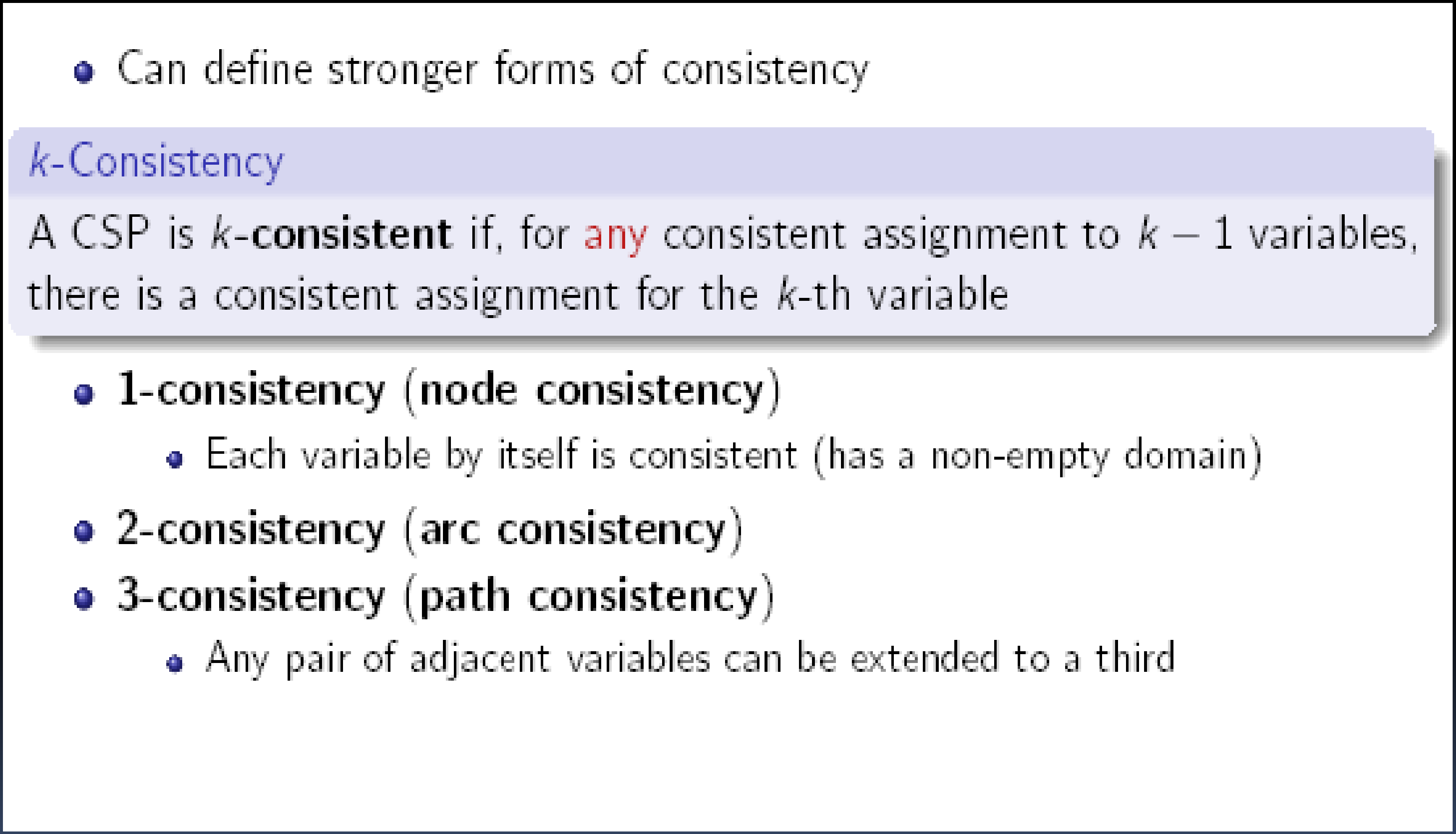


Figure: Australian Territories

- One method of constraint propagation is to enforce **arc consistency**
    - Stronger than forward checking
    - Fast
- *Arc* refers to a *directed* arc in the constraint graph
- Consider two nodes in the constraint graph (e.g., *SA* and *NSW*)
    - An arc is **consistent** if
    - For every value *x* of *SA*
    - There is some value *y* of *NSW* that is consistent with *x*
- Examine arcs for consistency in *both* directions

- K-consistency:



- Can define stronger forms of consistency

k-Consistency

A CSP is ***k*-consistent** if, for any consistent assignment to $k-1$ variables, there is a consistent assignment for the *k*-th variable

- **1-consistency** (**node consistency**)
  - Each variable by itself is consistent (has a non-empty domain)
- **2-consistency** (**arc consistency**)
- **3-consistency** (**path consistency**)
  - Any pair of adjacent variables can be extended to a third

- Handling special constraints
- Intelligent backtracking

Local Search for CSPs

- Local search algorithms good for many CSPs.
- Use complete-state formulation
    - Value assigned to every variable
    - Successor function changes one value at a time
- Choose values using **min-conflicts** heuristic
    - Value that results in the minimum number of conflicts with other variables.

The Structure of Problems

- Problem Structure
  - Consider ways in which the structure of the problem's constraint graph can help find solutions.
  - Real-world problems require decomposition into subproblems.

- Independent Subproblems:

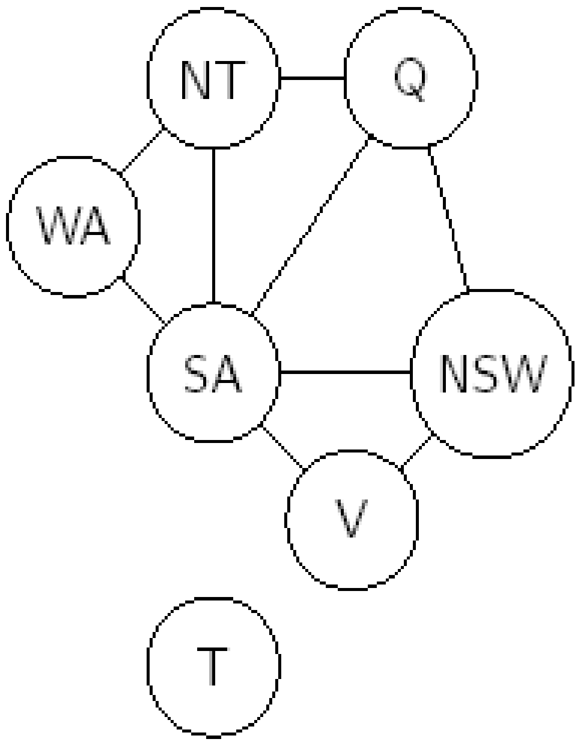


Figure: Australian Territories

- *T* is not connected
- Coloring *T* and coloring remaining nodes are **independent subproblems**
- *Any* solution for *T* combined with *any* solution for remaining nodes solves the problem
- Independent subproblems correspond to **connected components** of the constraint graph
- Sadly, such problems are rare

- Tree – Structured CSPs:

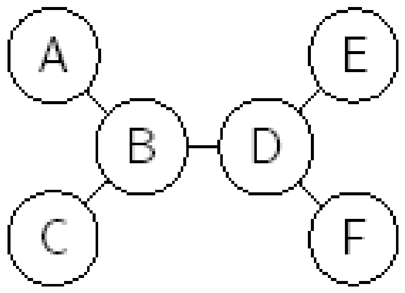


Figure: Tree-Structured CSP

- In most cases, CSPs are connected
- A simple case is when the constraint graph is a **tree**
- Can be solved in time linear in the number of variables
  - Order variables so that each parent precedes its children
  - Working "backward," apply arc consistency between child and parent
  - Working "forward," assign values consistent with parent

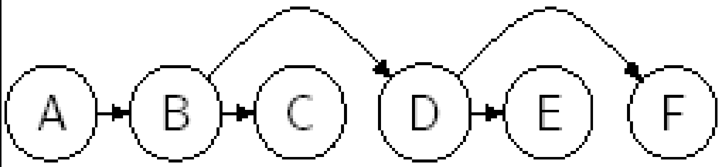


Figure: Linear ordering

•Removing nodes – Cutset conditioning
•Collapsing nodes together – Tree decomposition