Problem Solving by Searching

Chapter_3

Problem Solving in AI

- Problem solving involves finding a sequence of actions that lead from an **initial state** to a **goal state**.
- AI systems use **search** as a core strategy to solve problems where the solution is not immediately known.
- Search helps **navigate through possible states** (configurations) to find a desired goal state.
- Example: Navigating a robot through a maze, solving puzzles, playing chess.

What is State Space?

- **State**: A specific configuration or situation in the problem domain.
- **State Space**: The set of all states reachable from the initial state by applying a series of actions.
- **State Space Search** involves exploring the state space systematically to find a path from initial state to goal state.

Elements of a Search Problem

Element	Description
Initial State	The starting point of the problem
Goal Test	Method to determine if the current state is the goal
Successor Function	Defines available actions & resulting new states
Path Cost	Cost of reaching a state (optional for optimization problems)
Operators/Actions:	The possible moves or transformations that can be applied to move from one state to another within the search space
Search Strategy	to explore the search space and find a solution

Types of Search Strategies

Uninformed (Blind) Search Strategies

1. **Breadth-First Search (BFS)**: Explores all nodes at the present depth level before moving on to nodes at the next depth level. BFS guarantees finding the shortest path in terms of the number of actions if all actions have the same cost.
2. **Depth-First Search (DFS)**: Explores as far down a branch as possible before backtracking. It uses less memory than BFS but may get stuck in deep or infinite branches if not implemented with precautions like depth limits.
3. **Depth-Limited Search (DLS)**: Similar to DFS but with a limit on the depth to prevent infinite loops.

Types of Search Strategies

Informed (Heuristic) Search Strategies

1. **Greedy Best-First Search**: Uses a heuristic to estimate the cost from the current state to the goal, expanding the most promising node first. It's faster but doesn't guarantee the shortest path.
2. **A* Search**: Combines the cost to reach the current node and the heuristic estimate to the goal (f(n) = g(n) + h(n)), providing an optimal solution if the heuristic is admissible (never overestimates the true cost).
3. **Hill Climbing**: Continually moves towards the goal based on the heuristic, but can get stuck in local maxima or plateaus.
4. **Simulated Annealing**: Similar to hill climbing but occasionally accepts worse solutions to escape local maxima, inspired by the annealing process in metallurgy.

Problem Solving as Search

- Represented by a **Search Tree**:
  - **Root Node**: Initial state.
  - **Branches**: Actions.
  - **Child Nodes**: Resulting states.
- The AI agent explores this tree to find the goal.

Well-defined problems and solutions

A problem is defined by 5 components:

- ***Initial state***
- ***Actions***
- ***Transition model or (Successor functions)***
- ***Goal Test.***
- ***Path Cost.***

Well-defined problems and solutions

- A problem is defined by 4 components:
  - The ***initial state***
    - that the agent starts in
  - The set of possible **actions**
  - ***Transition model:*** *description of what each action does.*

  (***successor functions***): refer to any state reachable from given state by a single action
  - Initial state, actions and Transition model define the ***state space***
    - the set of all states reachable from the initial state by any sequence of actions.
  - A ***path*** in the state space:
    - any sequence of states connected by a sequence of actions.

Well-defined problems and solutions

- The ***goal test***
  - Applied to the current state to test
    - if the agent is in its goal

  -Sometimes there is an explicit set of possible goal states. (example: in Amman).

  -Sometimes the goal is described by the properties
    - instead of stating explicitly the set of states
  - Example: Chess
    - the agent wins if it can capture the KING of the opponent on next move (checkmate).
    - no matter what the opponent does

Well-defined problems and solutions

- A ***path cost*** function,
  - assigns a numeric cost to each path
  - = performance measure
  - denoted by ***g***
  - to distinguish the best path from others
- Usually the path cost is
  - the sum of the **step costs** of the individual actions (in the action list)

Well-defined problems and solutions

- Together a problem is defined by
  - Initial state
  - Actions
  - Successor function
  - Goal test
  - Path cost function
- The ***solution*** of a problem is then
  - *a path from the initial state to a state satisfying the goal test*
- ***Optimal*** solution
  - the solution with lowest path cost among all solutions

Formulating problems

- Besides the four components for problem formulation
  - anything else?
- ***Abstraction***
  - the process to take out the irrelevant information
  - leave the most essential parts to the description of the states

  (Remove detail from representation)
  - **Conclusion**: Only the most important parts *that are contributing to searching* are used

Evaluation Criteria

- formulation of a problem as search task
- basic search strategies
- important properties of search strategies
- selection of search strategies for specific tasks

(The ordering of the nodes in FRINGE defines the search strategy)

Problem-Solving Agents

- agents whose task is to solve a particular problem (steps)
  - goal formulation
    - what is the goal state
    - what are important characteristics of the goal state
    - how does the agent know that it has reached the goal
    - are there several possible goal states
      - are they equal or are some more preferable
  - problem formulation
    - what are the possible states of the world relevant for solving the problem
    - what information is accessible to the agent
    - how can the agent progress from state to state

Example 1: Water Jug Problem

- **Problem Statement:**
- Two jugs: 4-liter (Jug A) & 3-liter (Jug B).
- Goal: Measure exactly 2 liters in Jug A.
- **State Representation:**
- Each state is a pair: (x, y)
  - x = water in Jug A.
  - y = water in Jug B.
- **Initial State:**
- (0, 0) → Both jugs are empty.

Example 1: Water Jug Problem

- **Goal Test:**
- x == 2 (Jug A has exactly 2 liters).
- **Actions (Successor Function):**

1. Fill Jug A fully (x = 4).
2. Fill Jug B fully (y = 3).
3. Empty Jug A.
4. Empty Jug B.
5. Pour water from Jug A to Jug B.
6. Pour water from Jug B to Jug A.

Example 1: Water Jug Problem

- **State Space Example:**
- From (0,0):
  - Fill Jug A → (4,0)
  - Fill Jug B → (0,3)
  - etc.

Example:

WATER JUG PROBLEM

You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring mark on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?

=>

The state space for this problem can be described as the set of ordered pairs of integers (X, Y)
where,
X represents the quantity of water in the 4-gallon jug X= 0,1,2,3,4
Y represents the quantity of water in the 3-gallon jug Y=0,1,2,3

Then,

Initial State: (0,0)

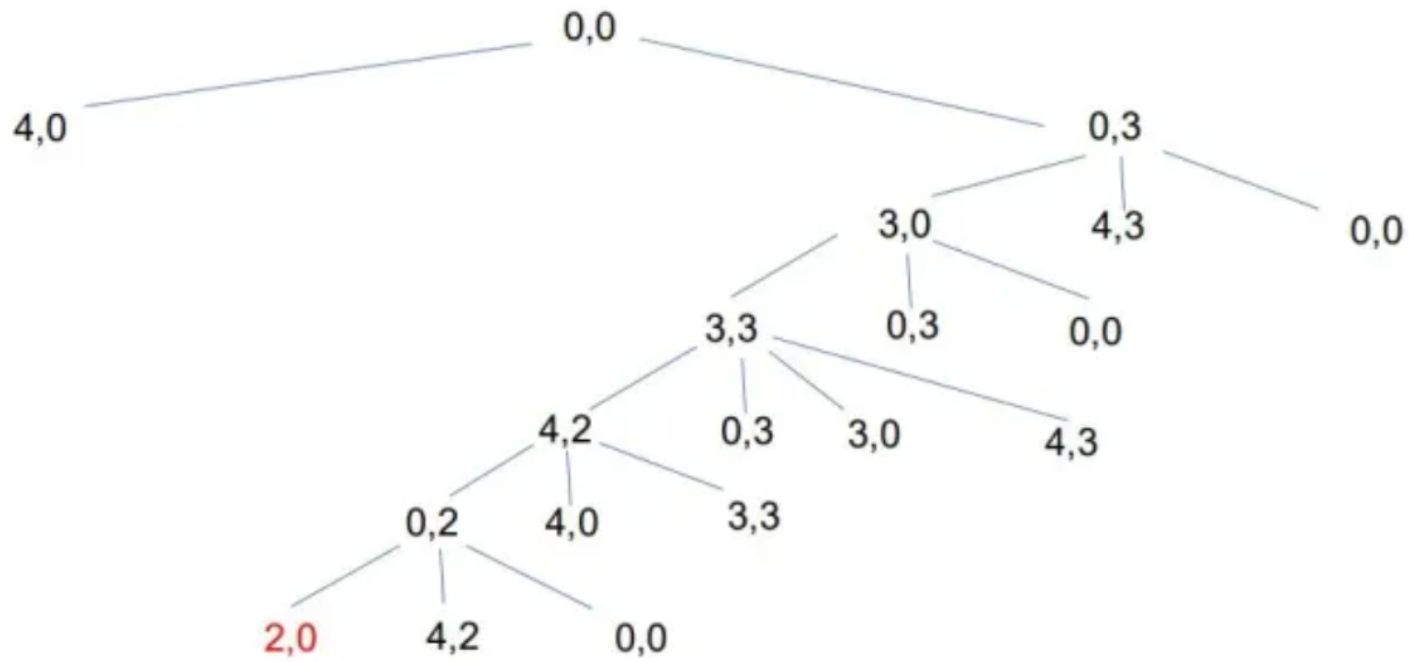
Goal State: (2,0)

Operator / Production Rule:

– Fill the 4 or 3-gallon jug.

– Empty the 4 or 3-gallon jug.

– Pour from either of the jugs to another.



State Space for Water Jug Problem

Production Rules for the Water Jug Problem in Artificial Intelligence

- To solve the water jug problem, many algorithms can be used. These include:
- Breadth-First Search: BFS or Breadth First Search visits the nodes in order of their distance from the starting node. This implies that it will visit the nearest node first.
- Depth First Search: DFS or Depth First Search visits the nodes in order of their depth.

Production Rules for the Water Jug Problem in Artificial Intelligence

- In production rules for the water jug problem, let x denote a 4-litre jug, and y denote a 3-litre jug, i.e. x=0,1,2,3,4 or y=0,1,2,3
- Start state (0,0)
- Goal state (2,n) from any n
- Start from the start state and end up at the goal state. Production rules for the water jug problem in AI are as follows:

1.	(x,y) is X<4 -> (4, Y)	Fill the 4-litre jug
2.	(x, y) if Y<3 -> (x, 3)	Fill the 3-litre jug
3.	(x, y) if x>0 -> (x-d, d)	Pour some water from a 4-litre jug
4.	(x, y) if Y>0 -> (d, y-d)	Pour some water from a 3-litre jug
5.	(x, y) if x>0 -> (0, y)	Empty 4-litre jug on the ground
6.	(x, y) if y>0 -> (x,0)	Empty 3-litre jug on the ground
7.	(x, y) if X+Y >= 4 and y>0 -> (4, y-(4-x	Pour water from a 3-litre jug into a 4-litre jug until it is full
8.	(x, y) if X+Y>=3 and x>0 -> (x-(3-y), 3))	Pour water from a 3-litre jug into a 4-litre jug until it is full
9.	(x, y) if X+Y <=4 and y>0 -> (x+y, 0	Pour all the water from a 3-litre jug into a 4-litre jug
10.	(x, y) if X+Y<=3 and x>0 -> (0, x+	Pour all the water from a 4-litre jug into a 3-litre jug
11.	(0, 2) -> (2, 0)	Pour 2-litre water from 3-litre jug into 4-litre jug
12.	(2, Y) -> (0, y)	Empty 2-litre in the 4-litre jug on the ground.

The Solution to the Water Jug Problem in Artificial Intelligence

- Here is the water jug problem solution in AI is as follows:
- Current state (0,0)
- Loop till the goal state (2,0) is reached.
- Apply a rule when the left side matches the current state
- Set the new current state to the resulting state

- Start state (0,0)
- (0,3) Apply Rule 2, Fill the 3-litre Jug
- (3,0) Apply Rule 9: Pour all the water from a 3-litre jug into a 4-litre jug
- (3,3) Apply Rule 2, Fill the 3-litre Jug
- (4,2) Apply Rule 7: Pour water from a 3-litre jug into a 4-litre jug until it is full
- (0,2) Apply Rule 5, Empty 4-litre jug on the ground
- (2,0) Apply Rule 9: Pour all the water from a 3-litre jug into a 4-litre jug

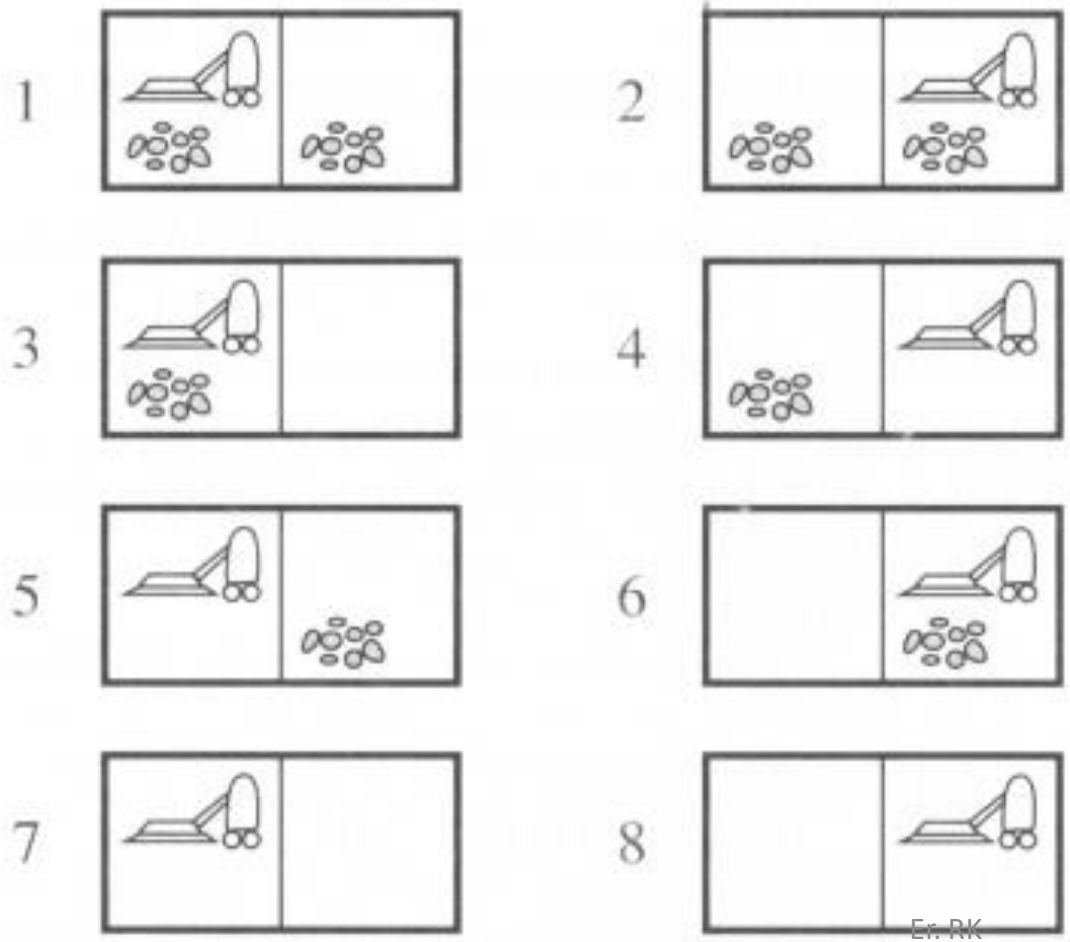
- Another water jug problem solution is:
- (0, 0) – Start State
- (4, 0) – Rule 1: Fill the 4-litre jug
- (1, 3) – Rule 8: Pour water from the 4-litre jug into the 3-litre jug until the 3-litre jug is full.
- (1, 0) – Rule 6: Empty the 3-litre jug on the ground
- (0, 1) – Rule 10: Pour all the water from the 4-litre jug into the 3-litre jug.
- (4, 1) – Rule 1: Fill the 4-litre jug
- .(2, 3) – Rule 8: Pour water from the 4-litre jug into the 3-litre jug until the 3-litre jug is full.
- Goal State reached

Example problems

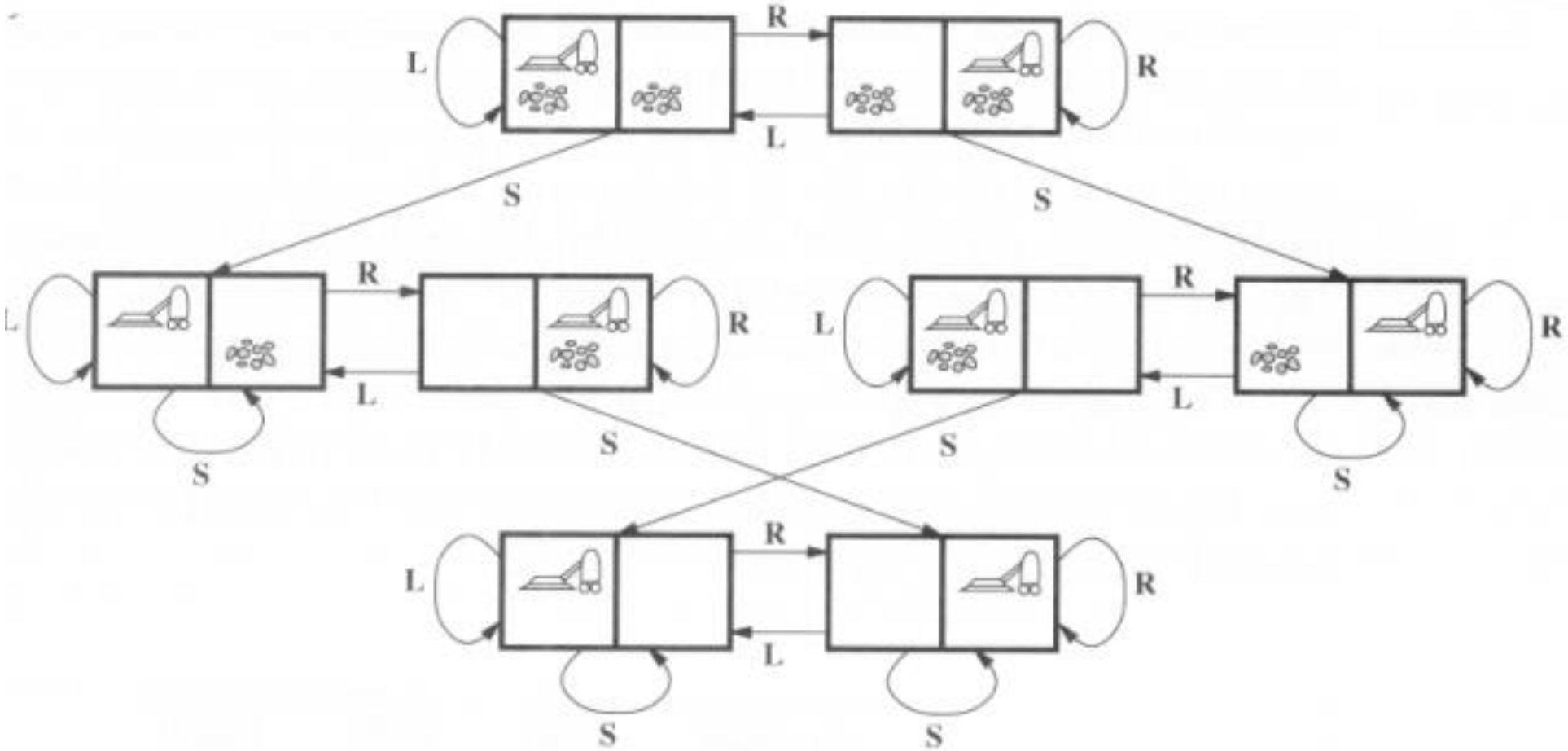
- *Toy problems*
  - those intended to illustrate or exercise various problem-solving methods
  - E.g., puzzle, chess, etc.
- *Real-world problems*
  - tend to be more difficult and whose solutions people actually care about
  - E.g., Design, planning, etc.

Toy problems

- Example: vacuum world

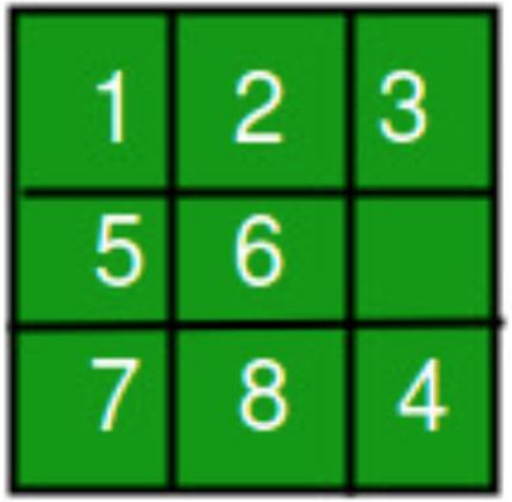


- Number of states: 8
- Initial state: Any
- Number of actions: 4
  - *left, right, suck, noOp*
- Goal: clean up all dirt
  - Goal states: {7, 8}
- Path Cost:
  - Each step costs 1

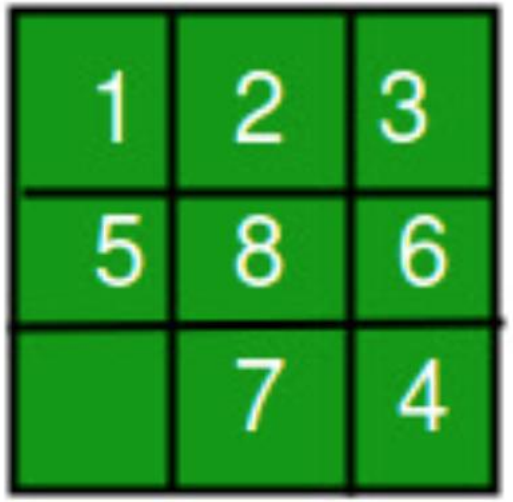

L
R
L
R
S
S
L
R
L
R
S
S
L
R
L
R
S
S
L
R
L
R
S
S

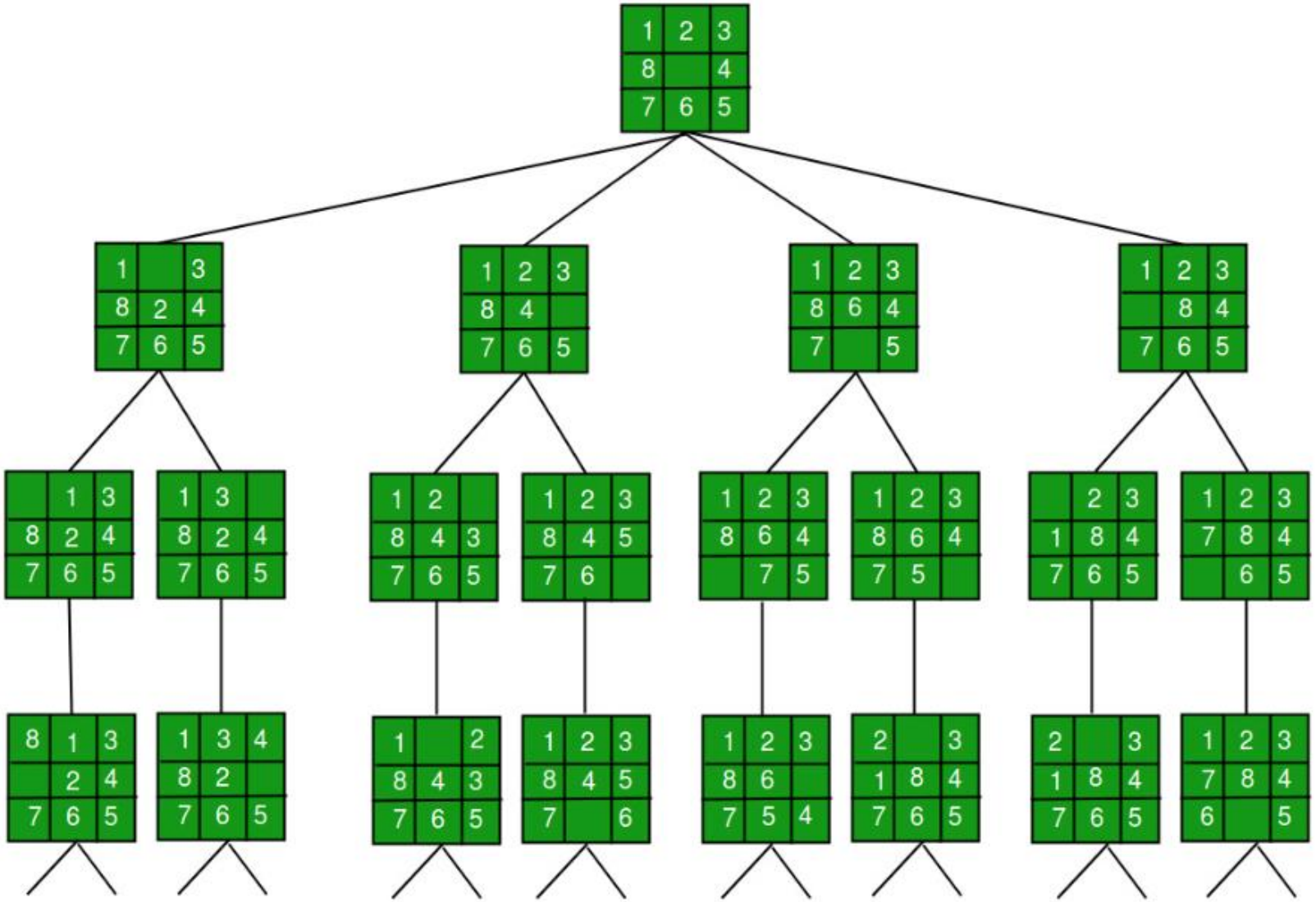
The 8-puzzle

Initial configuration

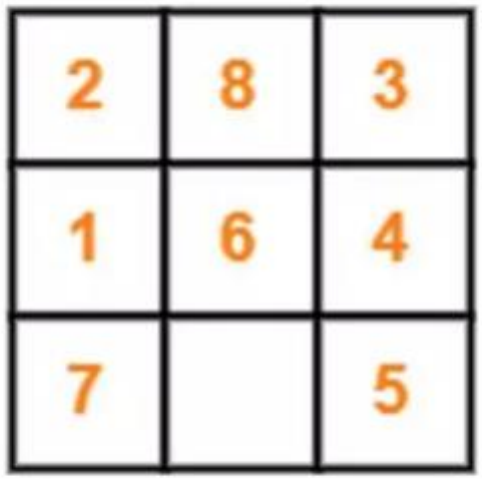


Final configuration

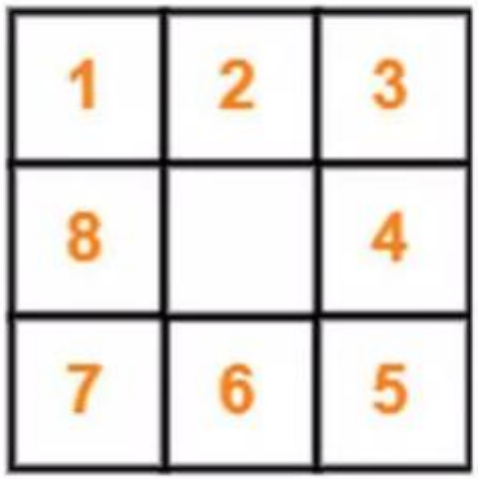



1 2 3
8 4
7 6 5
1 3
8 2 4
7 6 5
1 2 3
8 4
7 6 5
1 2 3
8 6 4
7 5
1 2 3
8 4
7 6 5
1 3
8 2 4
7 6 5
1 3
8 2 4
7 6 5
1 2
8 4 3
7 6 5
1 2 3
8 4 5
7 6
1 2 3
8 6 4
7 5
1 2 3
8 6 4
7 5
2 3
1 8 4
7 6 5
1 2 3
7 8 4
6 5
8 1 3
2 4
7 6 5
1 3 4
8 2
7 6 5
1 2
8 4 3
7 6 5
1 2 3
8 4 5
7 6
1 2 3
8 6
7 5 4
2 3
1 8 4
7 6 5
2 3
1 8 4
7 6 5
1 2 3
7 8 4
6 5

Given an initial state of a 8-puzzle problem and final state to be reached-

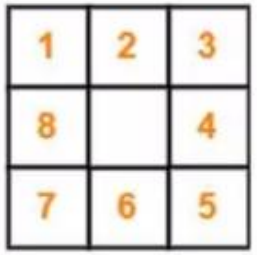


Initial State



Final State

Initial State



Final State

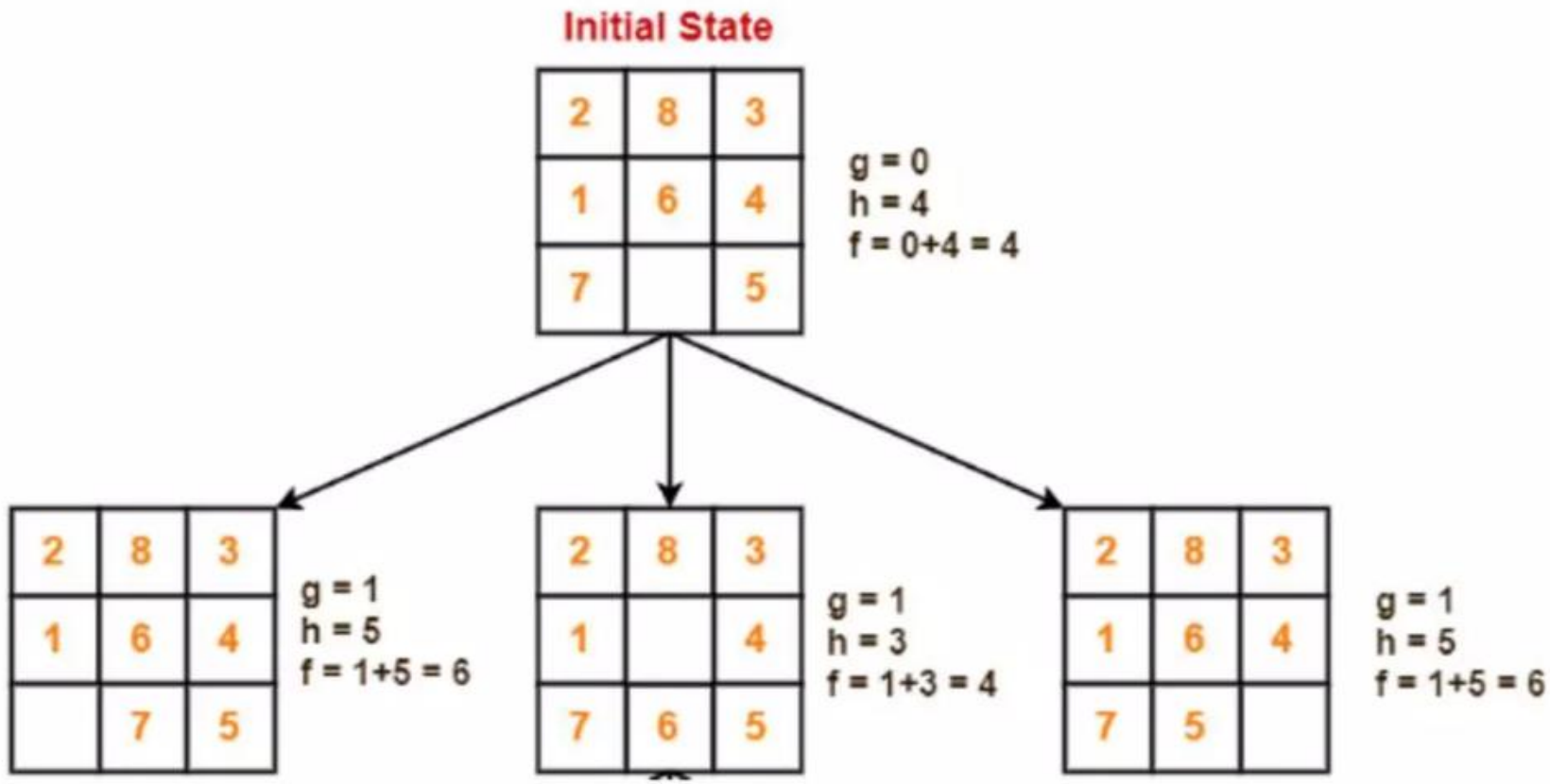
Initial State

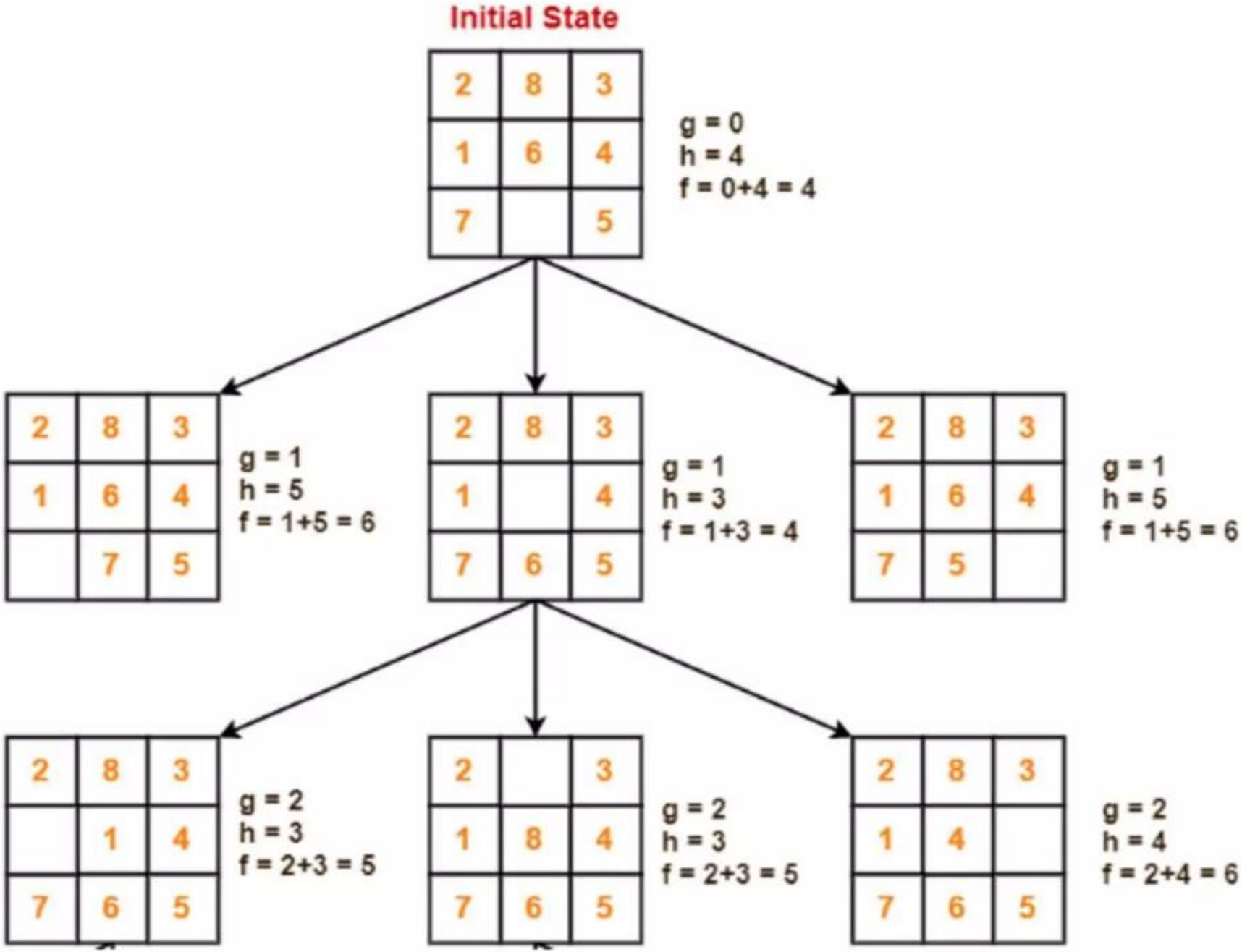
2	8	3
1	6	4
7		5

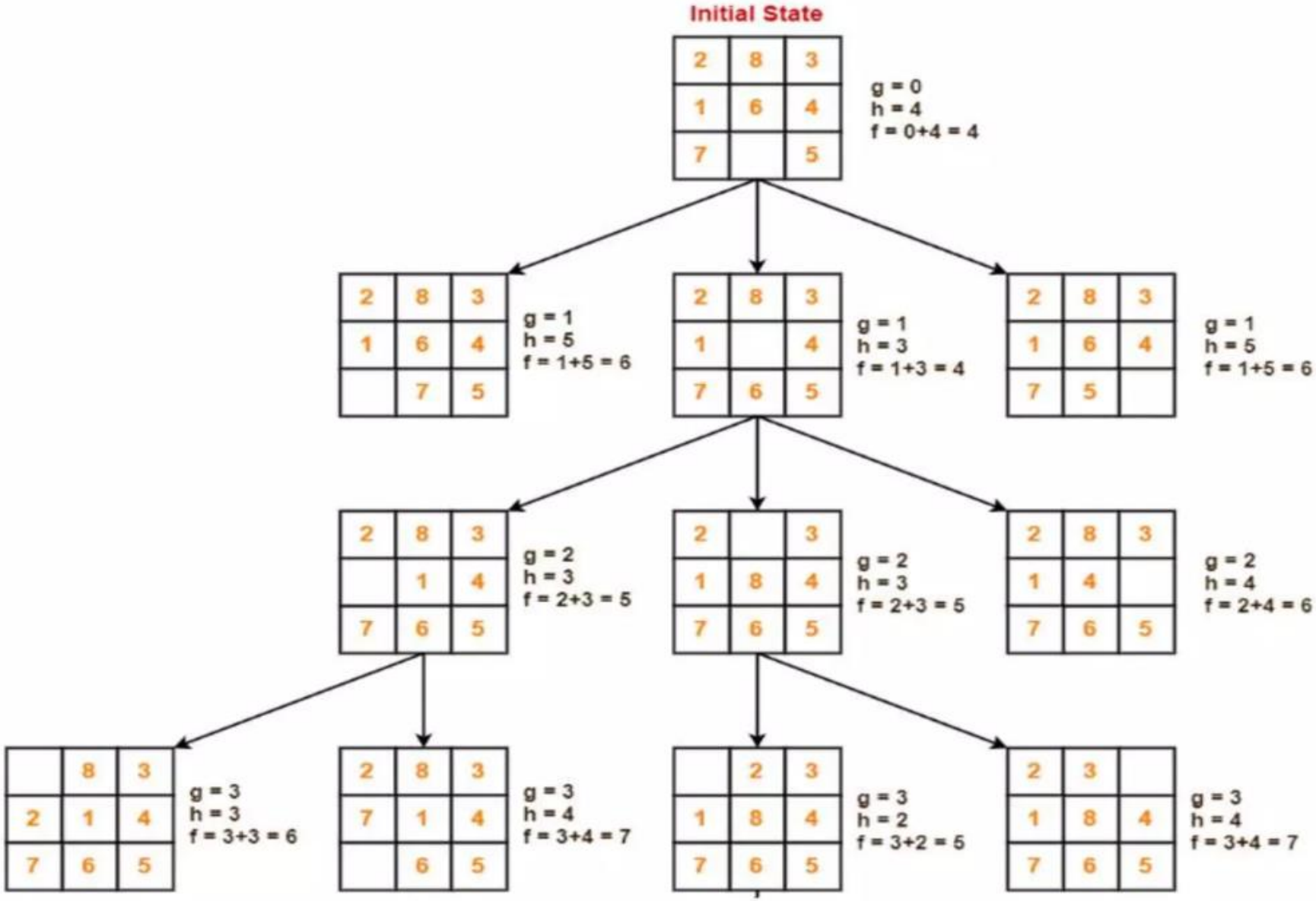
g = 0
h = 4
f = 0+4 = 4

Consider ,
g(n) = Depth of node
and
h(n) = Number of misplaced tiles.

1. Up
2. Down
3. Right or
4. Left


Initial State
2 8 3
1 6 4
7 5
g = 0
h = 4
f = 0+4 = 4
2 8 3
1 6 4
7 5
g = 1
h = 5
f = 1+5 = 6
2 8 3
1 4
7 6 5
g = 1
h = 3
f = 1+3 = 4
2 8 3
1 6 4
7 5
g = 1
h = 5
f = 1+5 = 6


Initial State
2 8 3
1 6 4
7 5
g = 0
h = 4
f = 0+4 = 4
2 8 3
1 6 4
7 5
g = 1
h = 5
f = 1+5 = 6
2 8 3
1 4
7 6 5
g = 1
h = 3
f = 1+3 = 4
2 8 3
1 6 4
7 5
g = 1
h = 5
f = 1+5 = 6
2 8 3
1 4
7 6 5
g = 2
h = 3
f = 2+3 = 5
2 3
1 8 4
7 6 5
g = 2
h = 3
f = 2+3 = 5
2 8 3
1 4
7 6 5
g = 2
h = 4
f = 2+4 = 6


Initial State
2 8 3
1 6 4
7 5
g = 0
h = 4
f = 0+4 = 4
2 8 3
1 6 4
7 5
g = 1
h = 5
f = 1+5 = 6
2 8 3
1 4
7 6 5
g = 1
h = 3
f = 1+3 = 4
2 8 3
1 6 4
7 5
g = 1
h = 5
f = 1+5 = 6
2 8 3
1 4
7 6 5
g = 2
h = 3
f = 2+3 = 5
2 3
1 8 4
7 6 5
g = 2
h = 3
f = 2+3 = 5
2 8 3
1 4
7 6 5
g = 2
h = 4
f = 2+4 = 6
8 3
2 1 4
7 6 5
g = 3
h = 3
f = 3+3 = 6
2 8 3
7 1 4
6 5
g = 3
h = 4
f = 3+4 = 7
2 3
1 8 4
7 6 5
g = 3
h = 2
f = 3+2 = 5
2 3
1 8 4
7 6 5
g = 3
h = 4
f = 3+4 = 7

The 8-puzzle

- **States**:
  - a state description specifies the location of each of the eight tiles and blank in one of the nine squares
- **Initial State**:
  - Any state in state space
- **Successor function**:
  - the blank moves *Left*, *Right*, *Up*, or *Down*
- **Goal test**:
  - current state matches the goal configuration
- **Path cost**:
  - each step costs 1, so the path cost is just the length of the path

The 8-queens

- There are two ways to formulate the problem
- All of them have the common followings:
  - Goal test: 8 queens on board, not attacking to each other
  - Path cost: zero

The 8-queens

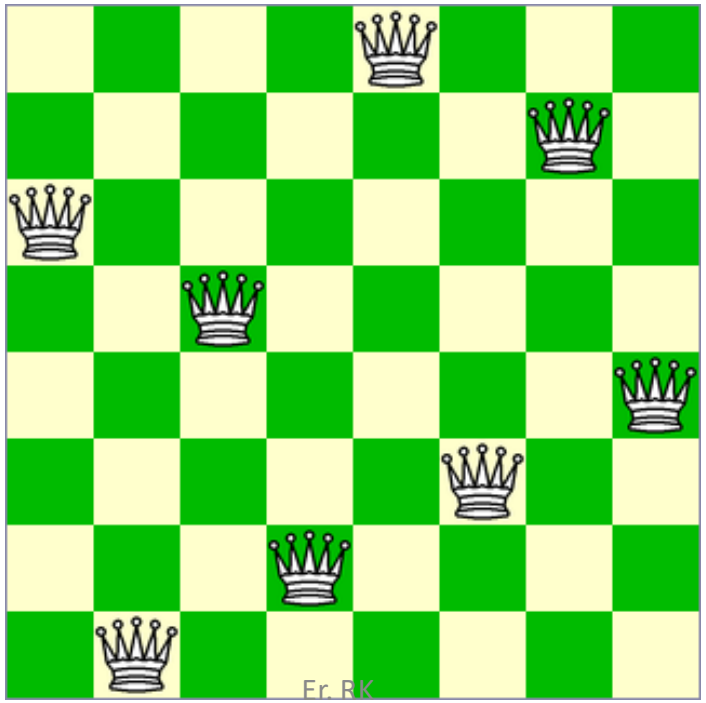
- (1) Incremental formulation
  - involves operators that augment the state description starting from an empty state
  - Each action adds a queen to the state
  - States:
    - any arrangement of 0 to 8 queens on board
  - Successor function:
    - add a queen to any empty square

The 8-queens

- (2) Complete-state formulation
  - starts with all 8 queens on the board
  - move the queens individually around
  - States:
    - any arrangement of 8 queens, one per column in the leftmost columns
  - Operators: move an attacked queen to a row, not attacked by any other

The 8-queens

- Conclusion:
  - the right formulation makes a big difference to the size of the search space



Example: River Crossing

- Items: Man, Wolf, Corn, Chicken.
- Man wants to cross river with all items.
  - Wolf will eat Chicken
  - Chicken will eat corn.
  - Boat will take max of two.

Challenges in State Space Search

- **State Explosion**: Large number of possible states.
- **Partially Observable States**: Incomplete information.
- **Cyclic Paths**: Re-exploring same states.
- Efficient search strategies (heuristics, pruning) are used to overcome these.

3.2 Searching for solutions

3.2 Searching for solutions

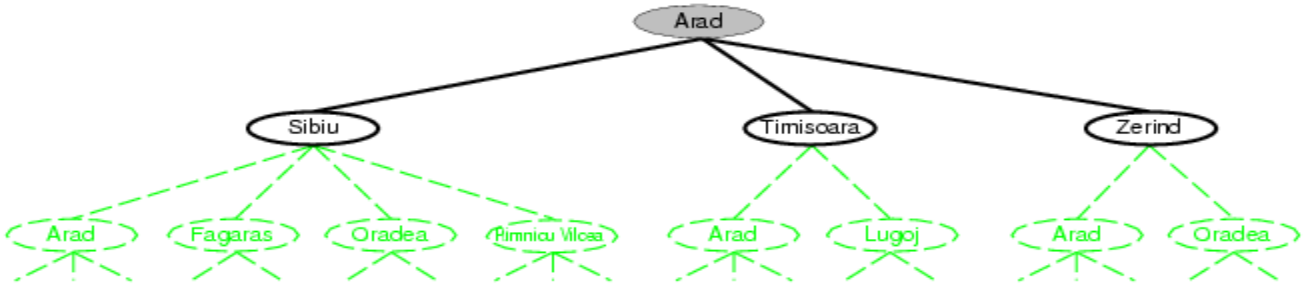
- Finding out a solution is done by
  - searching through the state space
- All problems are transformed
  - as a search tree
  - generated by the initial state and successor function

Search tree

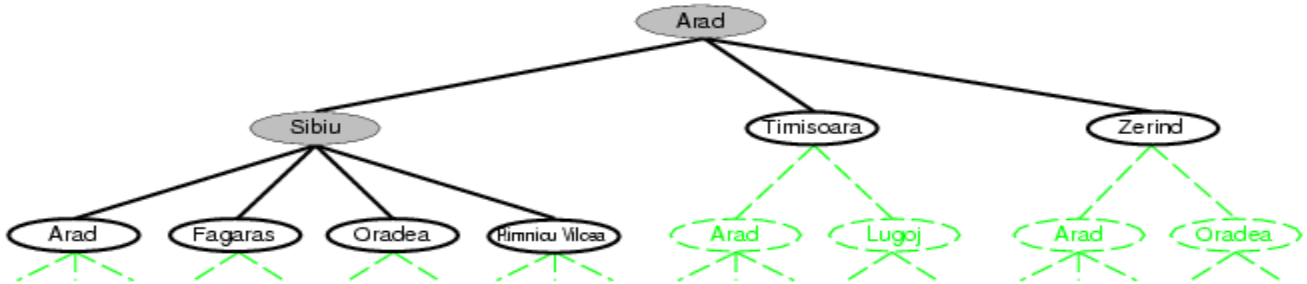
- **Initial state**
  - The root of the search tree is a **search node**
- **Expanding**
  - applying successor function to the current state
  - thereby generating a new set of states
- **leaf nodes**
  - the states having no successors

  Fringe : Set of search nodes that have not been expanded yet.
- Refer to next figure

Tree search example



Tree search example



Search tree

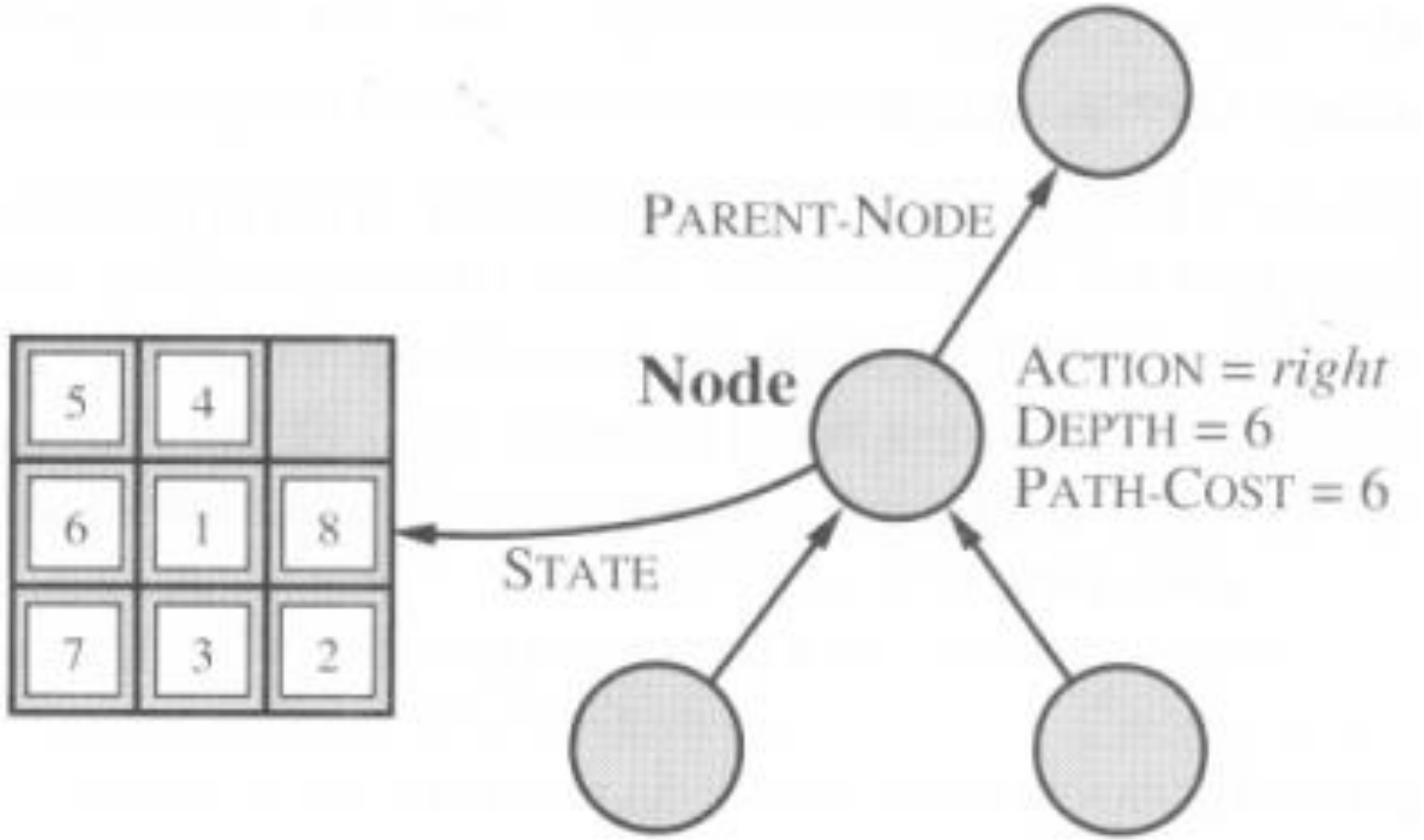
- The **essence** of searching
  - in case the first choice is not correct
  - choosing one option and keep others for later inspection
- Hence we have the **search strategy**
  - which determines the choice of which state to expand
  - good choice → fewer work → faster
- Important:
  - state space ≠ search tree

Search tree

- State space
  - has unique states {A, B}
  - while a search tree may have cyclic paths: A-B-A-B-A-B- …
- A good search strategy should avoid such paths

Search tree

- A node is having five components:
  - STATE: which state it is in the state space
  - PARENT-NODE: from which node it is generated
  - ACTION: which action applied to its parent-node to generate it
  - PATH-COST: the cost, g(n), from initial state to the node n itself
  - DEPTH: number of steps along the path from the initial state


PARENT-NODE
Node
ACTION = right
DEPTH = 6
PATH-COST = 6
STATE
5
4
6
1
8
7
3
2

Measuring problem-solving performance

- The evaluation of a search strategy
  - **Completeness:**
    - is the strategy guaranteed to find a solution when there is one?
  - **Optimality**:
    - does the strategy find the highest-quality solution when there are several different solutions?
  - **Time complexity**:
    - how long does it take to find a solution?
  - **Space complexity**:
    - how much memory is needed to perform the search?

Measuring problem-solving performance

- In AI, complexity is expressed in
  - **b**, branching factor, maximum number of successors of any node
  - **d**, the depth of the shallowest goal node.

  (depth of the least-cost solution)
  - **m**, the maximum length of any path in the state space
- Time and Space is measured in
  - number of nodes generated during the search
  - maximum number of nodes stored in memory

Measuring problem-solving performance

- For effectiveness of a search algorithm
  - we can just consider the total cost
  - **The total cost** = path cost (g) of the solution found **+** search cost
    - search cost = time necessary to find the solution
- Tradeoff:
  - (long time, optimal solution with least g)
  - vs. (shorter time, solution with slightly larger path cost g)

3.4 Uninformed Search Strategies

3.4 Uninformed search strategies

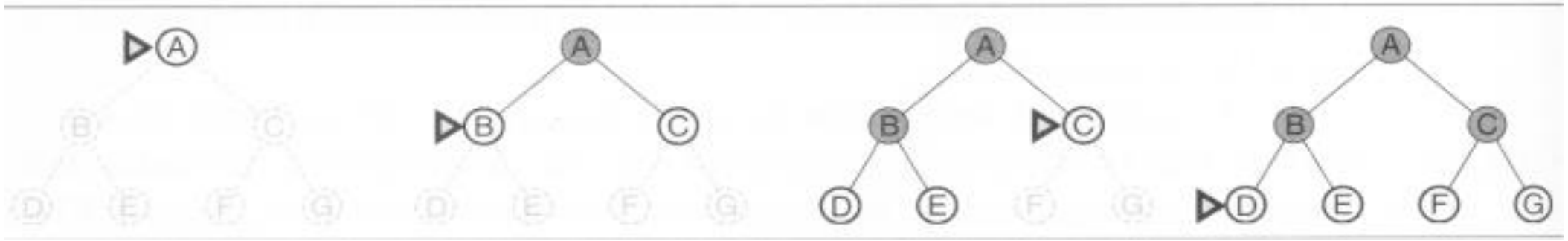
- **Uninformed search**
  - no information about the number of steps
  - or the path cost from the current state to the goal
  - search the state space **blindly**
- **Informed search, or heuristic search**
  - a cleverer strategy that searches toward the goal,
  - based on the information from the current state so far

Uninformed search strategies

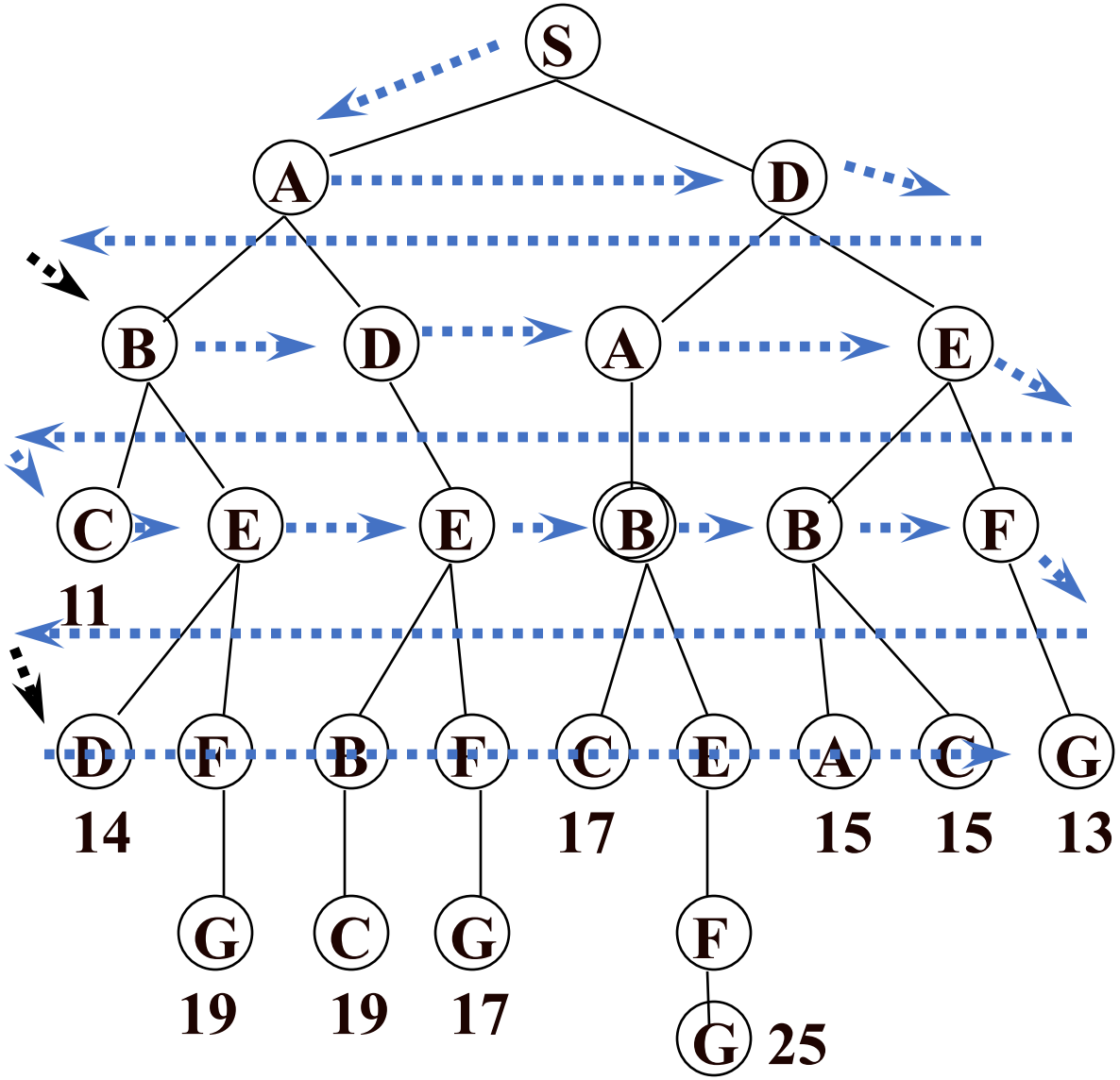
- Breadth-first search
  - Uniform cost search
- Depth-first search
  - Depth-limited search
  - Iterative deepening search
- Bidirectional search

Breadth-first search

- The root node is expanded first (FIFO)
- All the nodes generated by the root node are then expanded
- And then their successors and so on

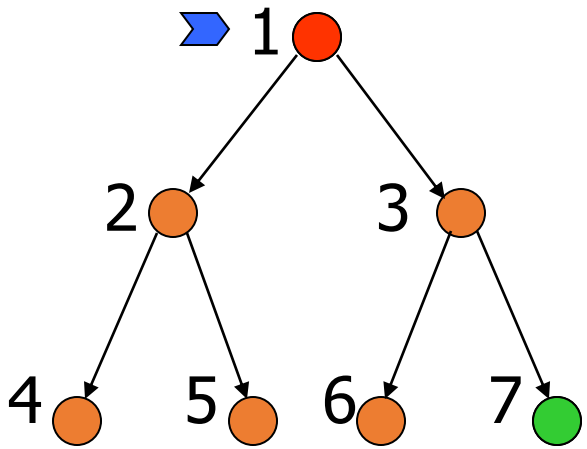


Breadth-first search



Breadth-First Strategy

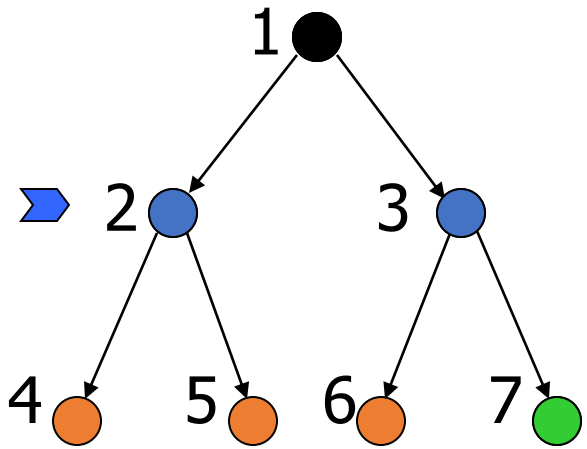
New nodes are inserted at the end of FRINGE



FRINGE = (1)

Breadth-First Strategy

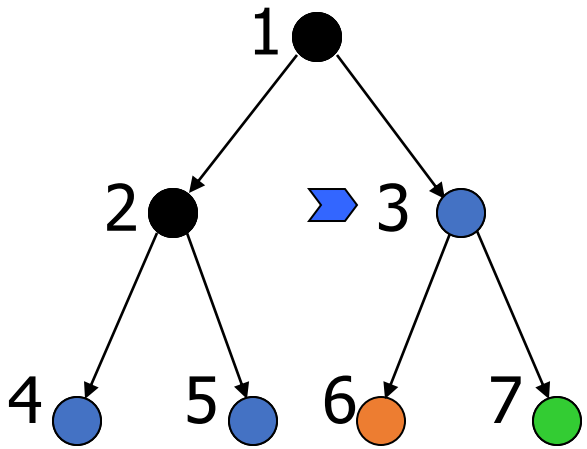
New nodes are inserted at the end of FRINGE



FRINGE = (2, 3)

Breadth-First Strategy

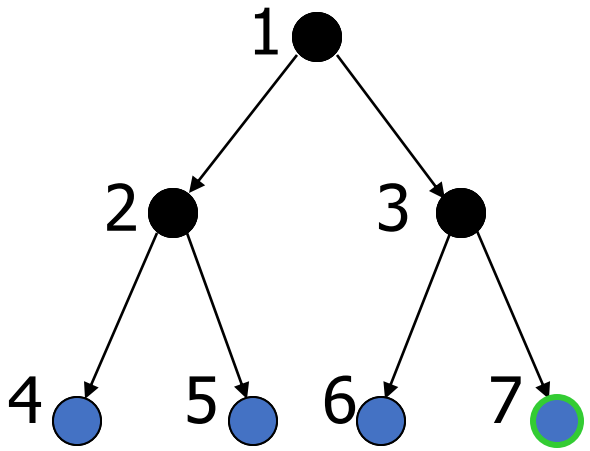
New nodes are inserted at the end of FRINGE



FRINGE = (3, 4, 5)

Breadth-First Strategy

New nodes are inserted at the end of FRINGE



FRINGE = (4, 5, 6, 7)

Breadth-first search (Analysis)

- Breadth-first search
  - Complete – find the solution eventually
  - Optimal, if step cost is 1

  The disadvantage
  - if the branching factor of a node is large,
  - for even small instances (e.g., chess)
    - the ***space complexity*** and the ***time complexity*** are enormous

Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d = b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
- Space is the bigger problem (more than time)

Breadth-first search (Analysis)

- assuming 10000 nodes can be processed per second, each with 1000 bytes of storage

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

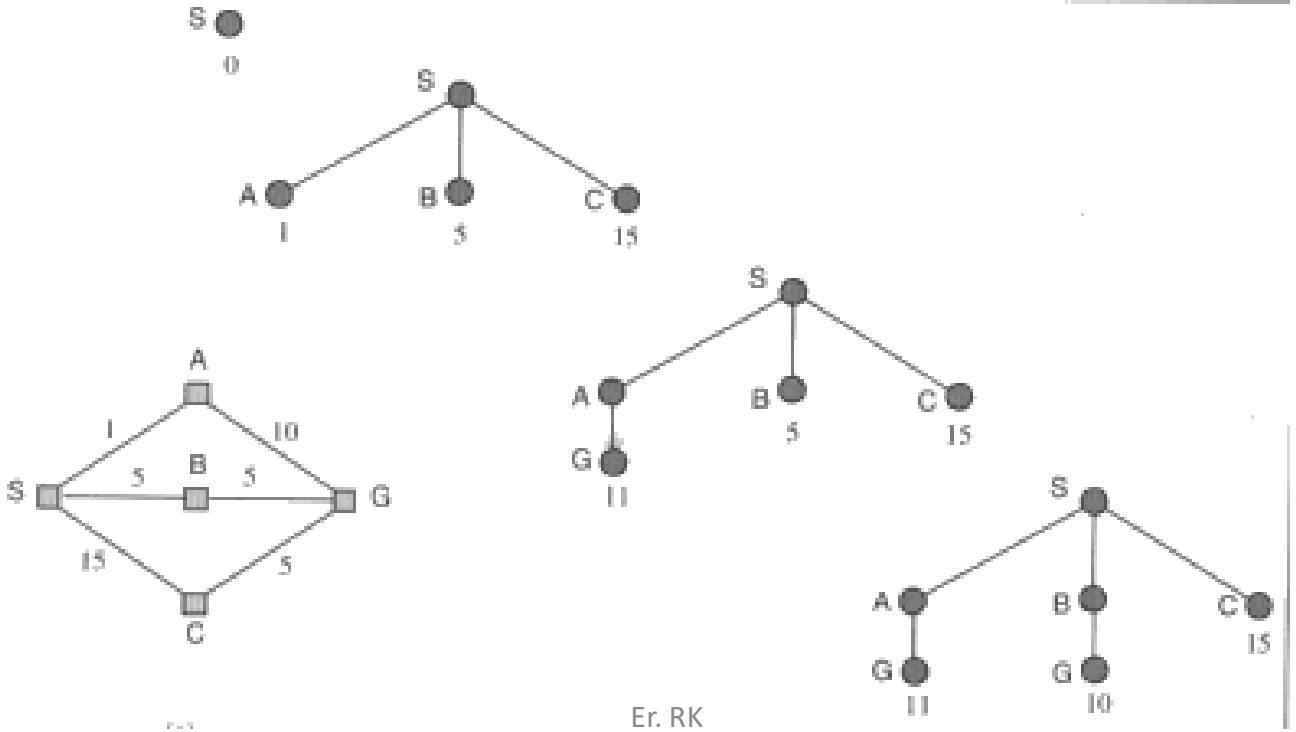
Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 10,000 nodes/second; 1000 bytes/node.

Uniform cost search

- **Breadth-first** finds the shallowest goal state
  - but not necessarily be the least-cost solution
  - work only if all step costs are equal
- **Uniform cost search**
  - modifies breadth-first strategy
    - by always expanding the lowest-cost node
  - The lowest-cost node is measured by the path cost *g(n)*

Uniform cost search

- the first found solution is guaranteed to be the cheapest
  - least in depth
  - But restrict to ***non-decreasing*** path cost
  - Unsuitable for operators with negative cost



Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost ≥ ε
- Time? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$

let

C be the cost of optimal solution.*

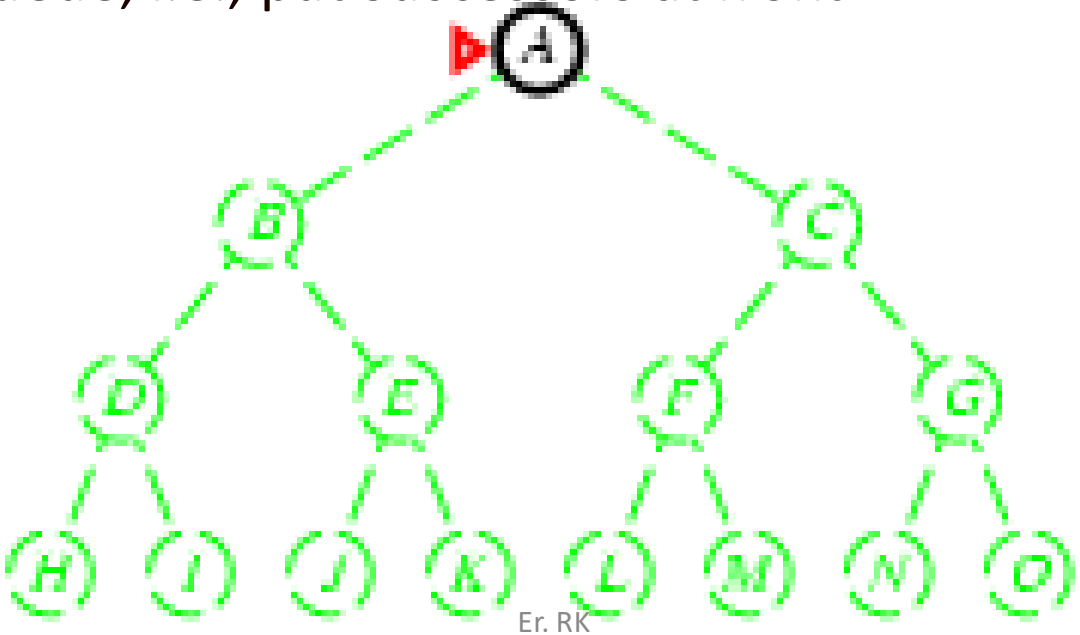
ε is possitive constant (every action cost)

Depth-First search (DFS)

- Always expands one of the nodes at the ***deepest*** level of the tree
- Only when the search hits a dead end
  - goes back and expands nodes at shallower levels
  - Dead end → leaf nodes but not the goal
- Backtracking search
  - only one successor is generated on expansion
  - rather than all successors
  - fewer memory

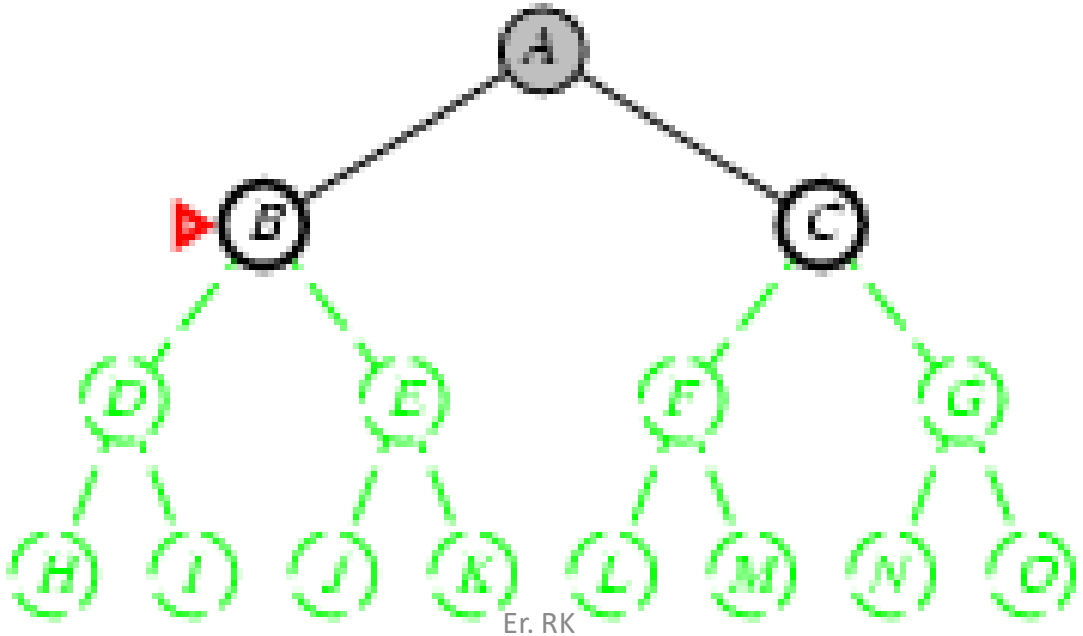
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



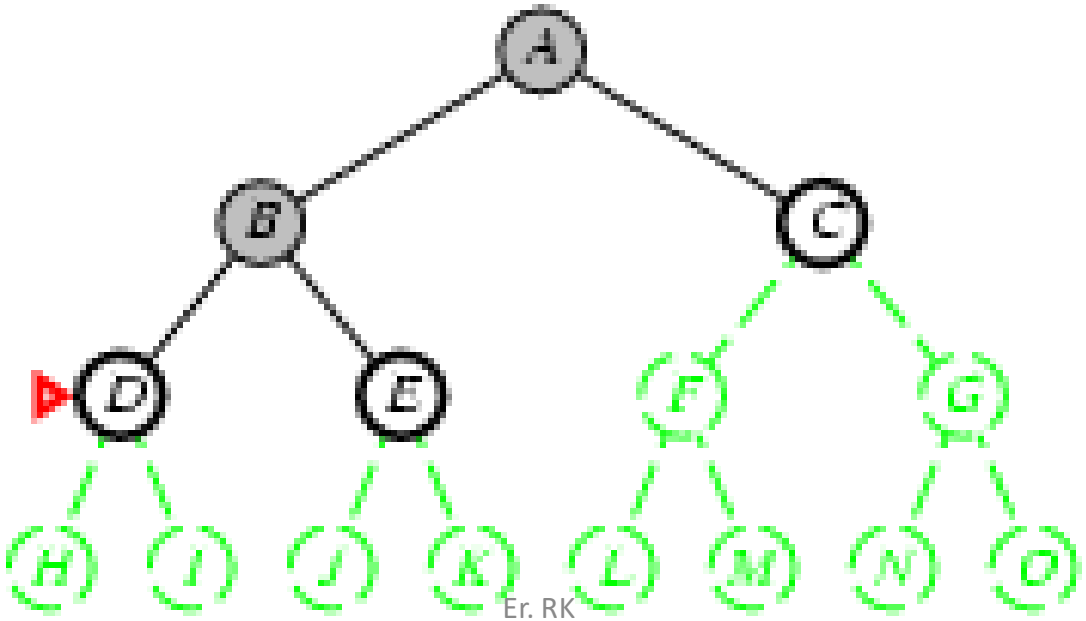
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



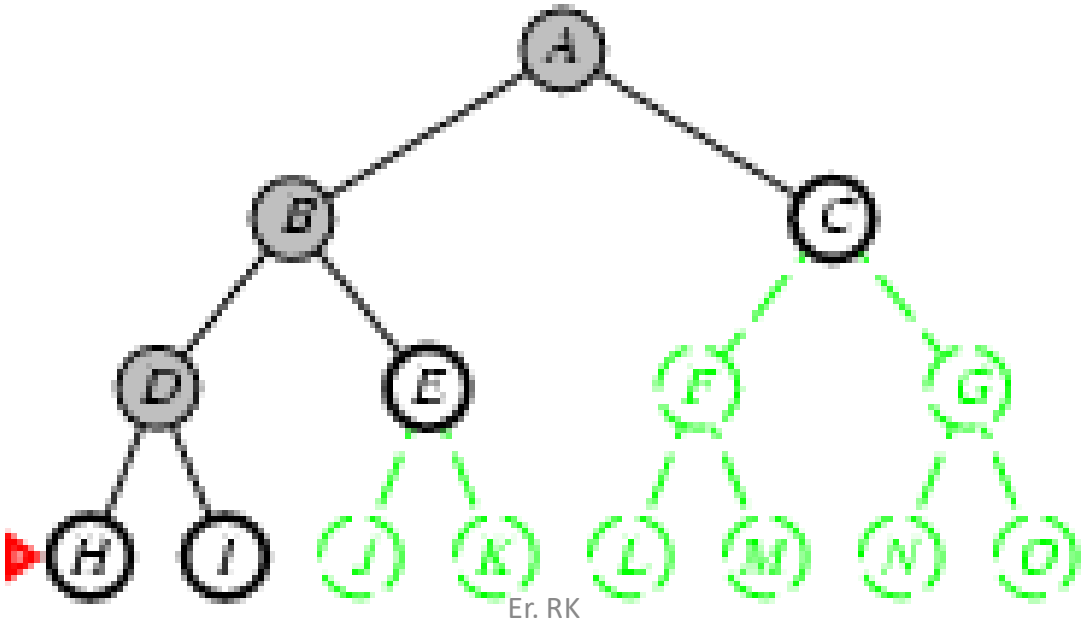
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



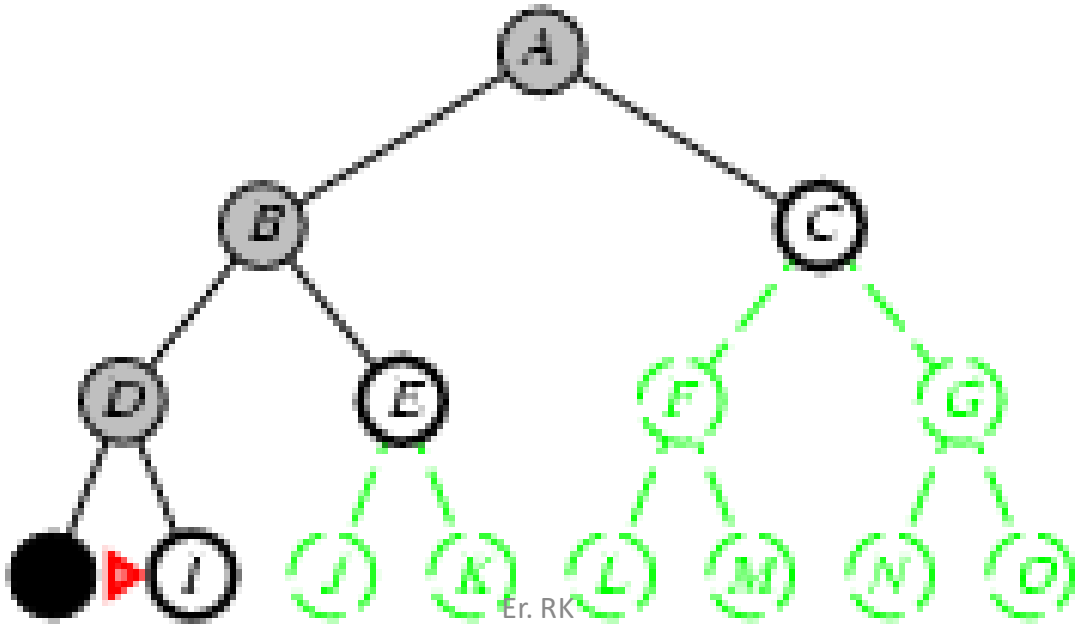
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



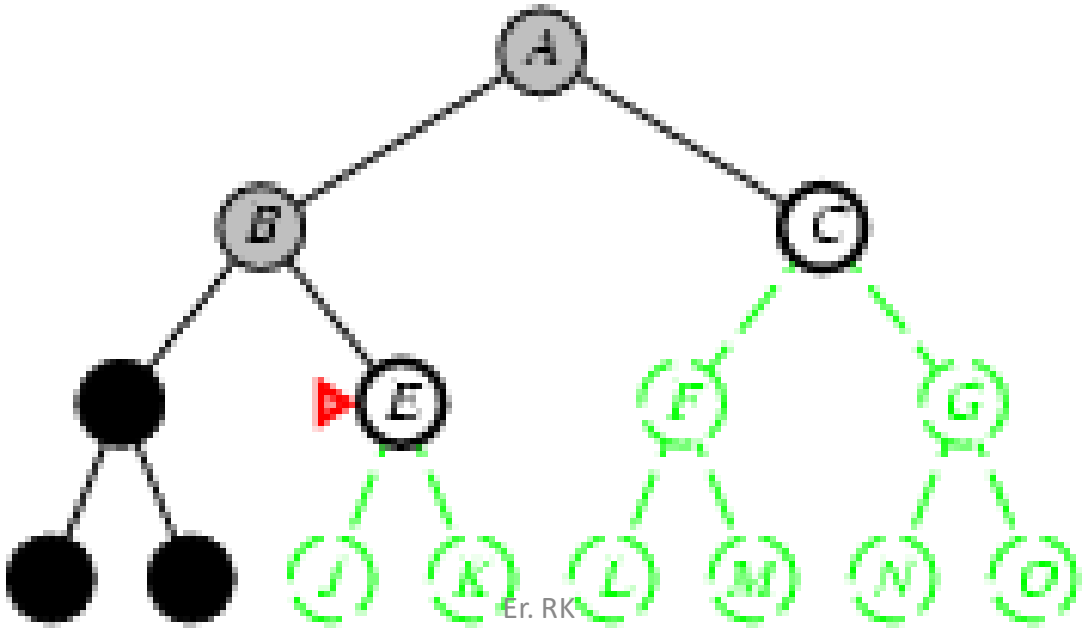
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



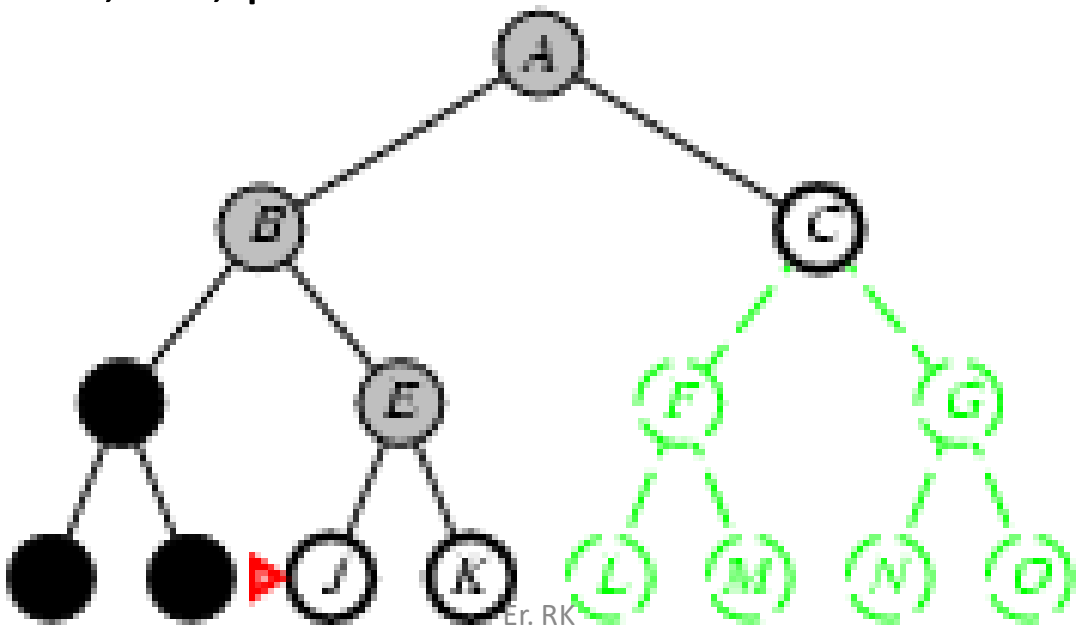
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



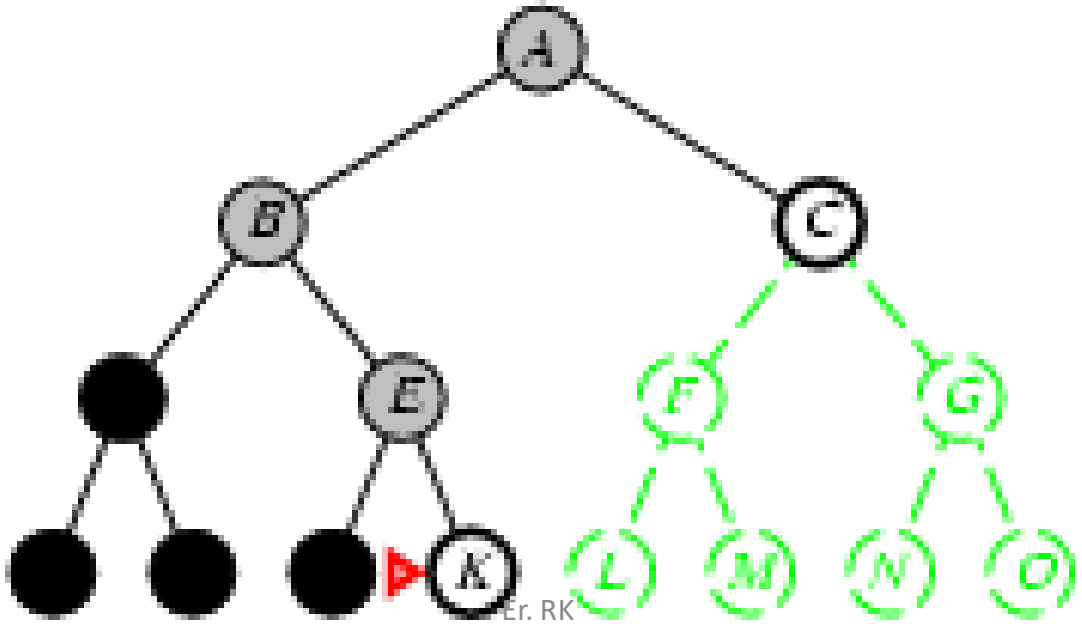
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



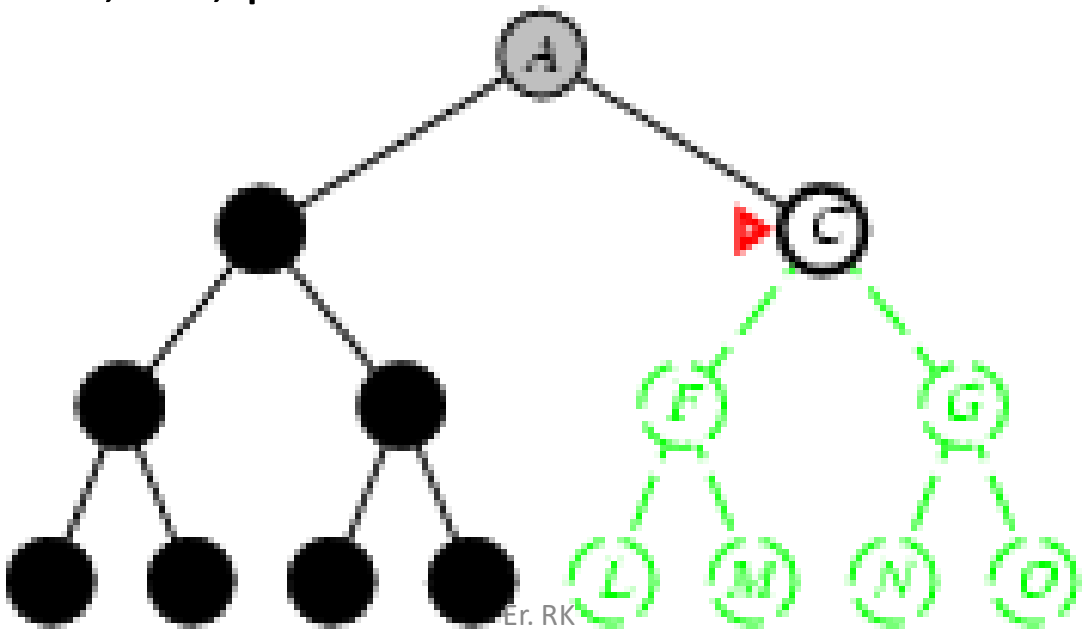
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



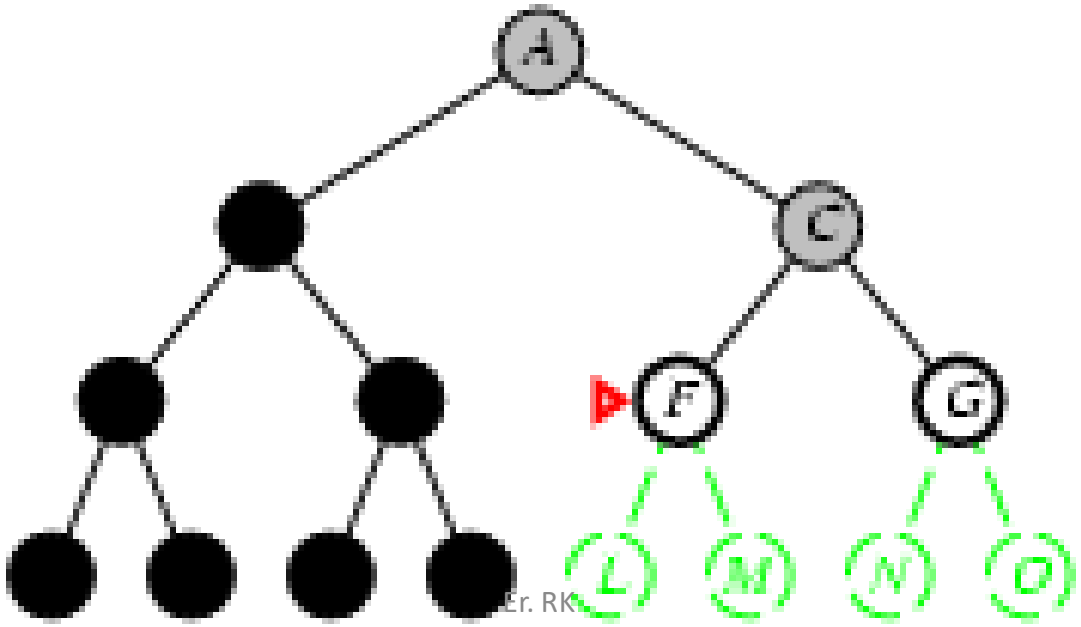
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



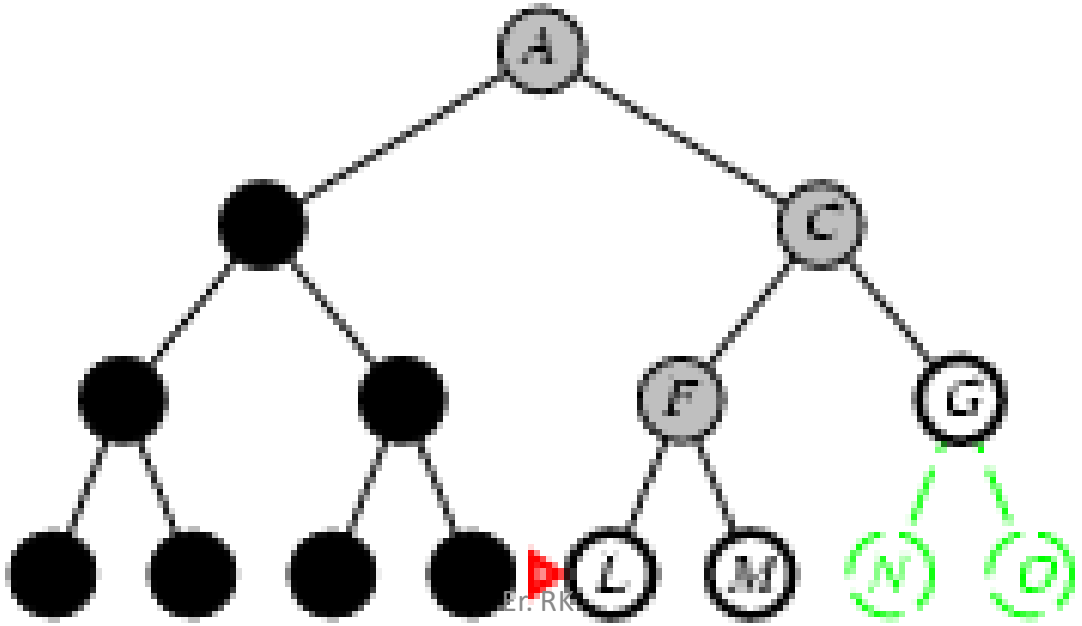
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



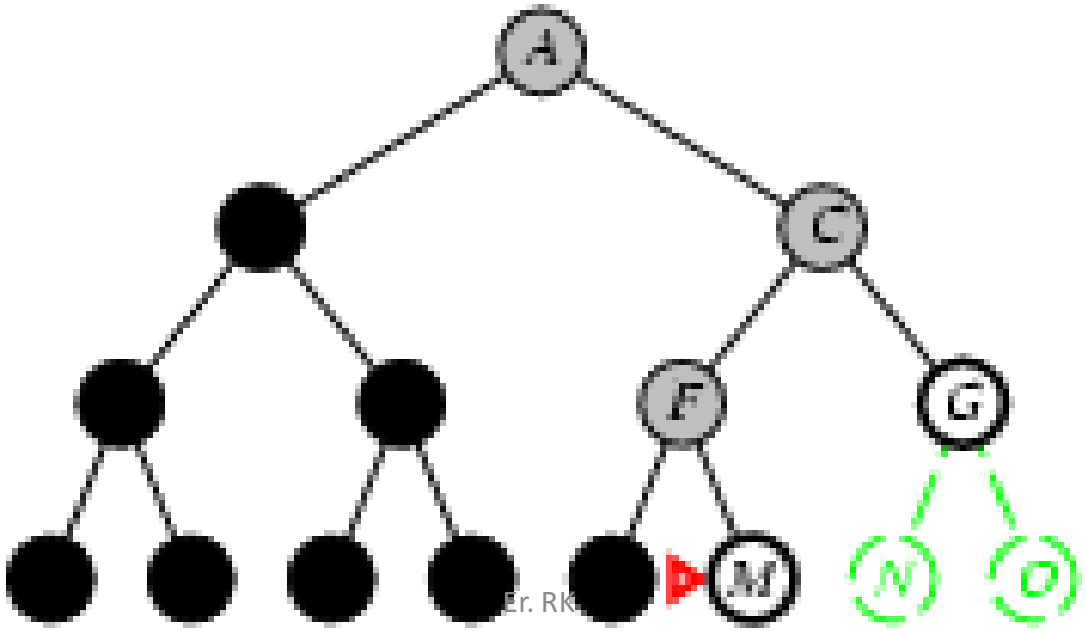
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -

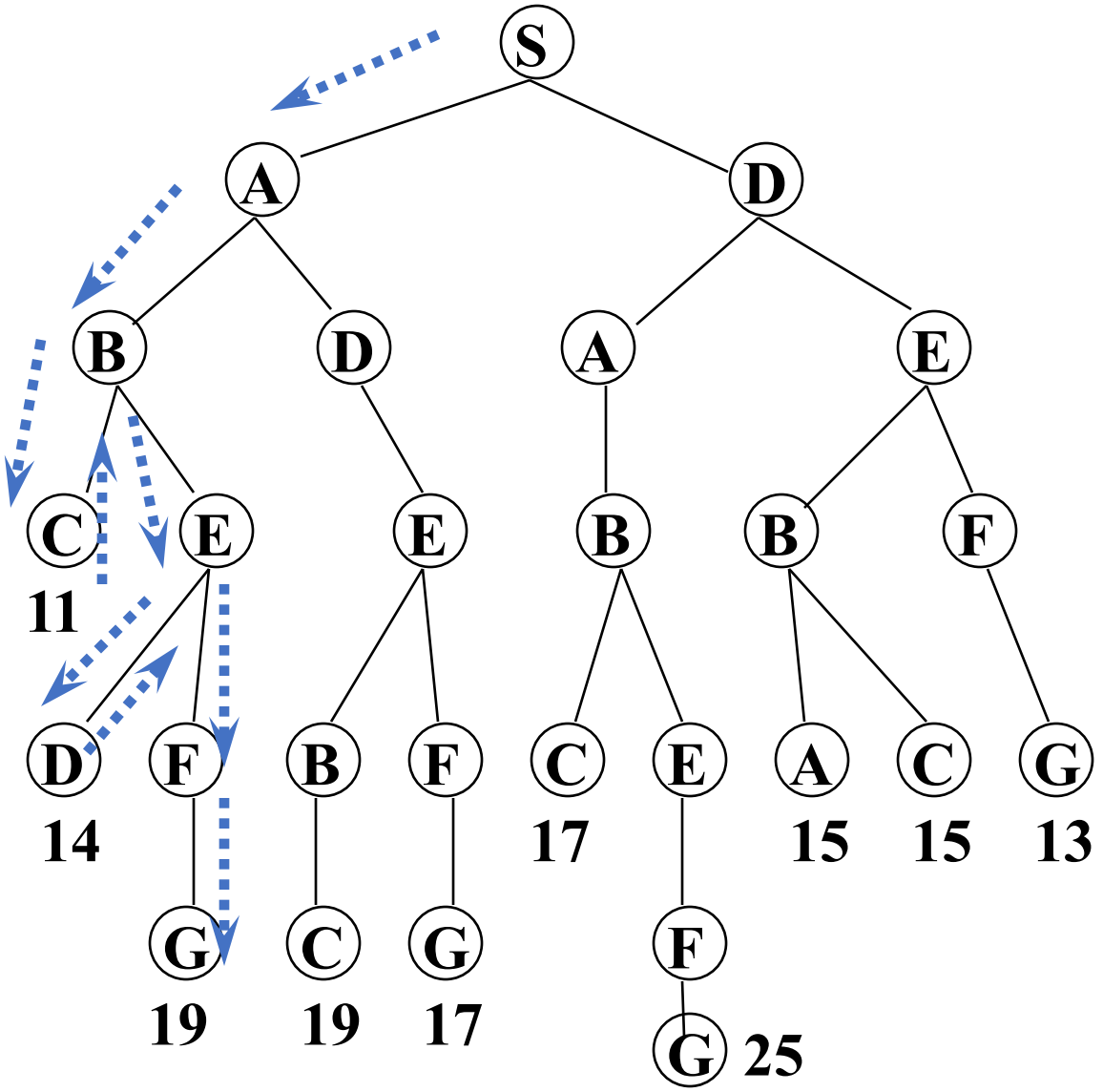


Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -



Depth-first search



Depth-first search (Analysis)

- Not complete
  - because a path may be infinite or looping
  - then the path will never fail and go back try another option
- Not optimal
  - it doesn't guarantee the best solution
- It overcomes
  - the time and space complexities

Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
  - → complete in finite 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 Strategy

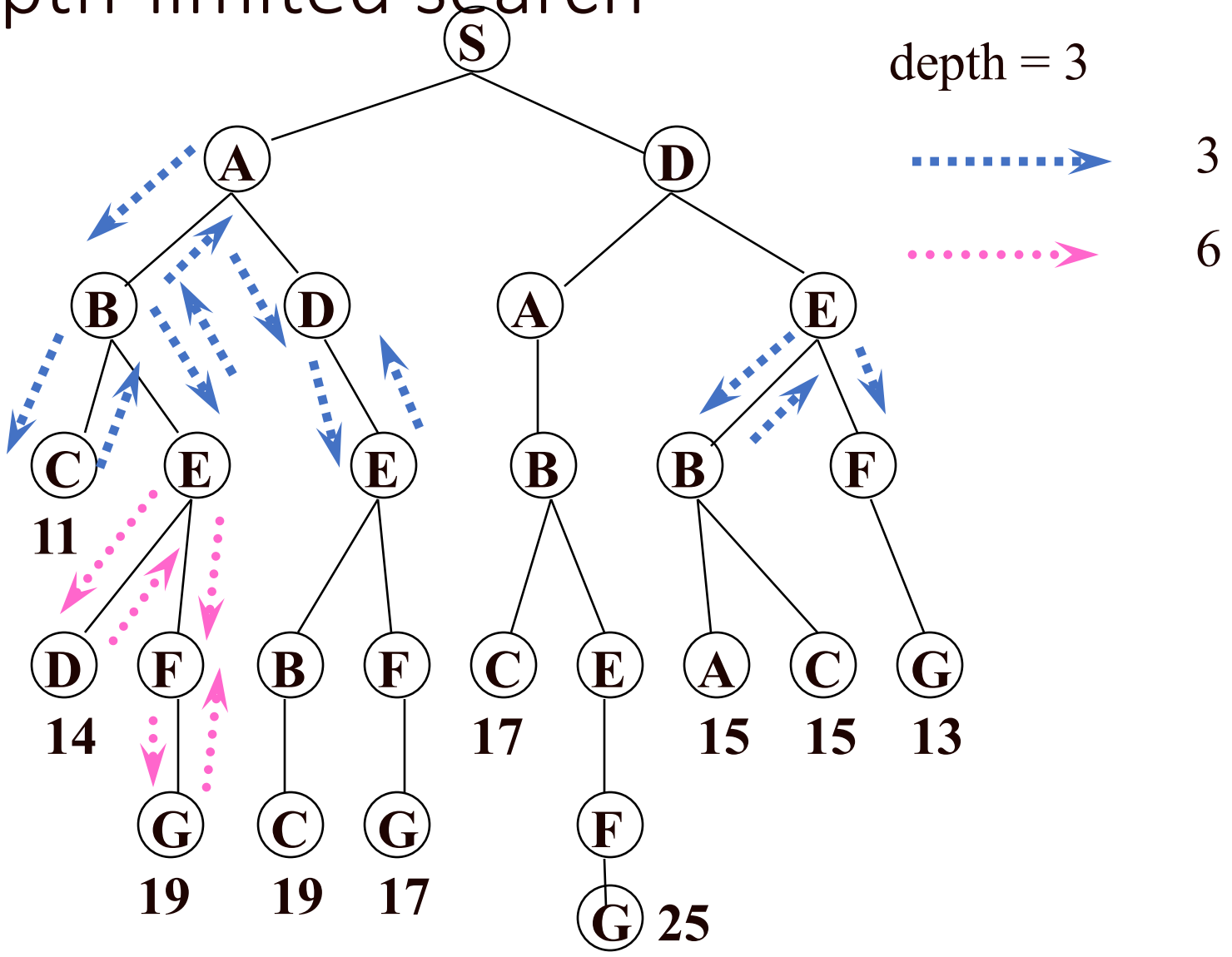
- Depth-first with depth cutoff k (maximal depth below which nodes are not expanded)

- Three possible outcomes:
  - Solution
  - Failure (no solution)
  - Cutoff (no solution within cutoff)

Depth-limited search

- It is depth-first search
  - with a **predefined** maximum depth
  - However, it is usually not easy to define the suitable maximum depth
  - too small → no solution can be found
  - too large → the same problems are suffered from
- Anyway the search is
  - complete
  - but still not optimal

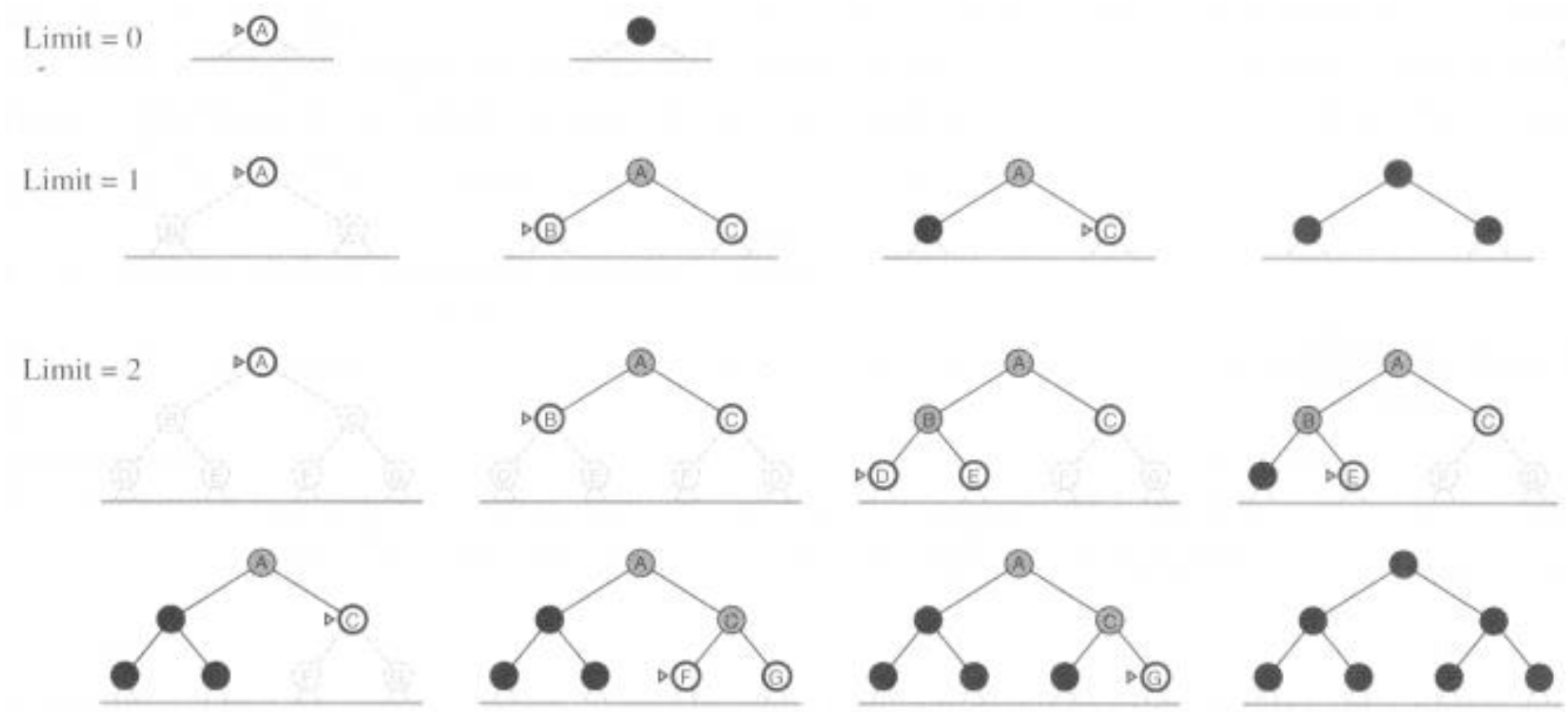
Depth-limited search



Iterative deepening search

- No choosing of the best depth limit
- It tries all possible depth limits:
  - first 0, then 1, 2, and so on
  - combines the benefits of depth-first and breadth-first search

Iterative deepening search



Iterative deepening search (Analysis)

- optimal
- complete
- Time and space complexities
  - reasonable
- suitable for the problem
  - having a large search space
  - and the depth of the solution is not known

Properties of iterative deepening search

- Complete? Yes
-
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
-
- Space? $O(bd)$
-
- Optimal? Yes, if step cost = 1

Iterative lengthening search

- IDS is using depth as limit
- ILS is using path cost as limit
  - an iterative version for uniform cost search

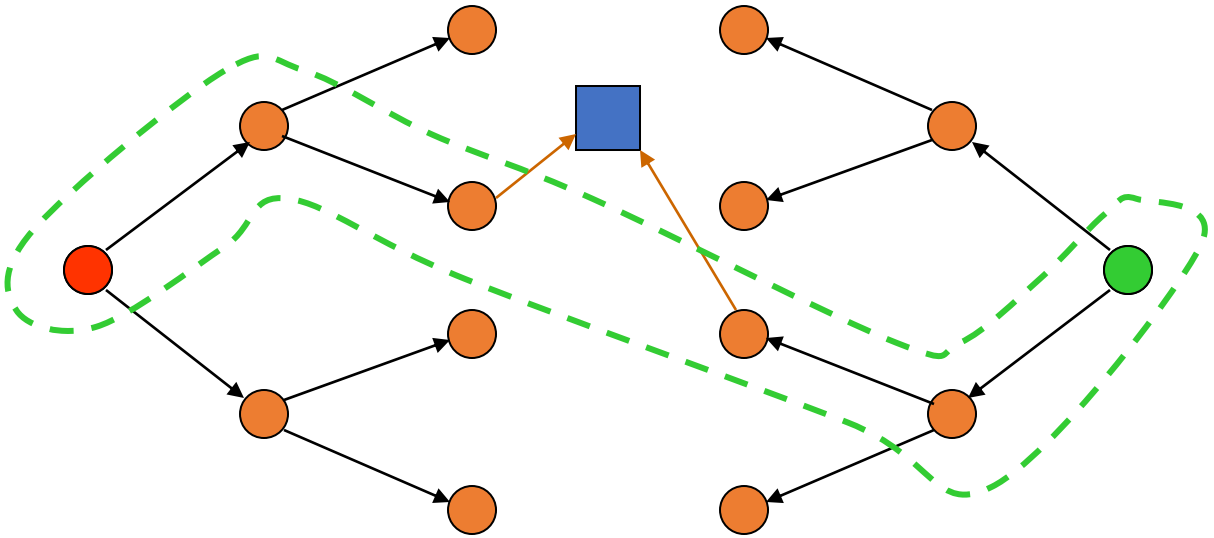
  has the advantages of uniform cost search
    - while avoiding its memory requirements
  - but ILS incurs substantial overhead
    - compared to uniform cost search

Bidirectional search

- Run two simultaneous searches
  - one forward from the initial state another backward from the goal
  - stop when the two searches meet
- However, computing backward is difficult
  - A huge amount of goal states
  - at the goal state, which actions are used to compute it?
  - can the actions be reversible to computer its predecessors?

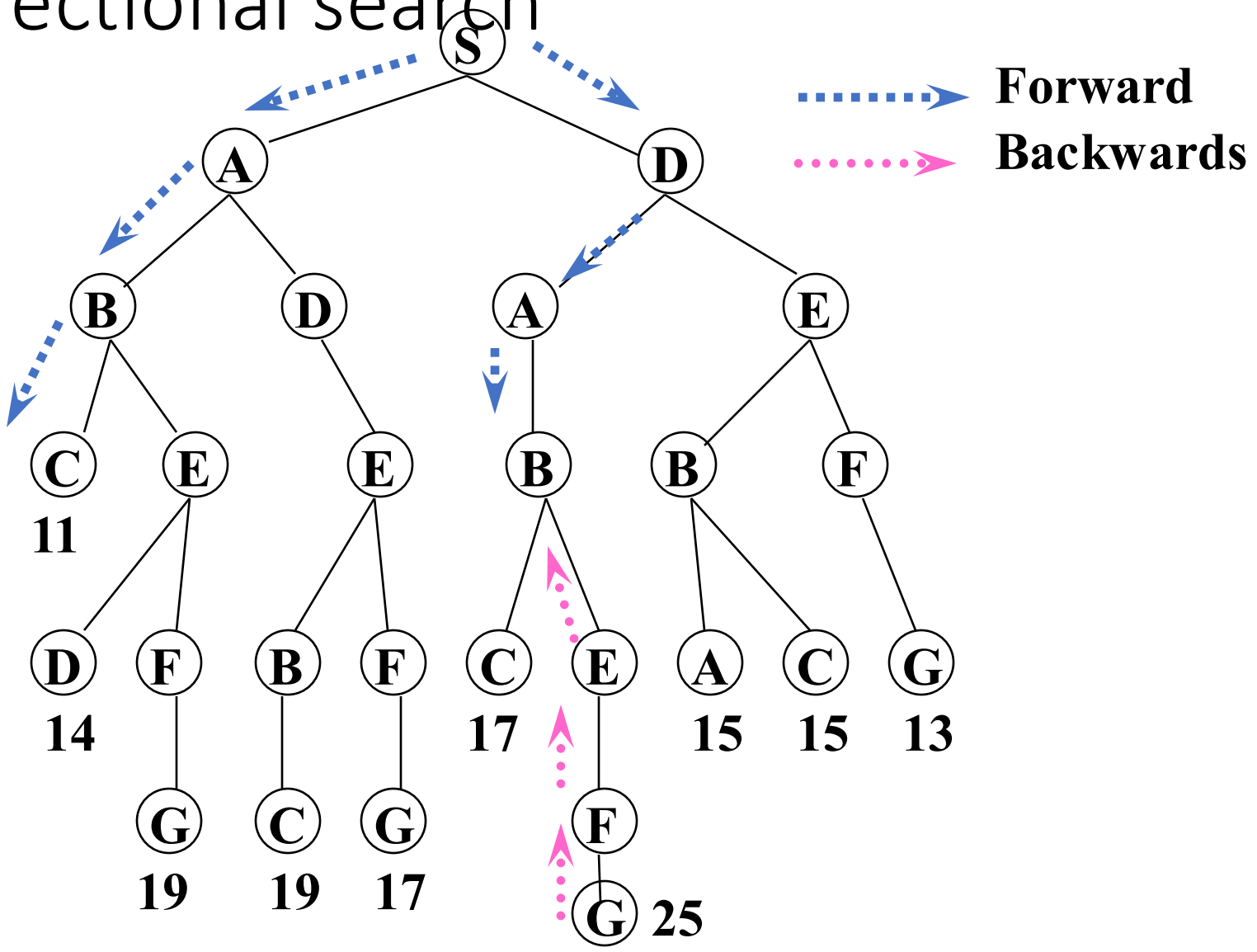
Bidirectional Strategy

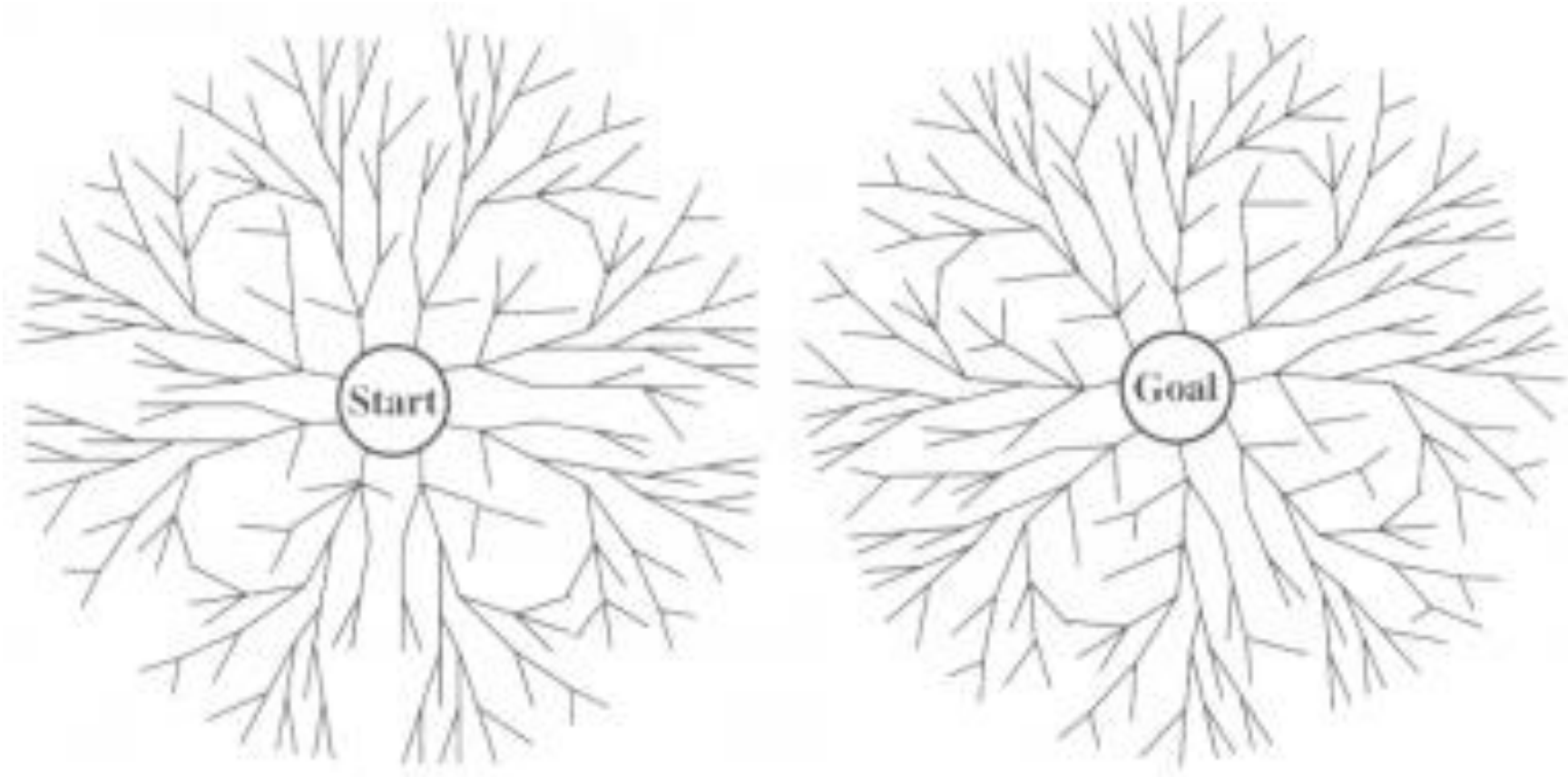
2 fringe queues: FRINGE1 and FRINGE2



Time and space complexity = $O(b^{d/2})$ << $O(b^{d})$

Bidirectional search




Start
Goal

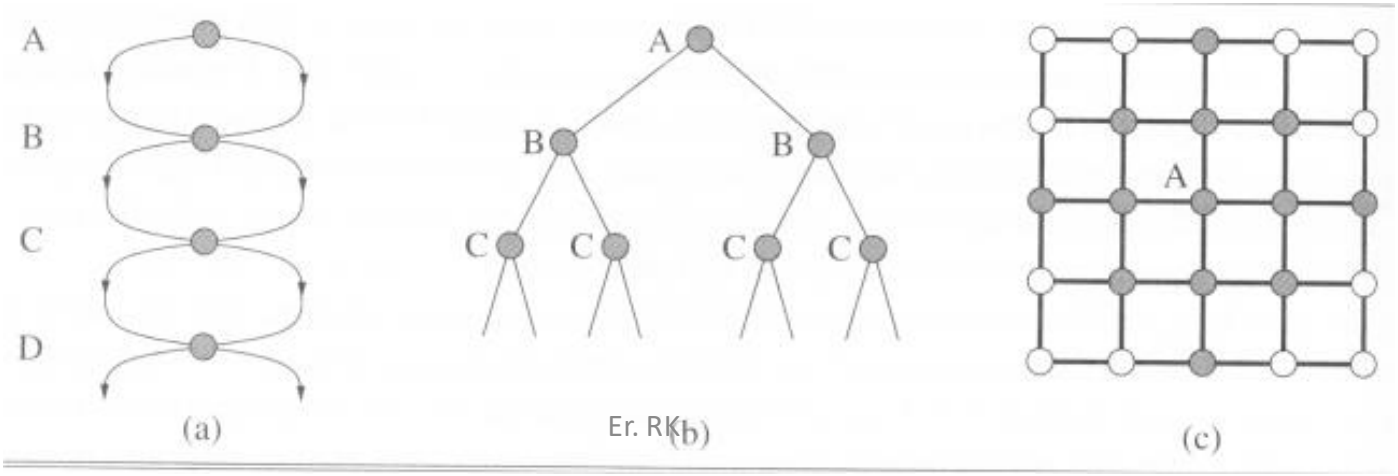
Comparing search strategies

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	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.17 Evaluation of search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Avoiding repeated states

- for all search strategies
  - There is possibility of expanding states
    - that have already been encountered and expanded before, on some other path
  - may cause the path to be infinite → loop forever
  - Algorithms that forget their history
    - are doomed to repeat it



Avoiding repeated states

- Three ways to deal with this possibility
  - Do not return to the state it just came from
    - Refuse generation of any successor same as its **parent state**
  - Do not create paths with cycles
    - Refuse generation of any successor same as its **ancestor states**
  - Do not generate any *generated* state
    - Not only its ancestor states, but also all other expanded states have to be checked against

Avoiding repeated states

- We then define a data structure
  - closed list:

  a set storing every expanded node so far
  - If the current node matches a node on the closed list, discard it.

```
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 EMPTY?( fringe) then return failure
      node ← REMOVE-FIRST( fringe)
      if GOAL-TEST[ problem](STATE[node]) then return SOLUTION(node)
      if STATE[node] is not in closed then
          add STATE[node] to closed
          fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

INFORMED SEARCH TECHNIQUE HEURISTIC_SEARCH

Greedy best-first search

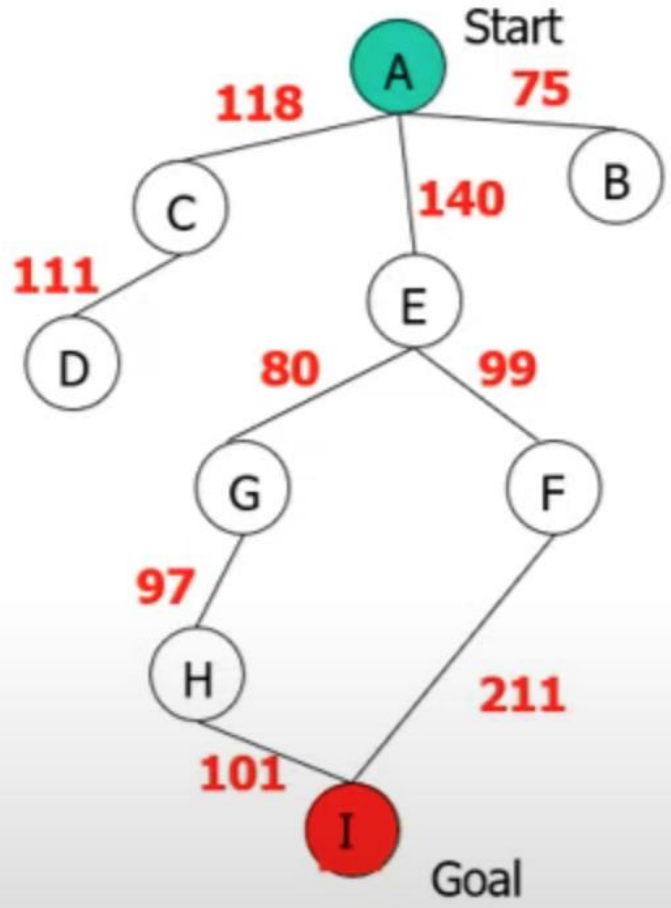
- Greedy best-first search algorithm always selects the path which appears best at that moment.
- It is the combination of depth-first search and breadth-first search algorithms.
- It uses the heuristic function and search.
- With the help of best-first search, at each step, we can choose the most promising node.
- In the best first search algorithm, we expand the node which is closest to the goal node and the minimum cost is estimated by heuristic function

Greedy best-first search

- The evaluation function is f(n) = h(n)
- Were, h(n)= estimated cost from node n to the goal.

- Greedy search ignores the cost of the path that has already been traversed to reach n
- Therefore, the solution given is not necessarily optimal

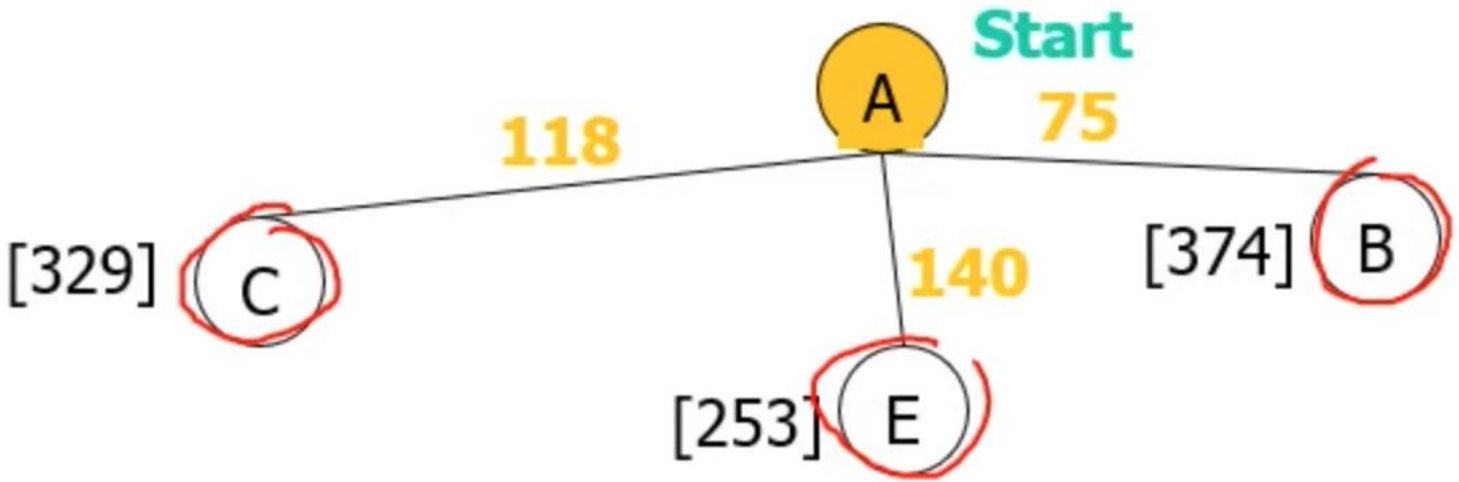
Greedy best-first search…



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

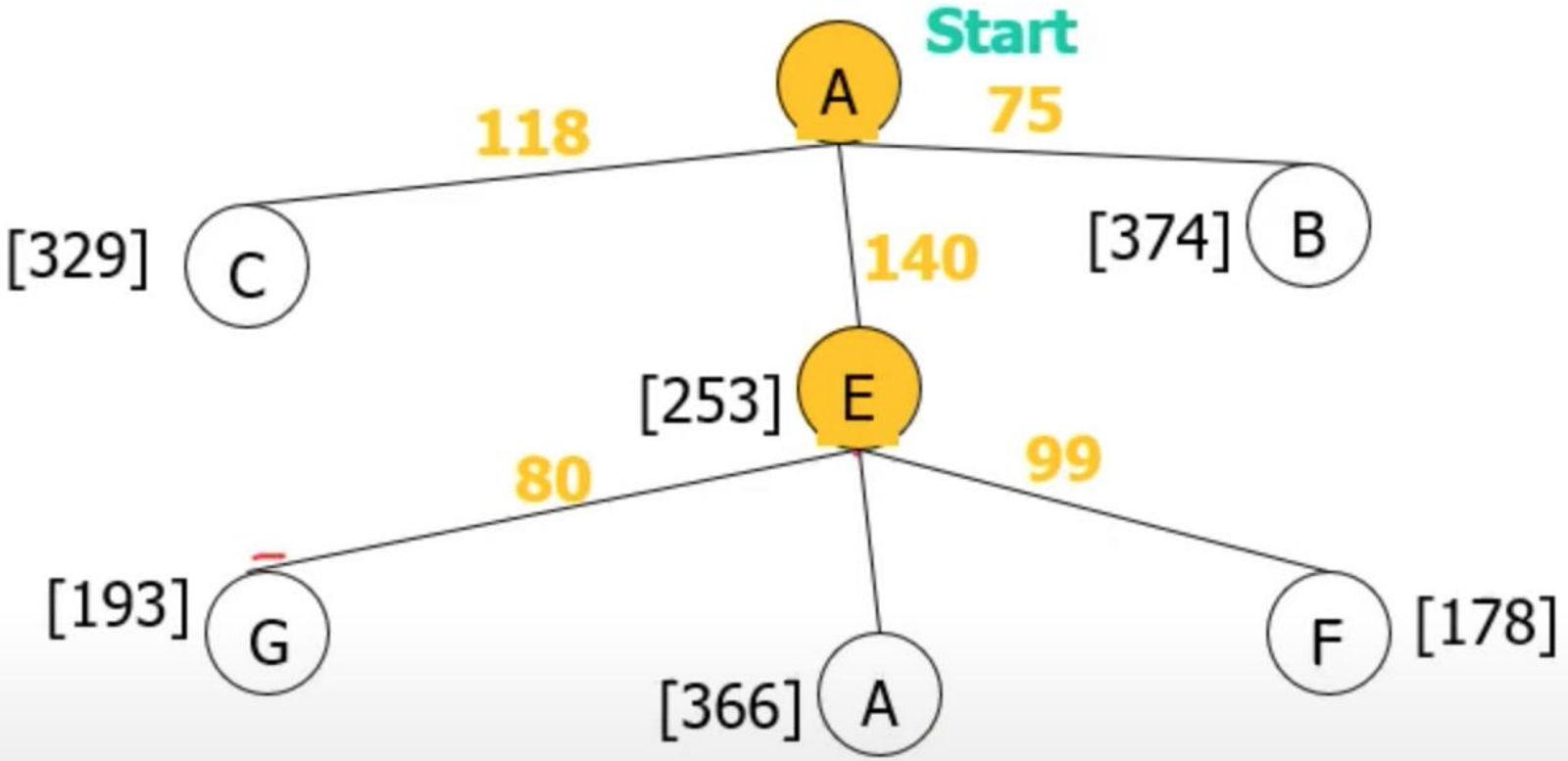
f(n) = h(n) = straight-line distance heuristic

Greedy Search: Tree Search



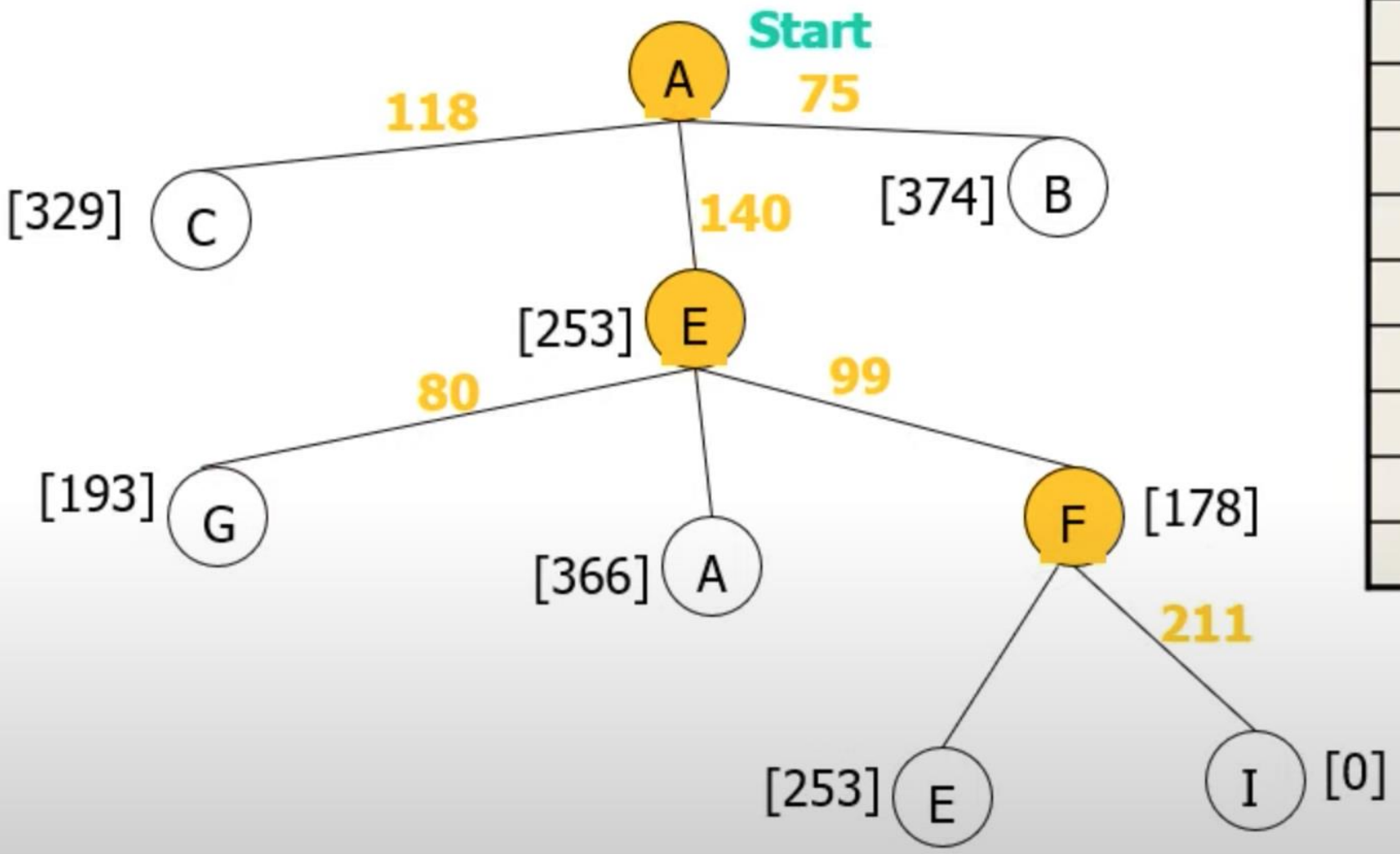
State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

Greedy Search: Tree Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

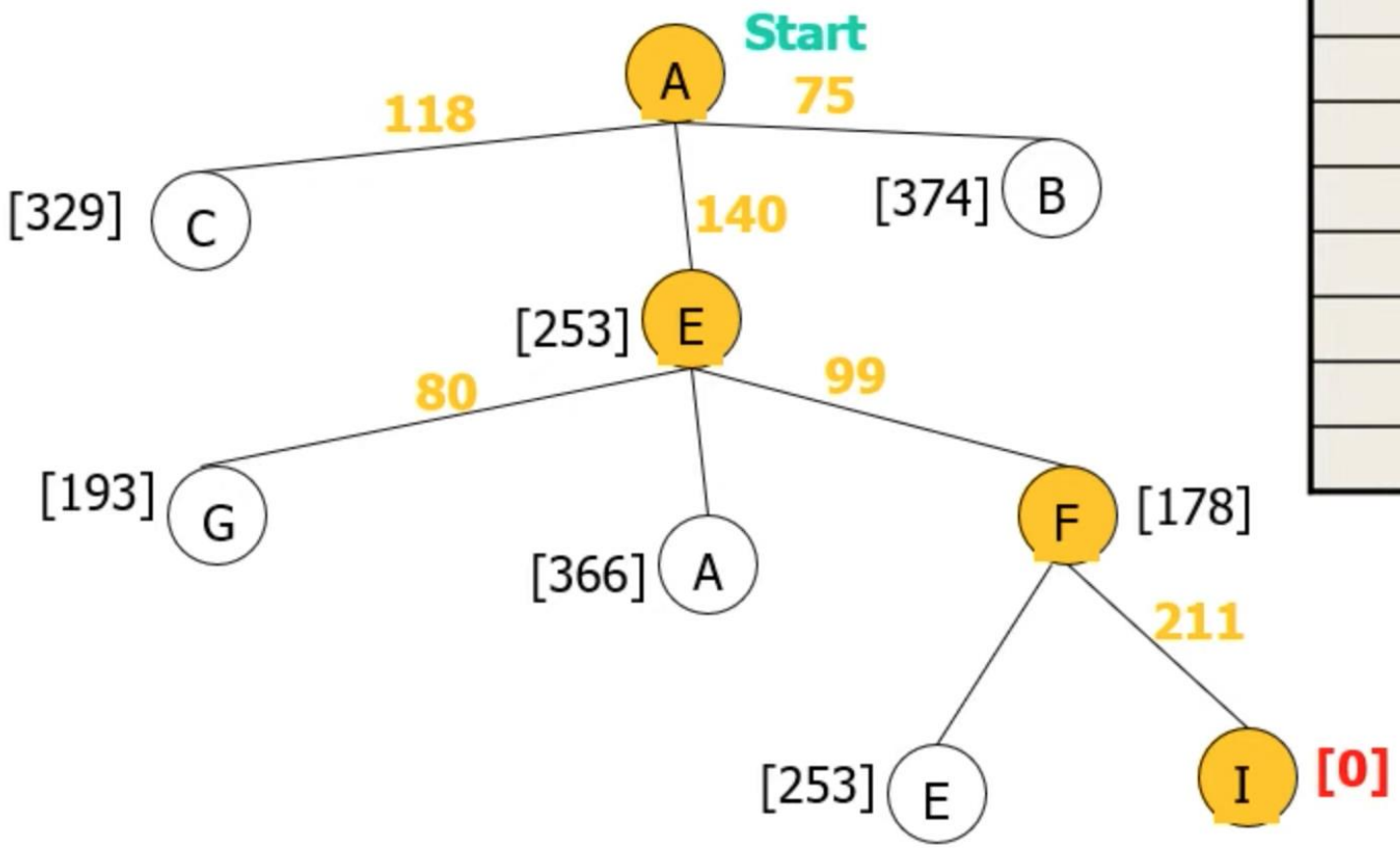
Greedy Search: Tree Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

Greedy Search: Tree Search

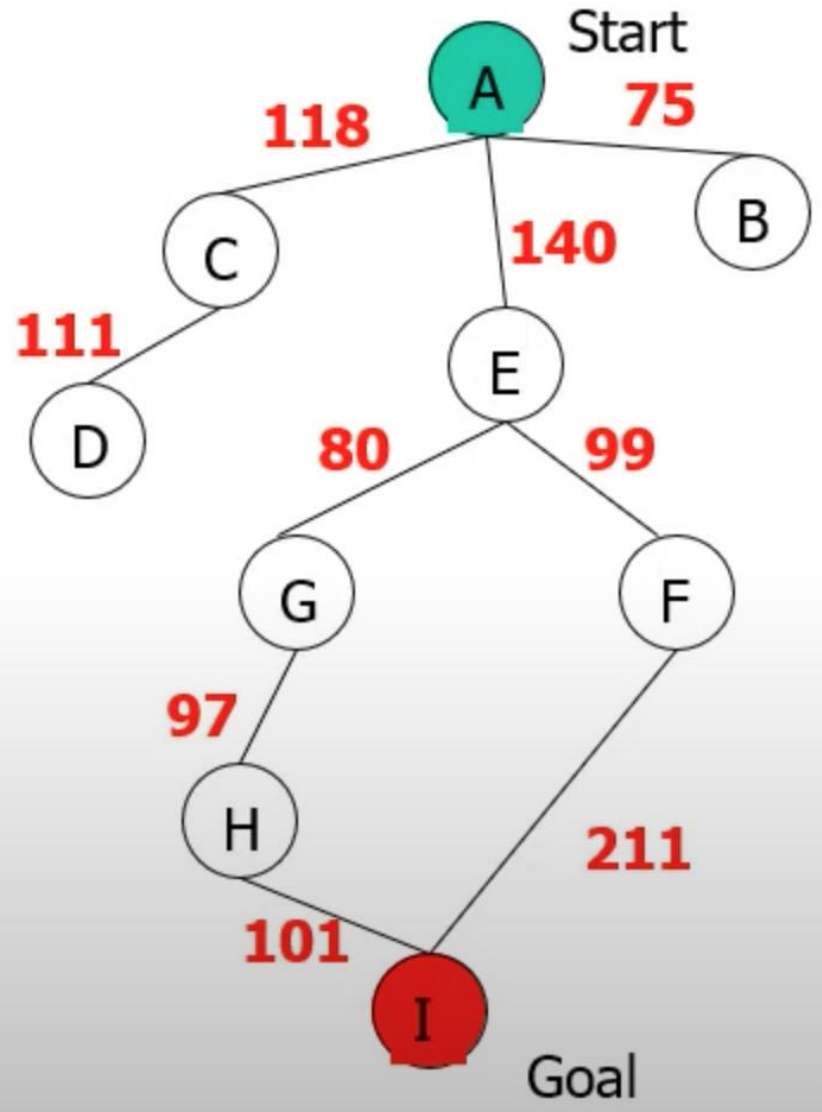
State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0



Path cost(A-E-F-I) = 253 + 178 + 0 = 431

dist(A-E-F-I)= 140 +99+211=450

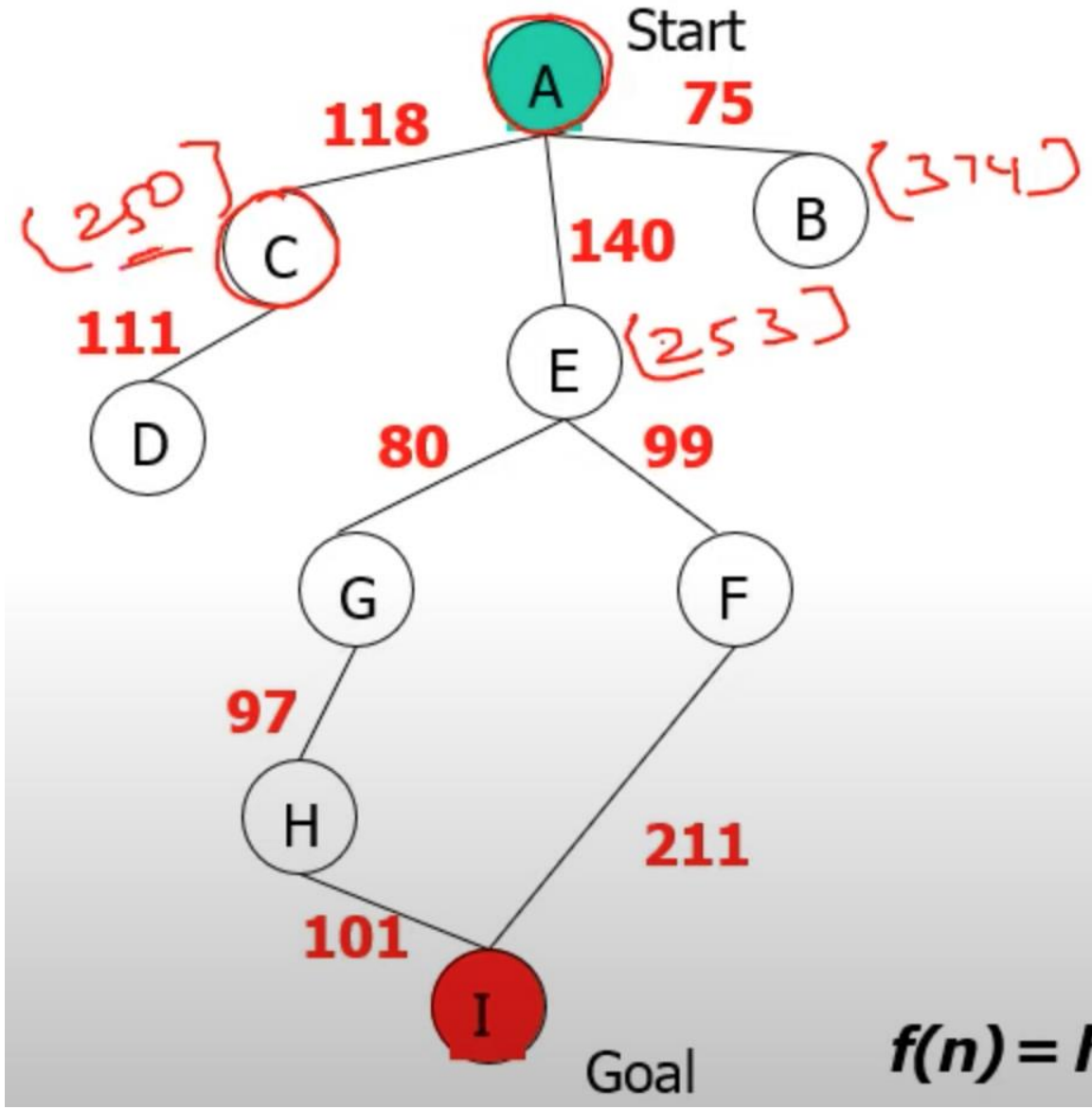
Greedy Search: Complete ?



State	Heuristic: h(n)
A	366
B	374
** C	250
D	244
E	253
F	178
G	193
H	98
I	0

***f(n) = h (n)* = straight-line distance heuristic**

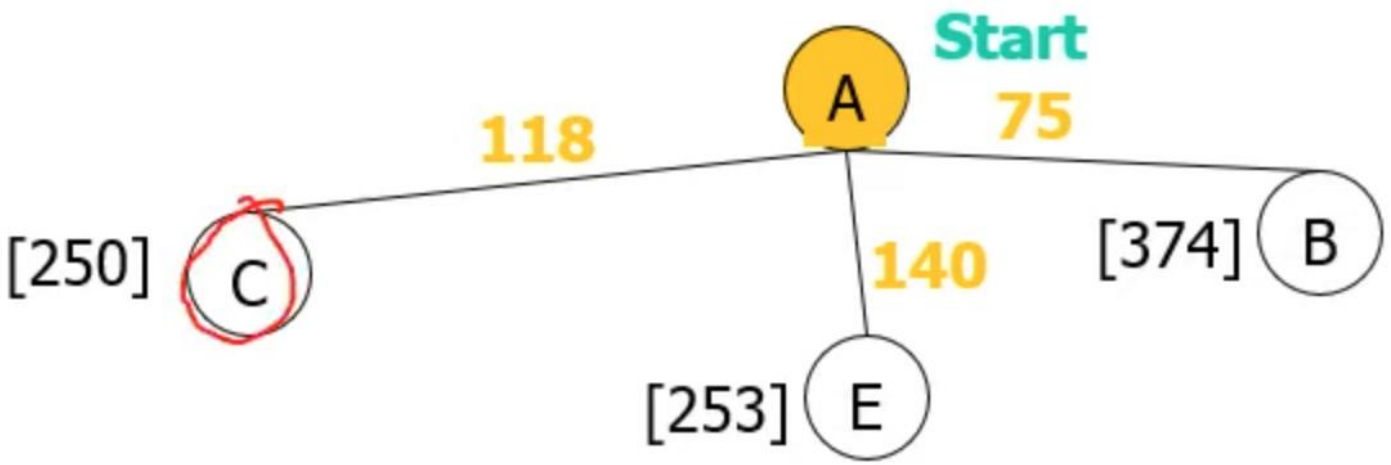
Greedy Search: Complete ?



State	Heuristic: h(n)
A	366
B	374
** C	250
D	244
E	253
F	178
G	193
H	98
I	0

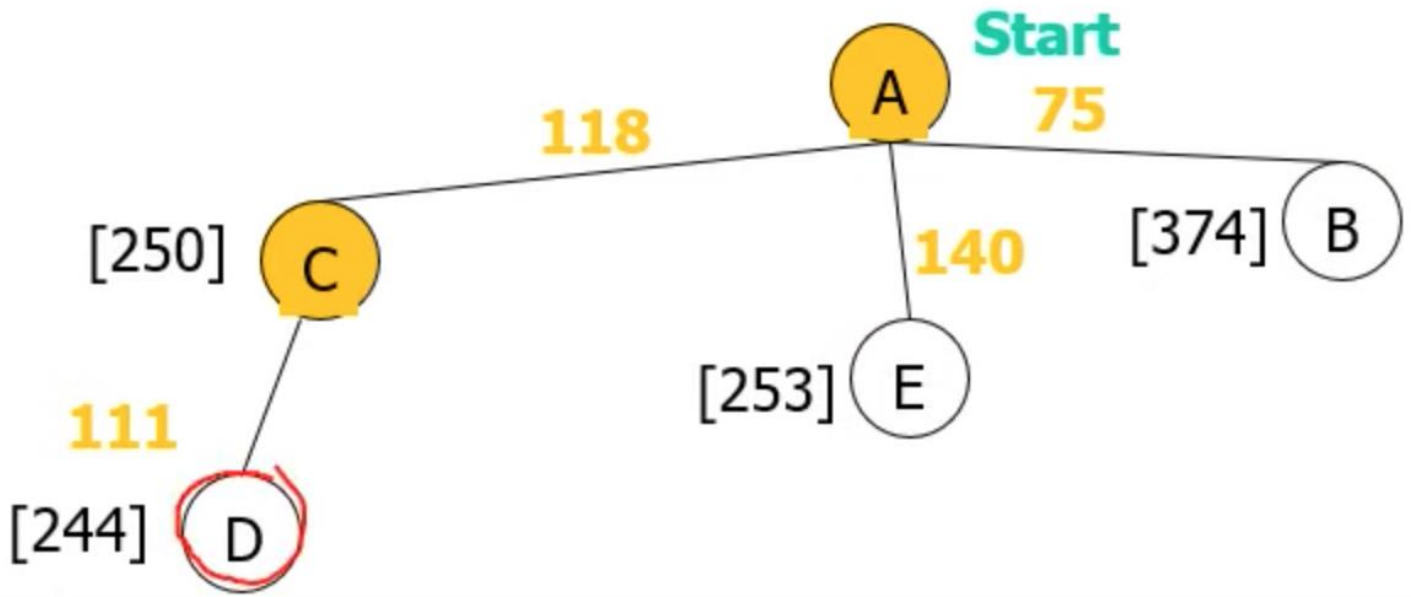
$f(n) = h(n)$ = straight-line distance heuristic

Greedy Search: Tree Search



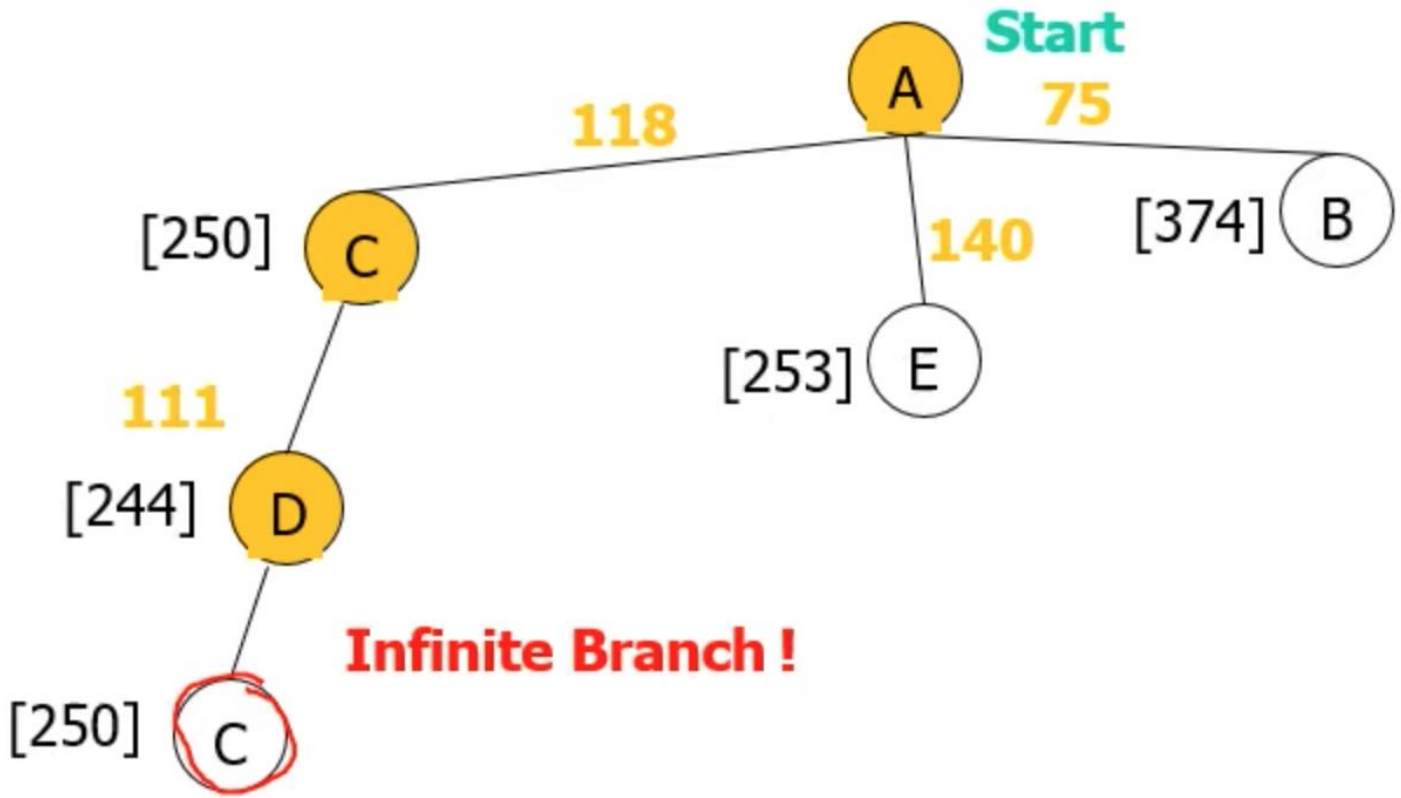
State	Heuristic: h(n)
A	366
B	374
** C	**250**
D	244
E	253
F	178
G	193
H	98
I	0

Greedy Search: Tree Search



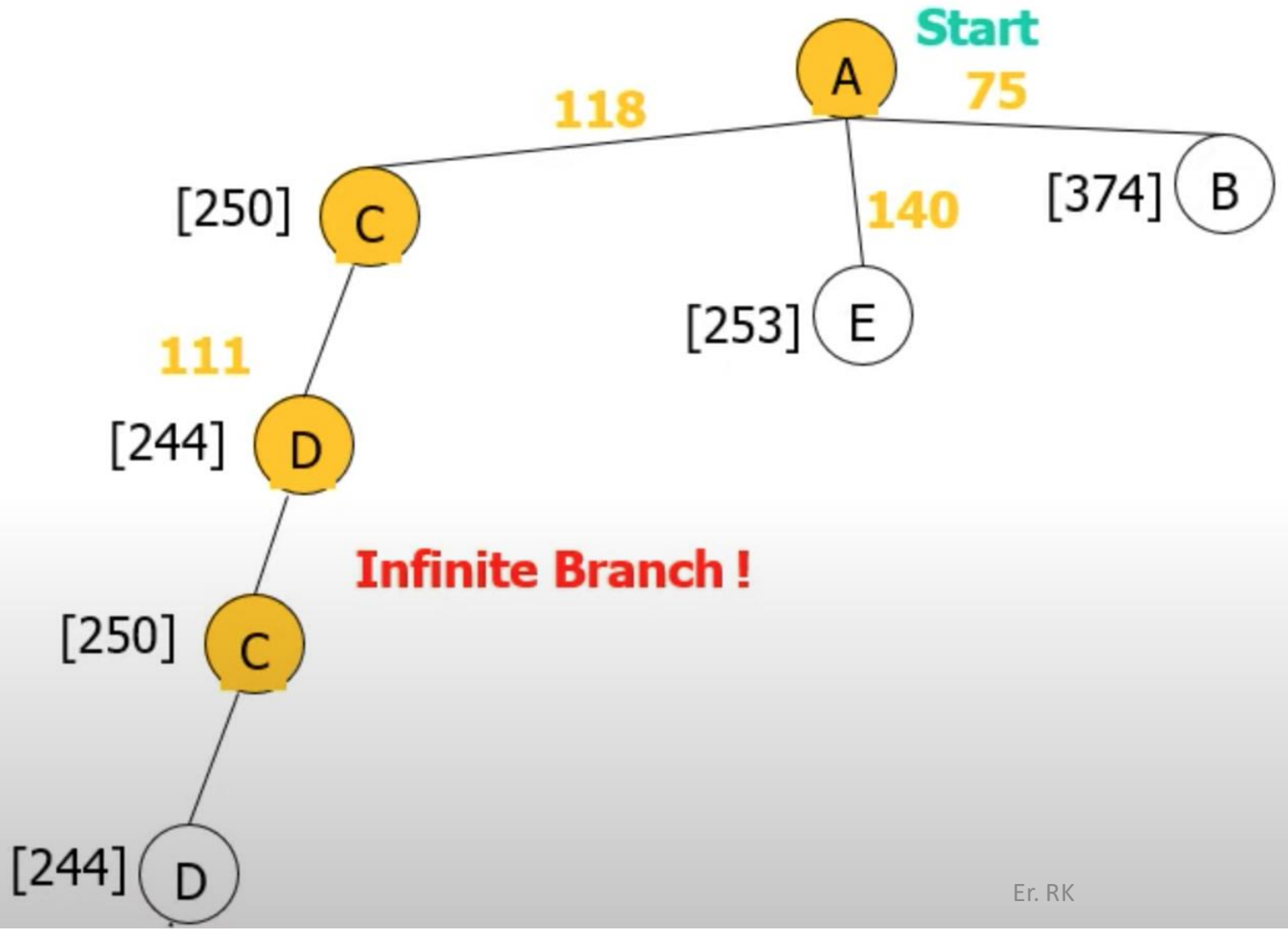
State	Heuristic: h(n)
A	366
B	374
** C	**250**
D	244
E	253
F	178
G	193
H	98
I	0

Greedy Search: Tree Search



State	Heuristic: h(n)
A	366
B	374
** C	**250**
D	244
E	253
F	178
G	193
H	98
I	0

Greedy Search: Tree Search



State	Heuristic: h(n)
A	366
B	374
** C	**250**
D	244
E	253
F	178
G	193
H	98
I	0

Greedy best-first search…

- Greedy best-first search can start down an infinite path and never return to try other possibilities, it is incomplete
- Because of its greediness the search makes choices that can lead to a dead end; then one backs up in the search tree to the deepest unexpanded node
- Greedy best-first search resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead end
- The quality of the heuristic function determines the practical usability of greedy search

Greedy best-first search…

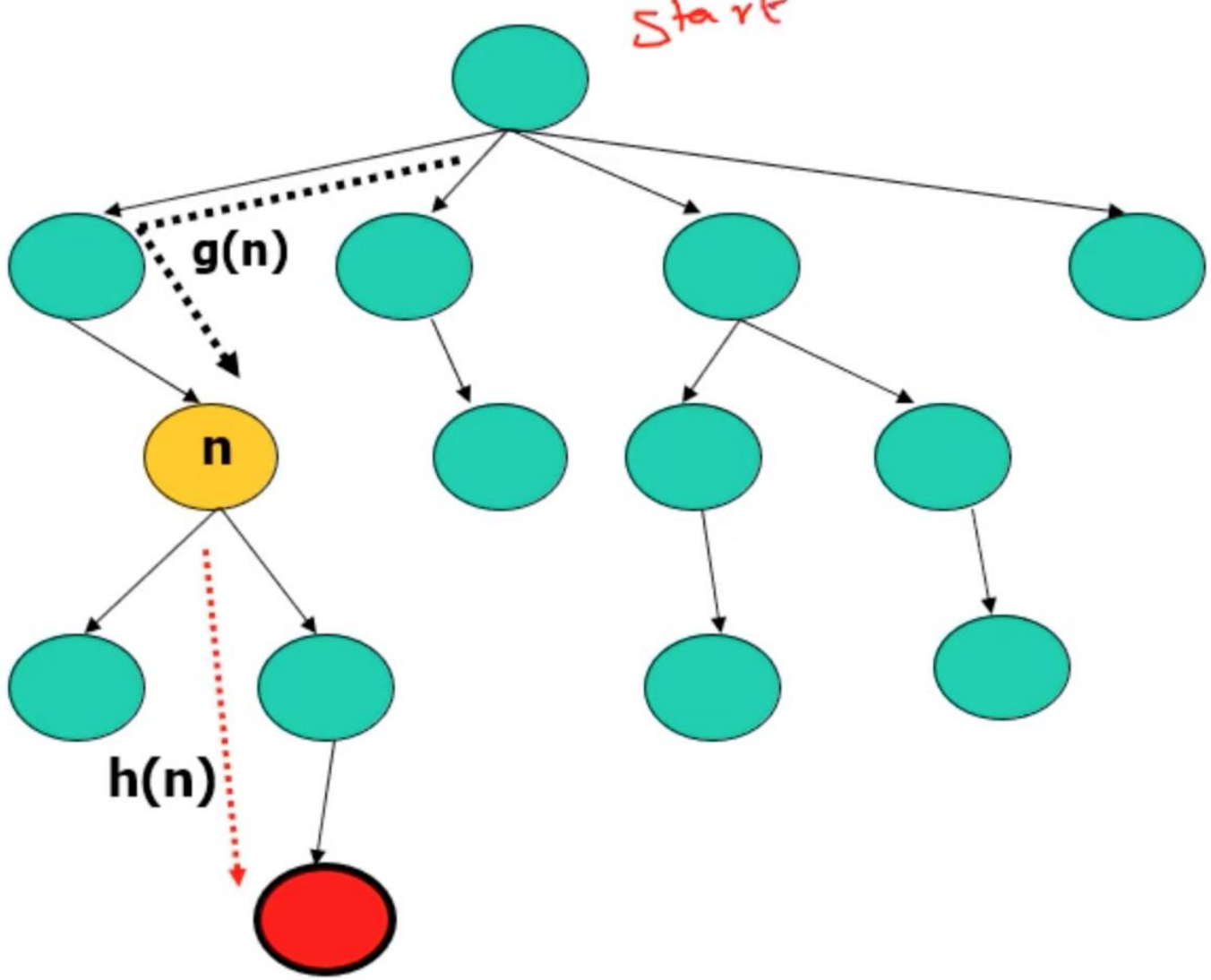
- Greedy search is not optimal
- Greedy search is incomplete without systematic checking of repeated states.
- In the worst case, the Time and Space Complexity of Greedy Search are both $O(b^m)$,
- Where
  - b is the branching factor and
  - m the maximum path length

A* Search

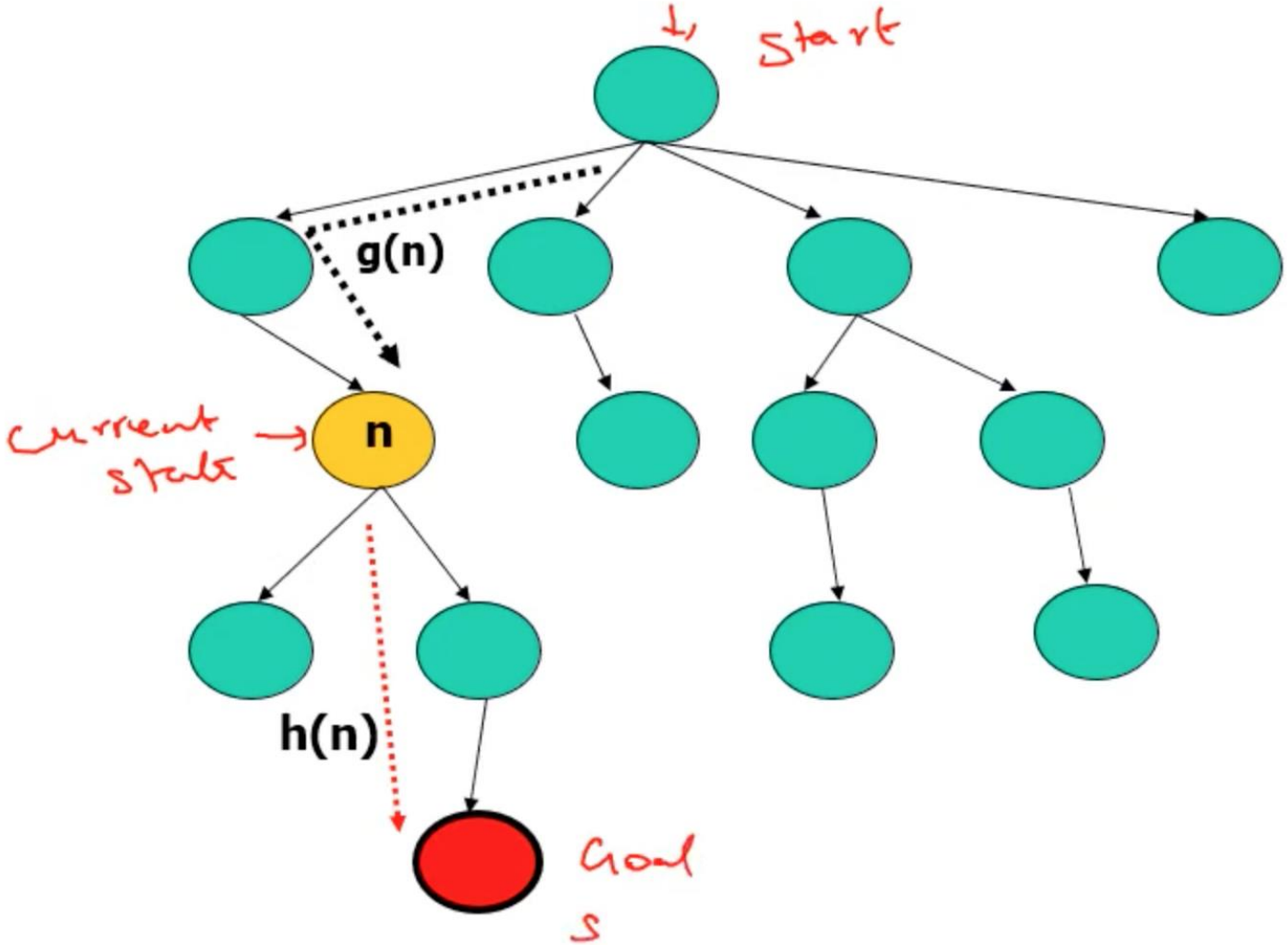
- Greedy Best First Search minimizes a heuristic h(n) which is an estimated cost from a current state n to the goal state.
- Greedy Best First Search is efficient but it is not optimal and not complete.
- Uniform Cost Search minimizes the cost g(n) from the initial state to current state n.
- Uniform Cost Search is optimal and complete but not efficient.
- **A* Search**: Combine Greedy Best First Search and Uniform Cost Search to get an efficient algorithm which is complete and optimal.

A* Search...

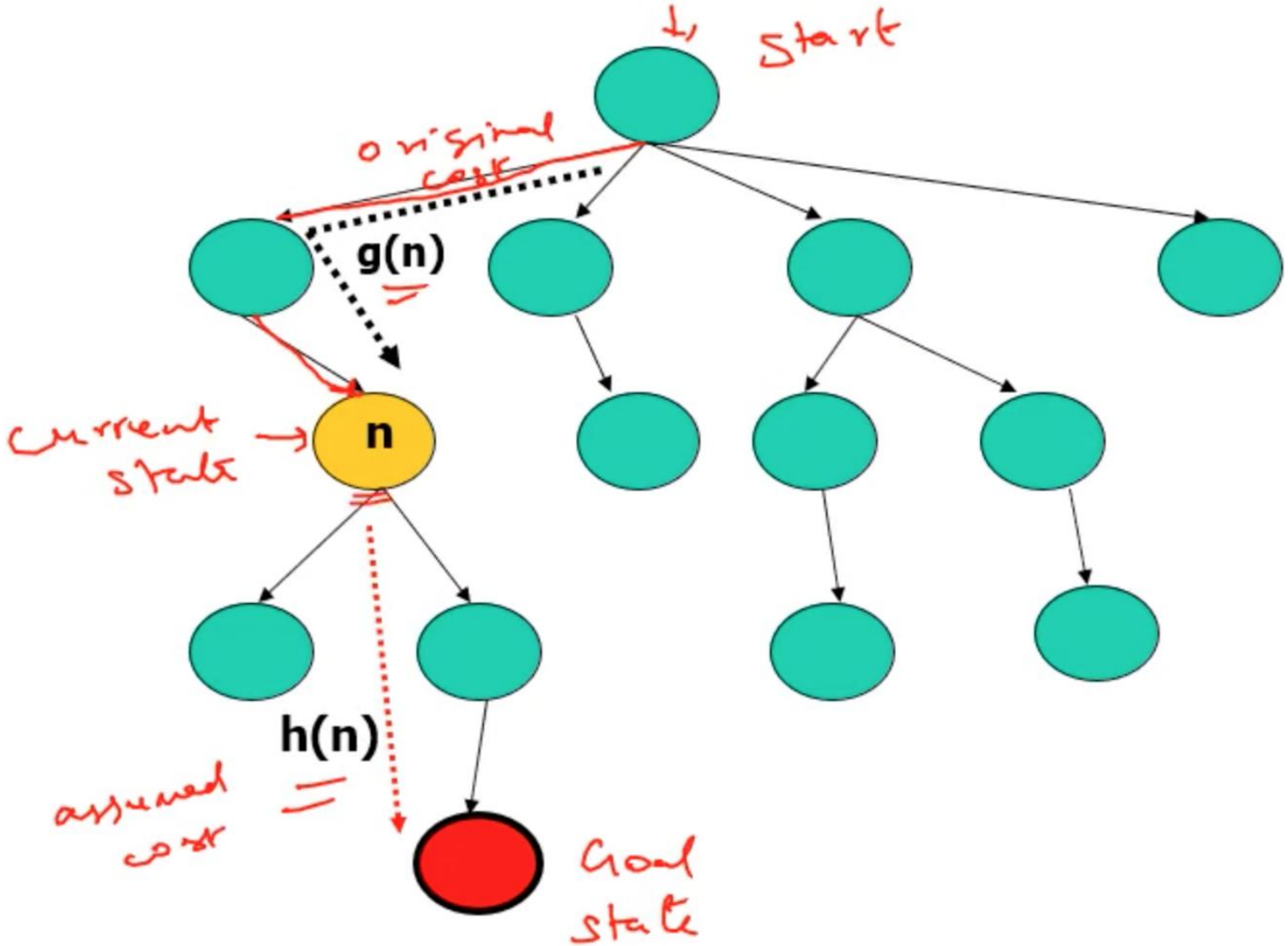
- **A* search** evaluates nodes by combining g(n), the cost to reach the node and h(n), the cost to get from the node to the goal.
- **f(n) = g(n) + h(n)**
- f(n) is the evaluation function which gives the cheapest solution cost
- g(n) is the exact cost to reach node n from the initial state.
- h(n) is an estimation of the assumed cost from current state (n) to reach the goal.



$$f(n) = g(n)+h(n)$$

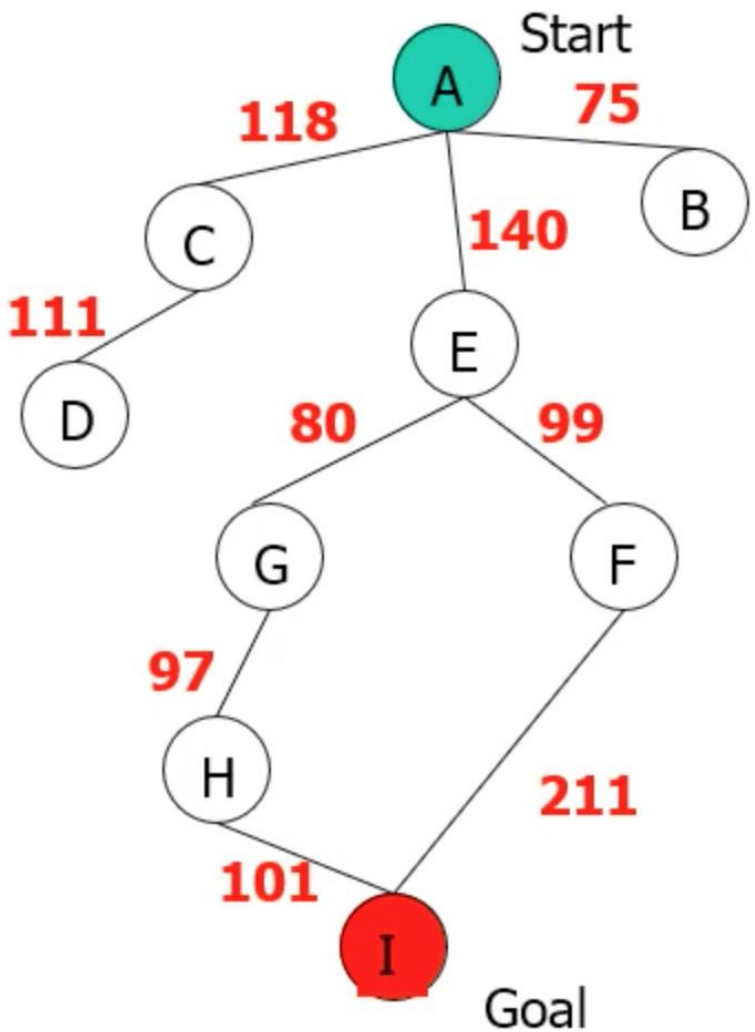


f(n) = g(n)+h(n)


Start
original cost
g(n)
current state
n
h(n)
assumed cost
Goal state

$$f(n) = g(n)+h(n)$$

A* Search

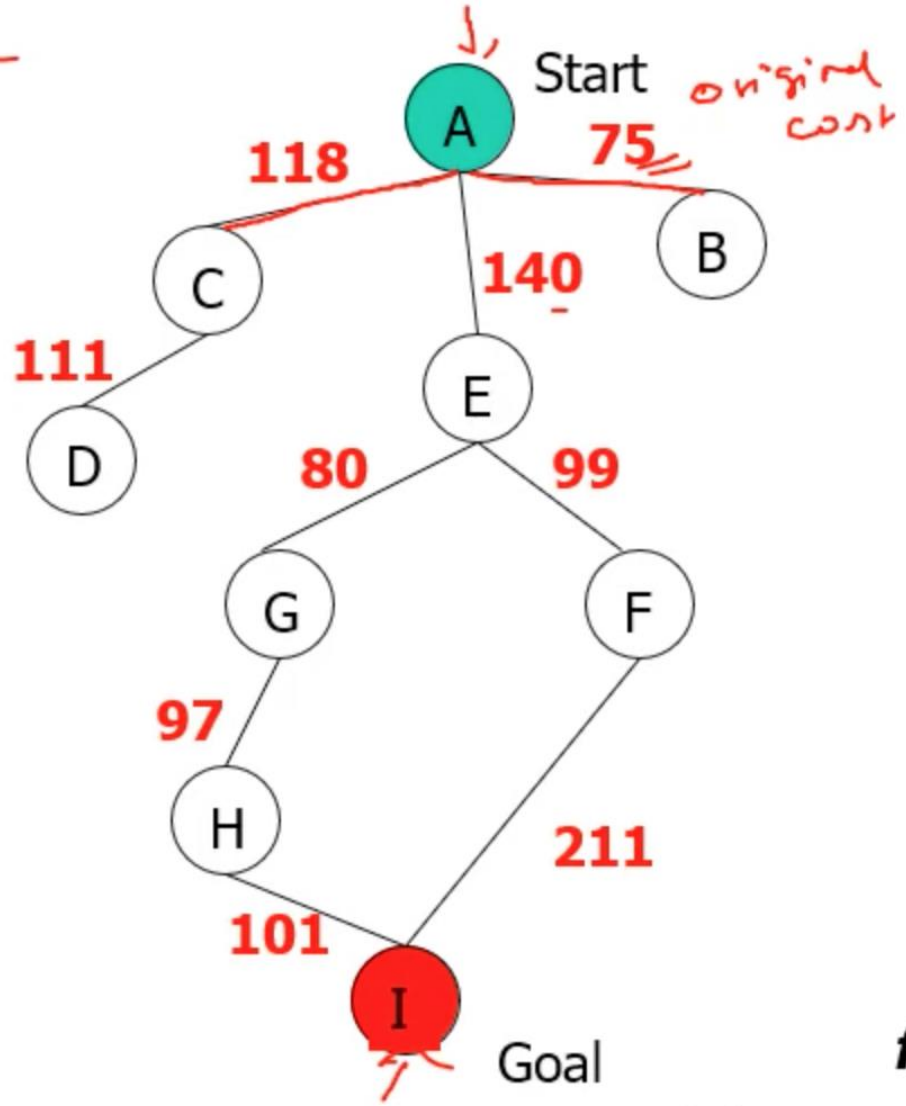


State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

$$f(n) = g(n) + h(n)$$

g(n):exact cost to reach node *n* from the initial state.

A* Search



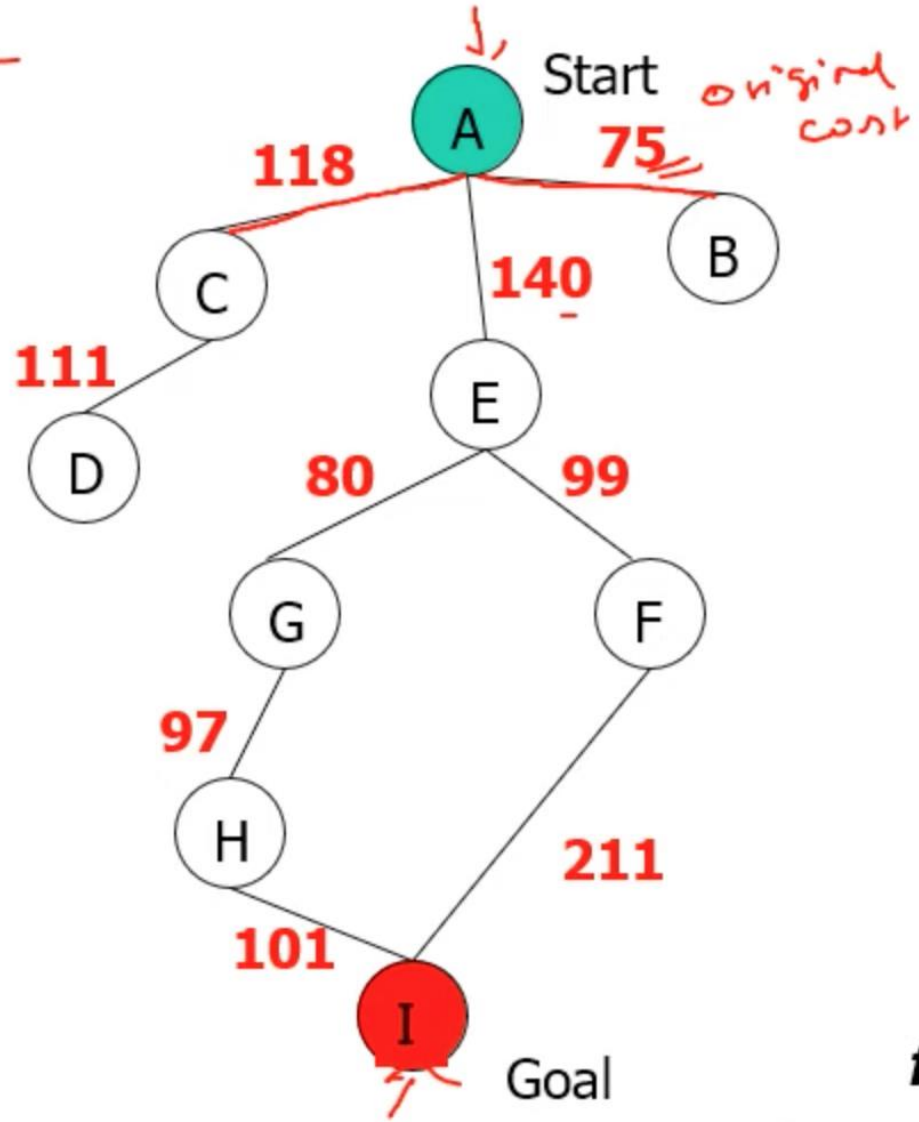
State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

$$f(n) = g(n) + h(n)$$

g(n):exact cost to reach node *n* from the initial state.

A* Search

Graph



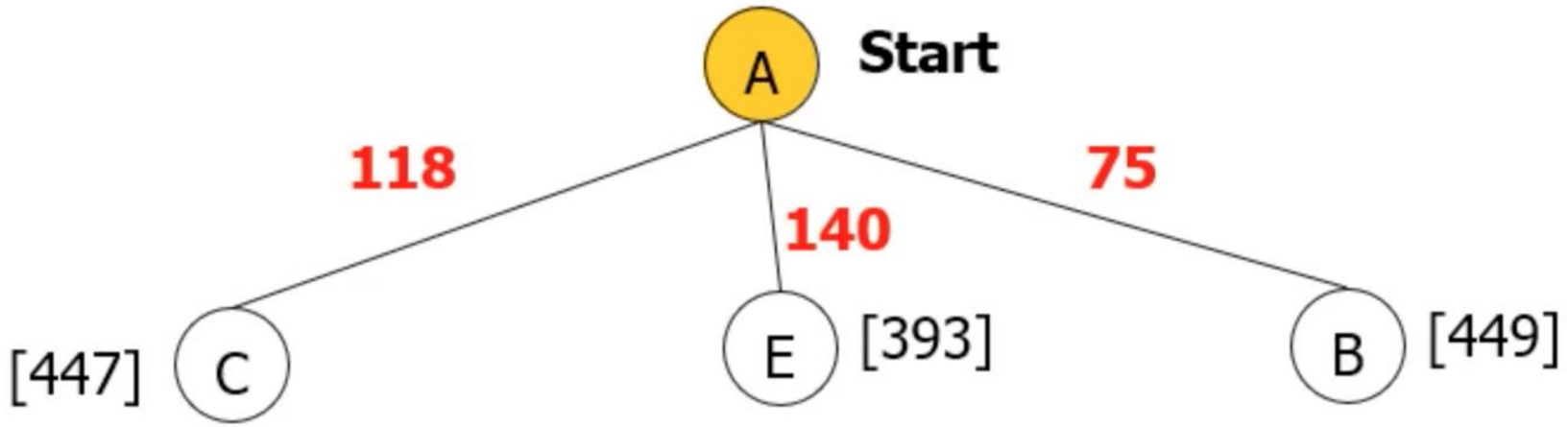
Heuristic Table

State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

$$f(n) = g(n) + h(n)$$

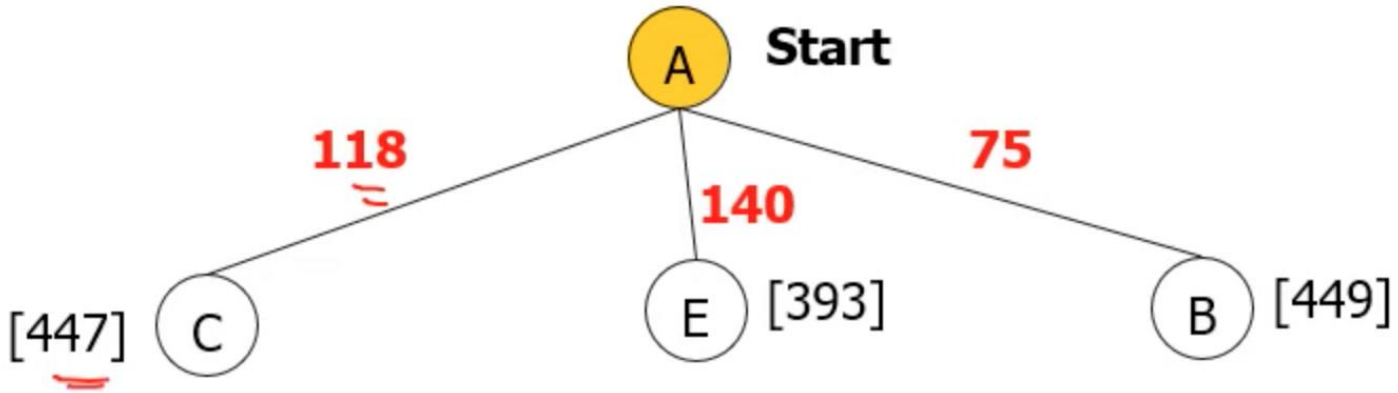
g(n):exact cost to reach node *n* from the initial state.

A* Search: Tree Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

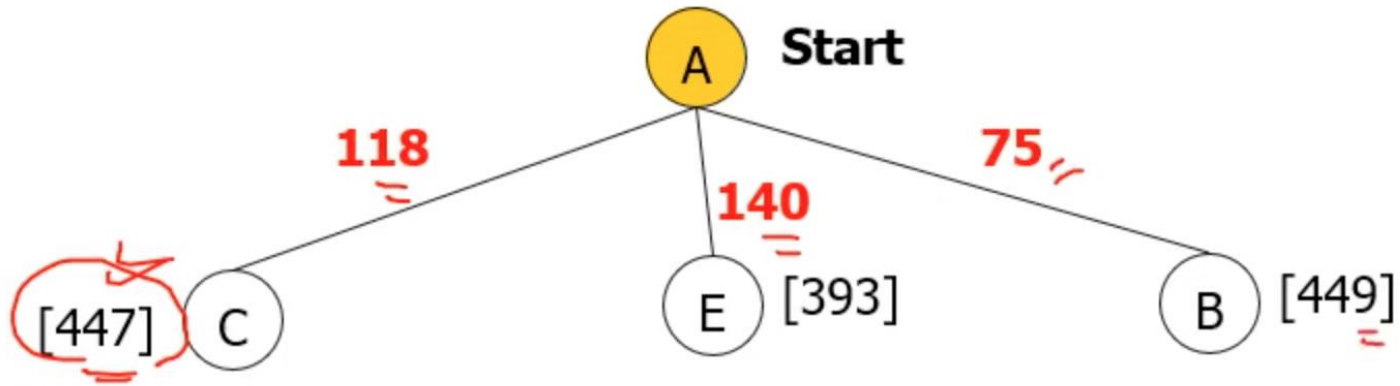
A* Search: Tree Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

$f(C) = g(C) + h(C)$
$= 118 + 329$
$= 447$

A* Search: Tree Search



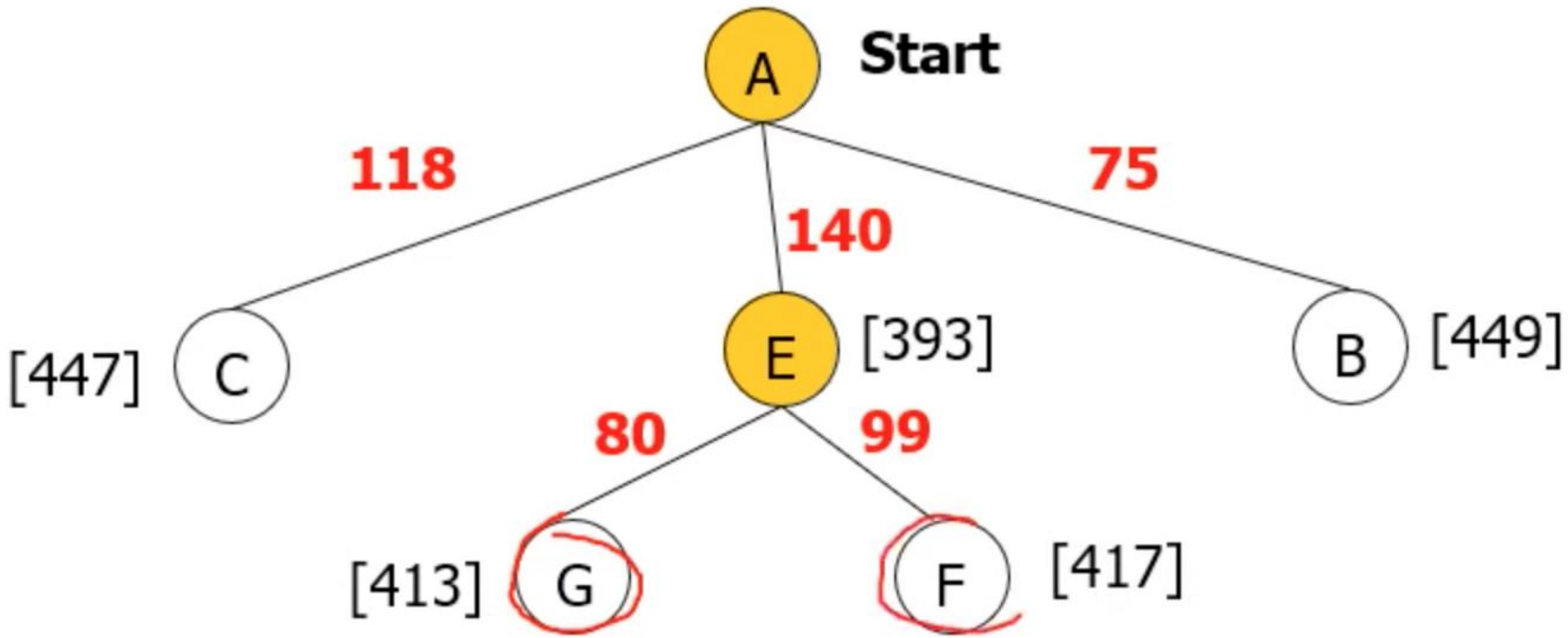
State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

$f(C) = g(C) + h(C)$
$= 118 + 329$
$= 447$

$f(B) = g(B) + h(B)$
$= 75 + 374$
$= 449$

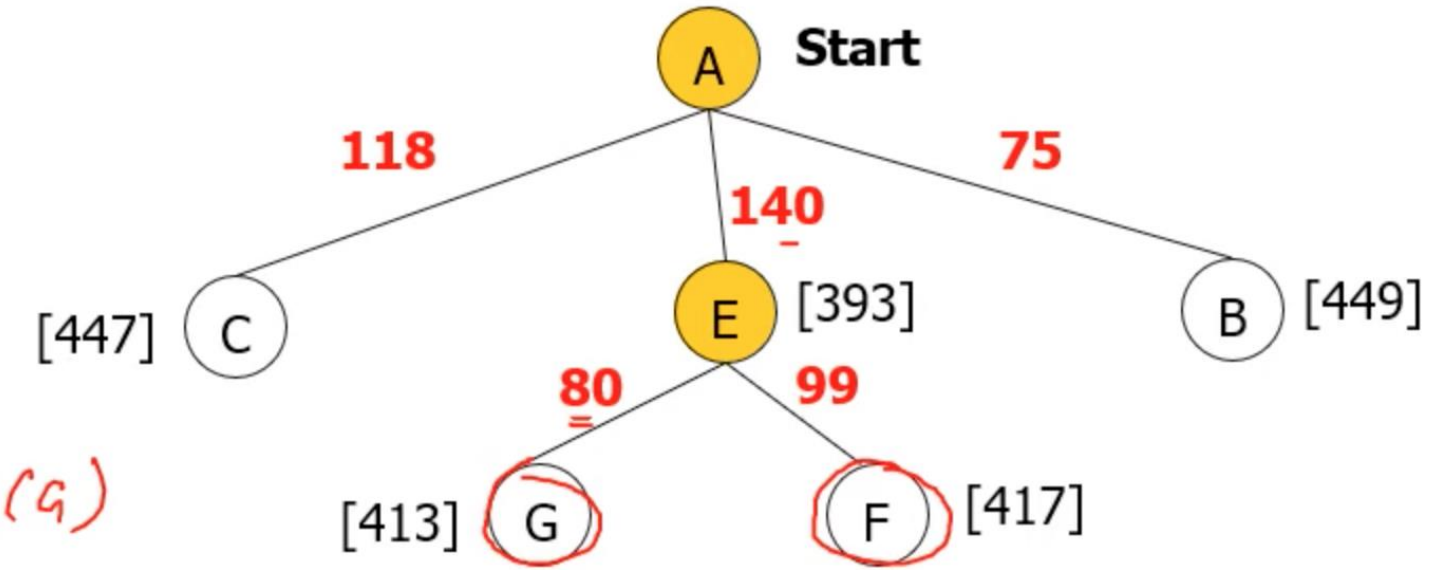
$f(E) = g(E) + h(E)$
$= 140 + 253$
$= 393$

A* Search: Tree Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

A* Search: Tree Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

$f(G) = g(G) + h(G)$

$= (140 + 80) + 193$

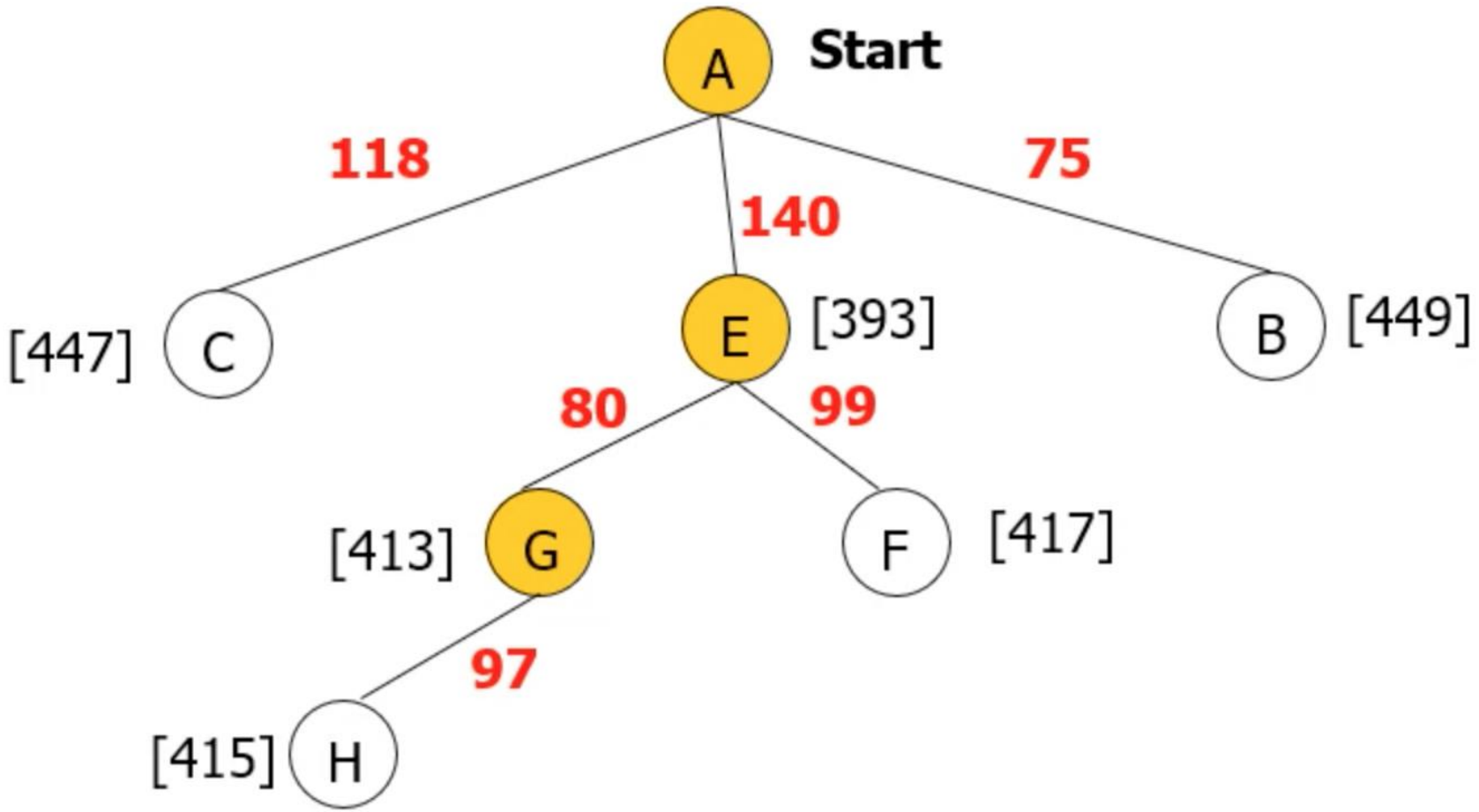
$= 413$

$f(F) = g(F) + h(F)$

$= (140 + 99) + 178$

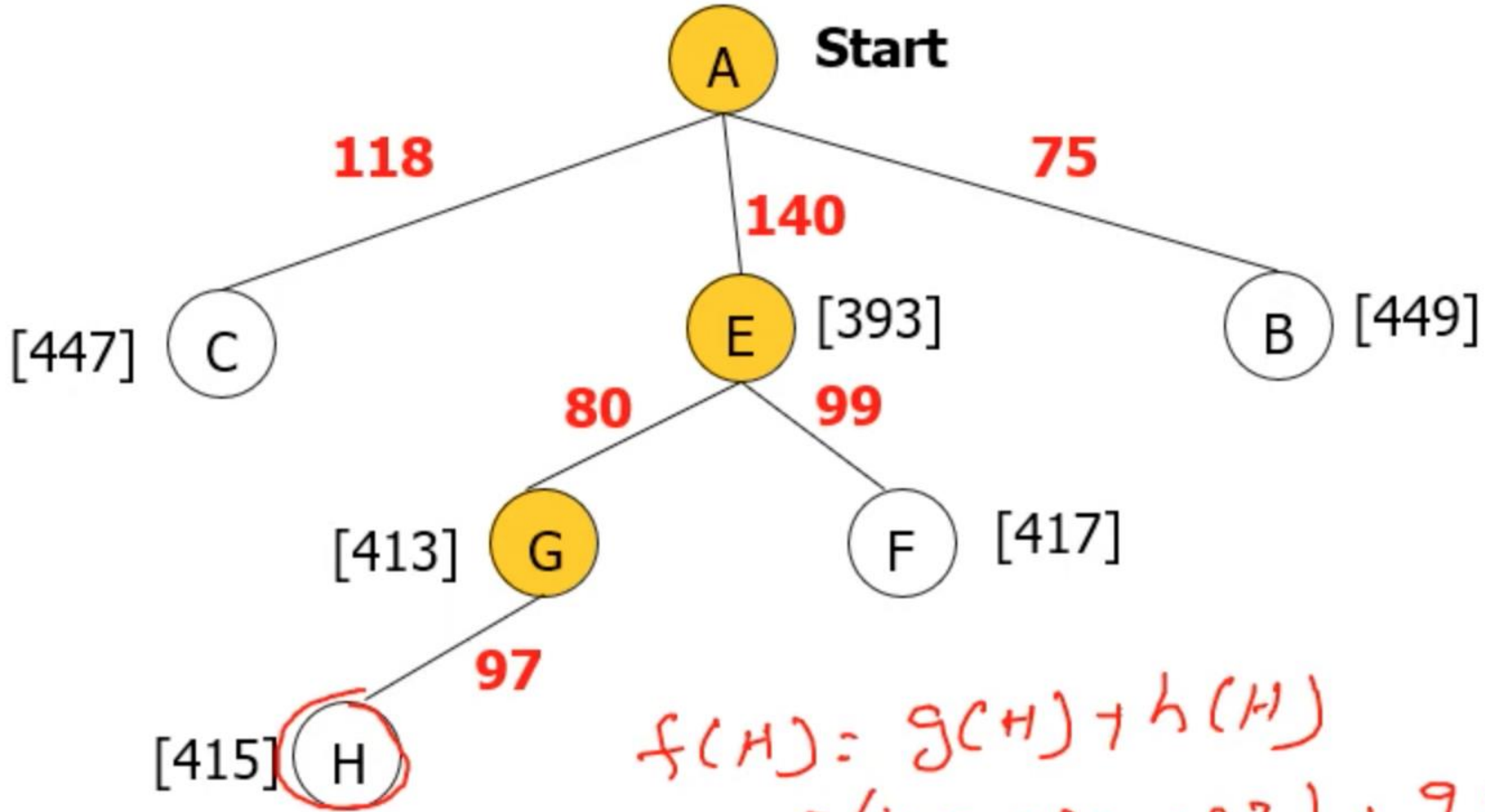
$= 417$

A* Search: Tree Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

A* Search: Tree Search



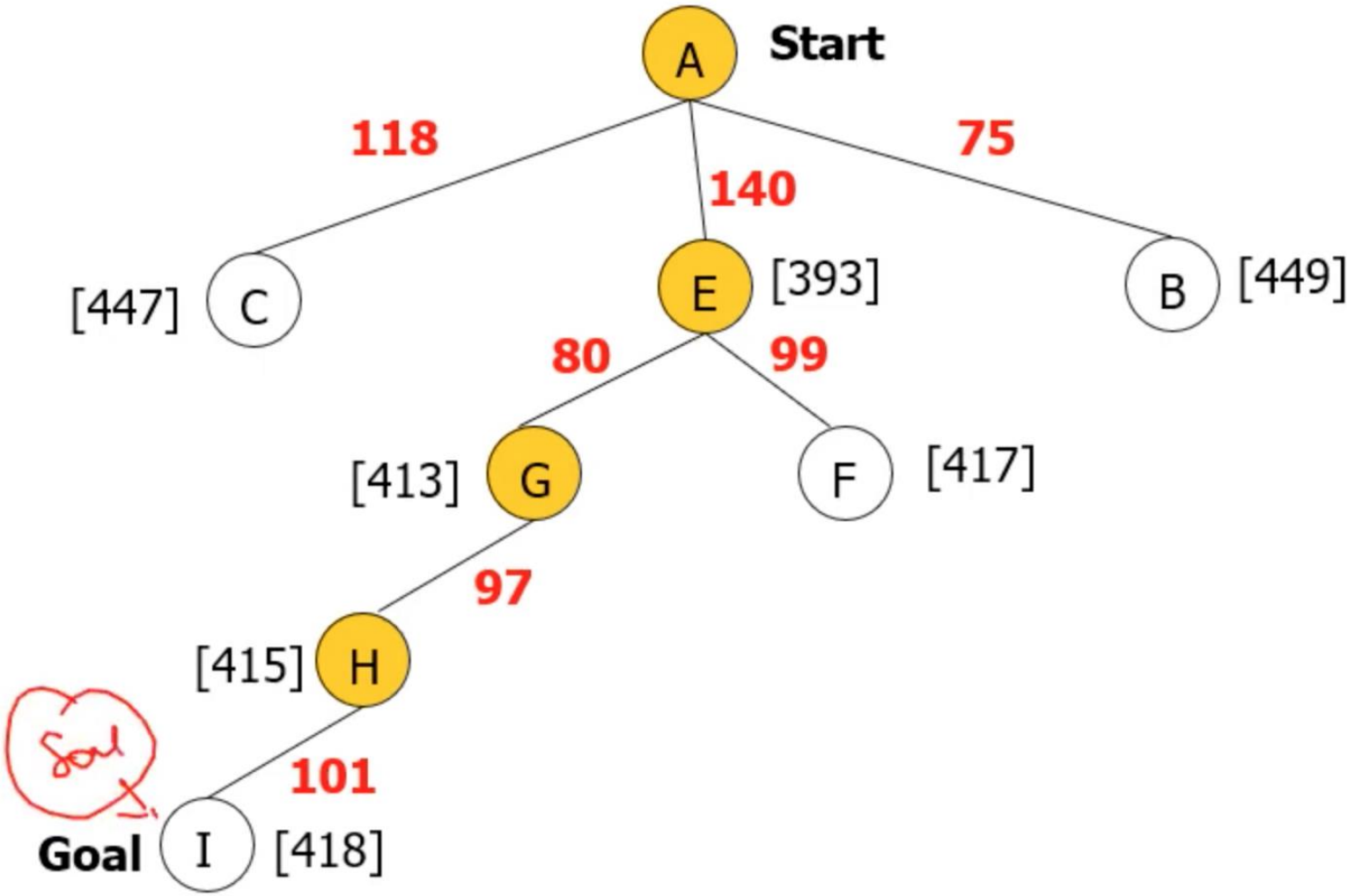
State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

$f(H) = g(H) + h(H)$

$= (140 + 80 + 97) + 98$

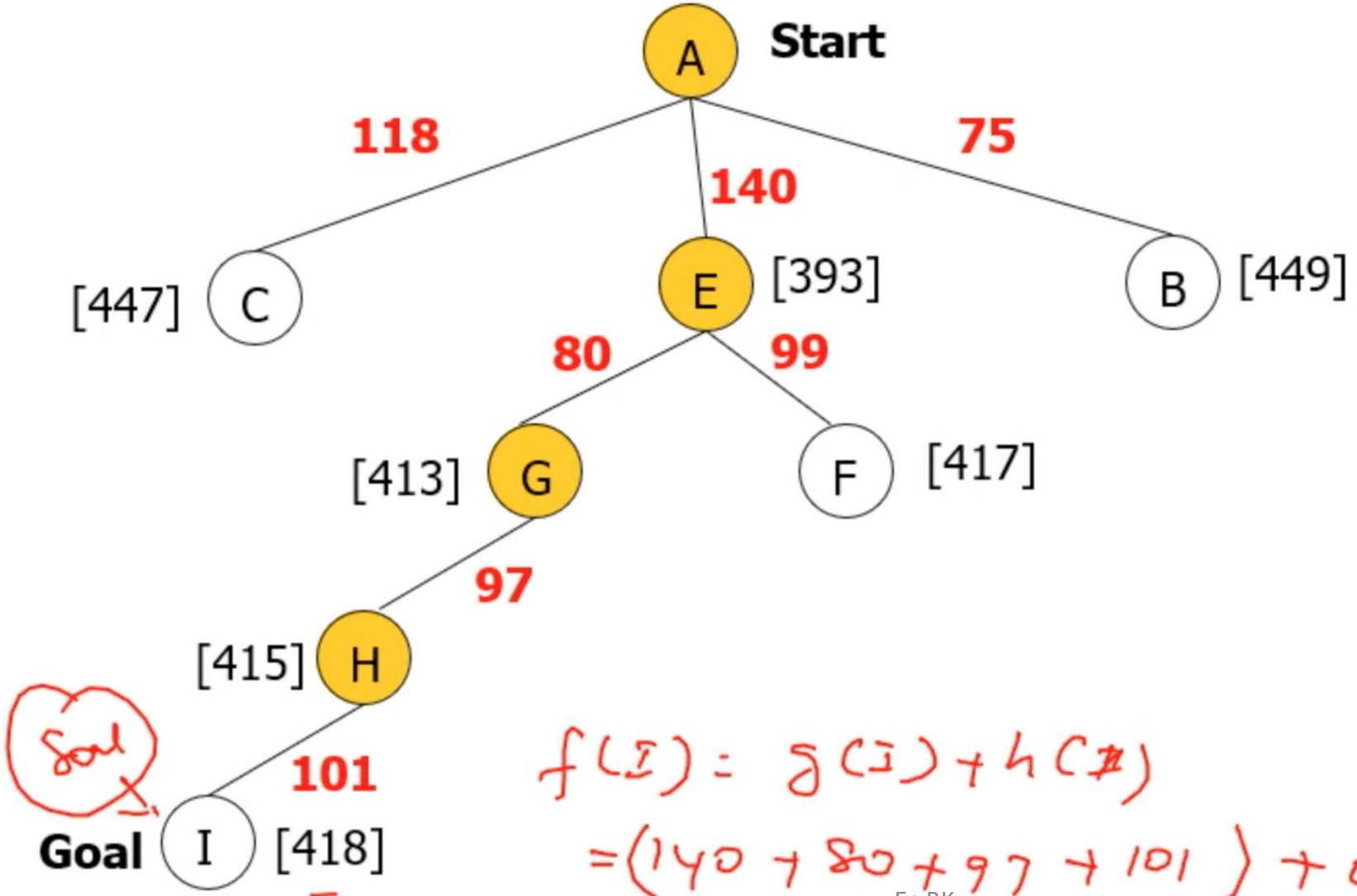
$= 415$

A* Search: Tree Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

A* Search: Tree Search



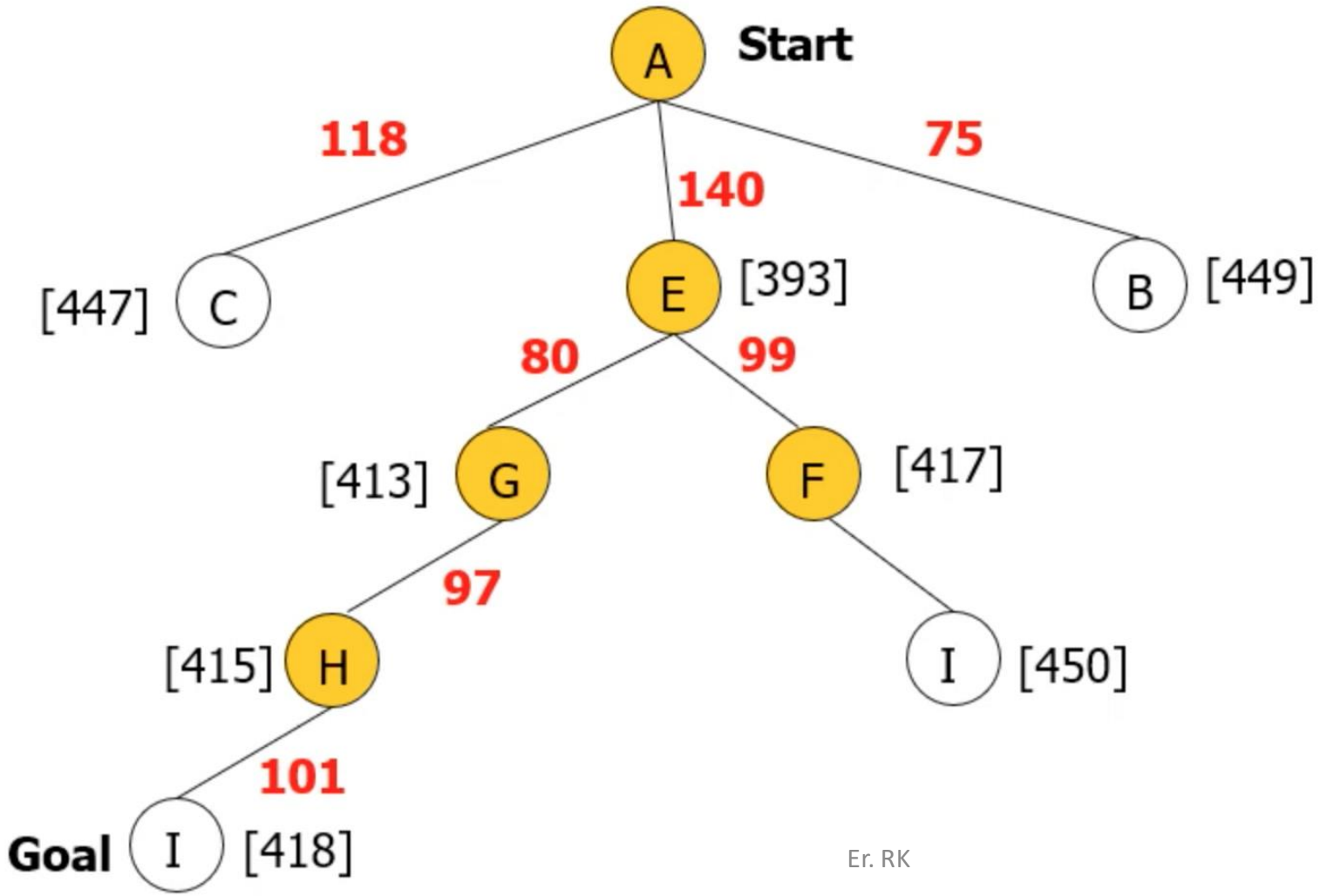
State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

$f(I) = g(I) + h(I)$

$= (140 + 80 + 97 + 101) + 0$

$= 418$

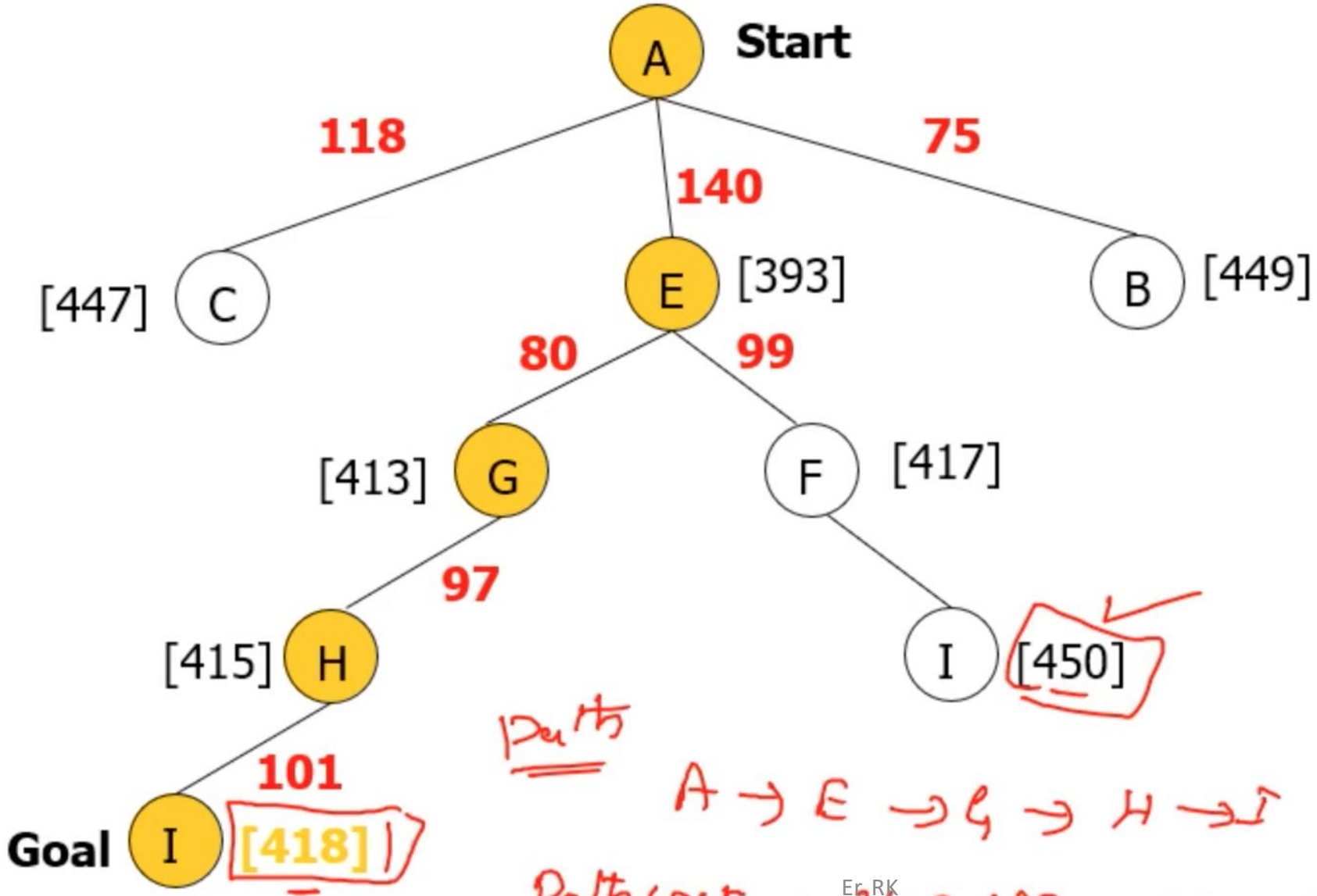
A* Search: Tree Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

A* Search: Tree Search

State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0



Paths

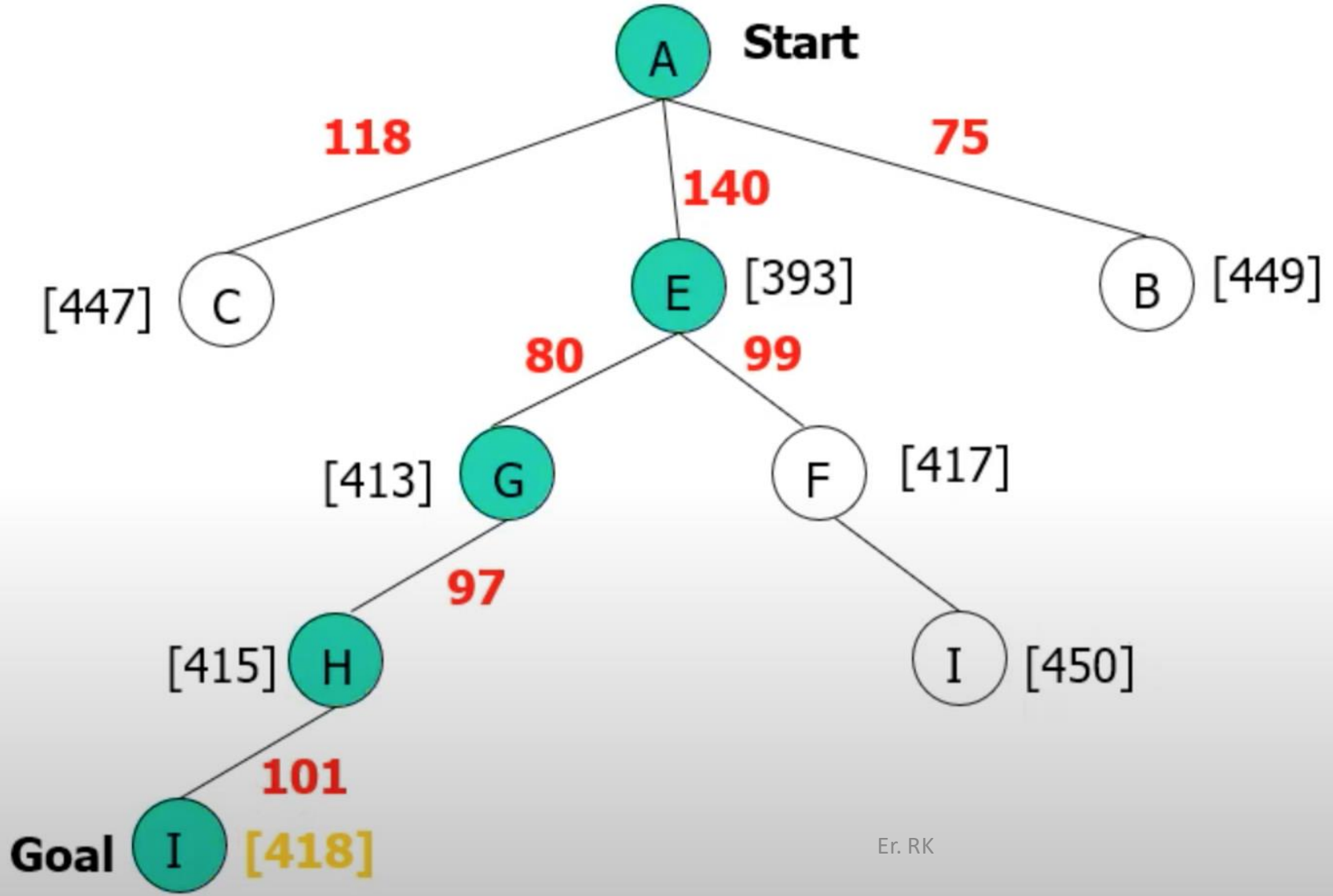
A → E → G → H → I

Path cost = 140 + 80 + 97 + 101

Answer

A* Search: Tree Search

State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0



Continue…

- A* with f() not Admissible when h() overestimates the cost to reach the goal state

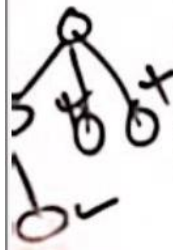
SIMPLE HILL CLIBING SEARCH ALGORITHM

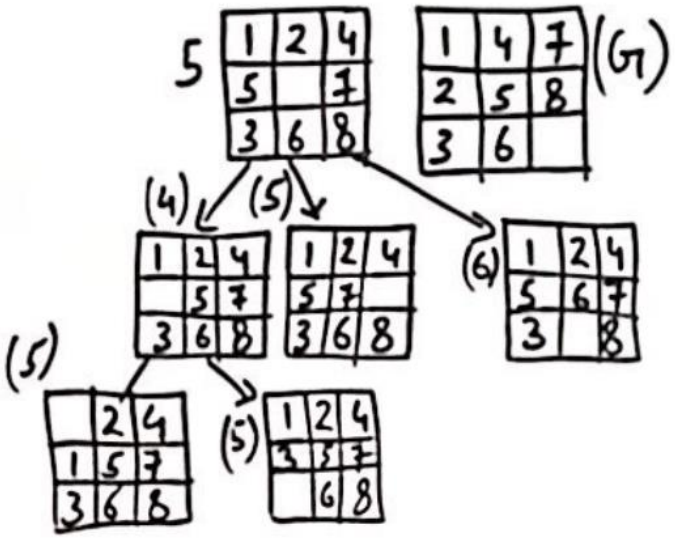
Simple Hill Climbing Algorithm (Local search Algo, greedy approach, No backtrac

Evaluate the initial state

Loop Until a solution is found or there are no operators left

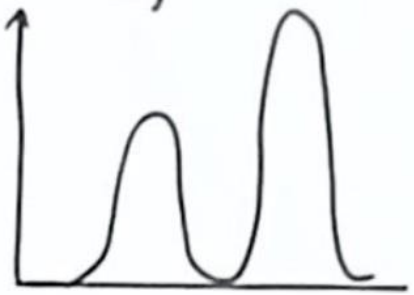
- Select and Apply a new operator
- Evaluate the new state:
  - if goal then quit
  - if better than current state then it is new current state.



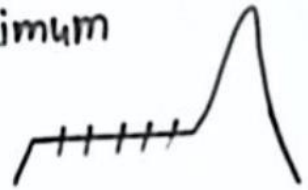


Problems in Hill climbing

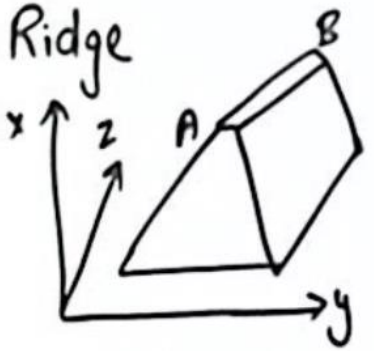
1) Local Maximum



2) Plateau / Flat Maximum

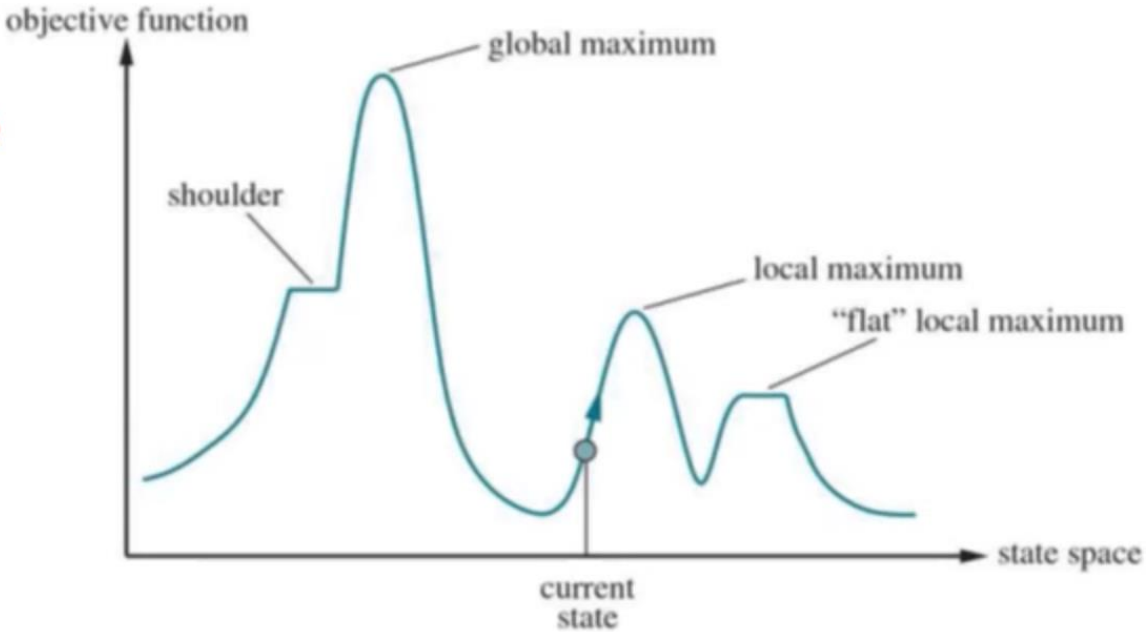


3) Ridge



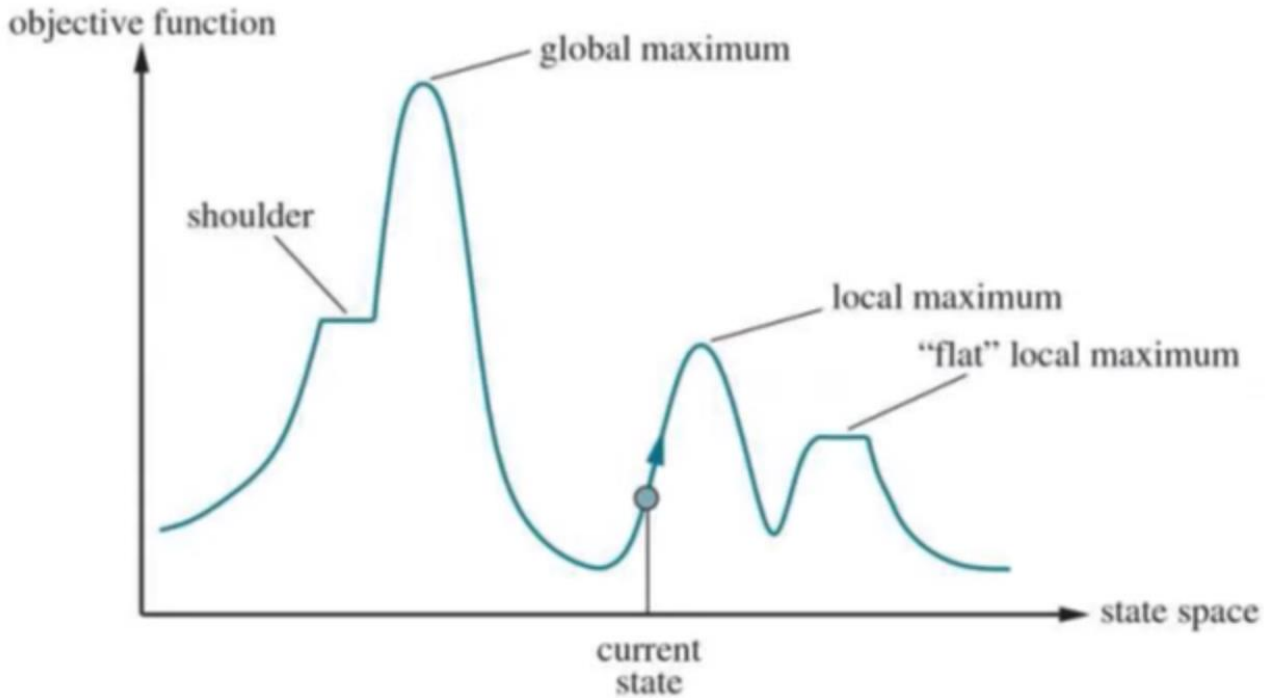
Hill-climbing Search Algorithm

- Hill climbing algorithm is a Heuristic search algorithm which continuously moves in the direction of increasing value to **find the peak of the mountain or best solution to the problem**.
- It keeps track of one current state and on each iteration moves to the neighboring state with highest value—that is, it heads in the direction that provides the **steepest ascent**.



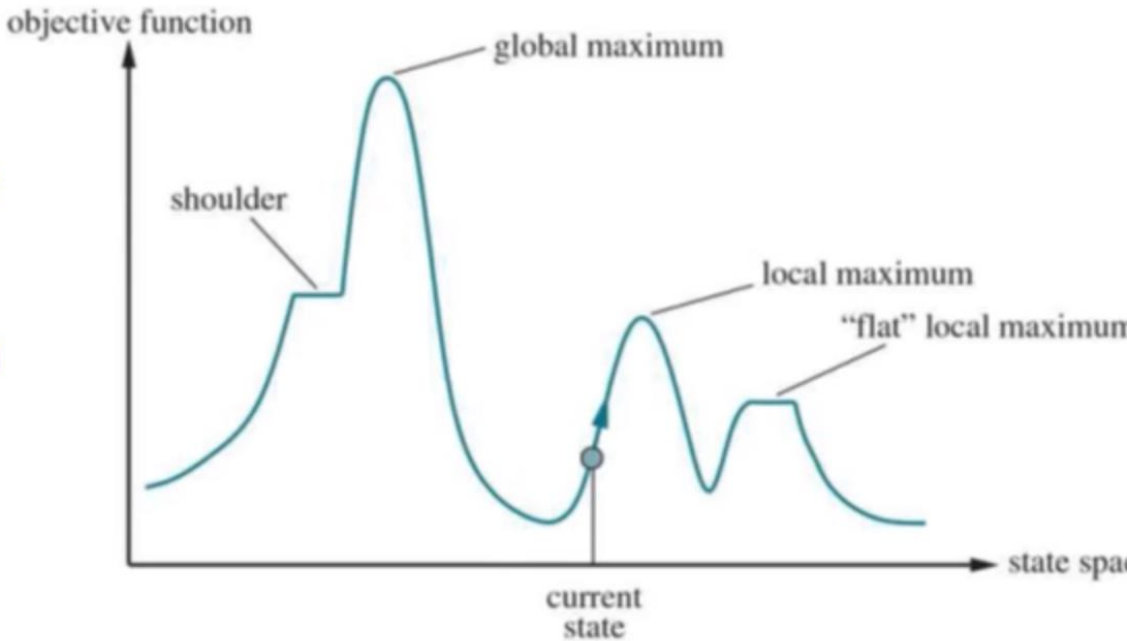
Hill-climbing Search Algorithm…

- In this algorithm, when it reaches a peak value where no neighbor has a higher value, then it terminates.
- It is also called greedy local search as it only searches its good immediate neighbor state and not beyond that.
- Hill Climbing is mostly used when a good heuristic is available.



Hill-climbing Search Algorithm

- Different regions in the state space landscape:
- **Local Maximum** is a state which is better than its neighbor states, but there is also another state which is higher than it.
- **Global Maximum** is the best possible state of state space landscape. It has the highest value of objective function.
- **Current state** is a state in a landscape diagram where an agent is currently present.
- **Flat local maximum** is a flat space in the landscape where all the neighbor states of current states have the same value.
- **Shoulder** is a plateau region which has an uphill edge.



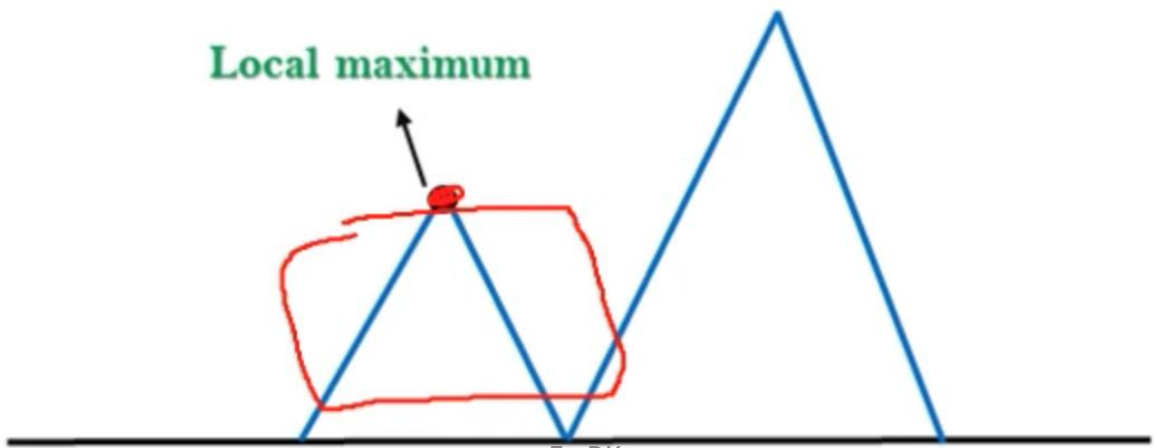
Hill-climbing Search Algorithm...

- The hill-climbing search algorithm, which is the most basic local search technique.
- At each step the current node is replaced by the best neighbor.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    current ← problem.INITIAL
    while true do
        neighbor ← a highest-valued successor state of current
        if VALUE(neighbor) ≤ VALUE(current) then return current
        current ← neighbor
```

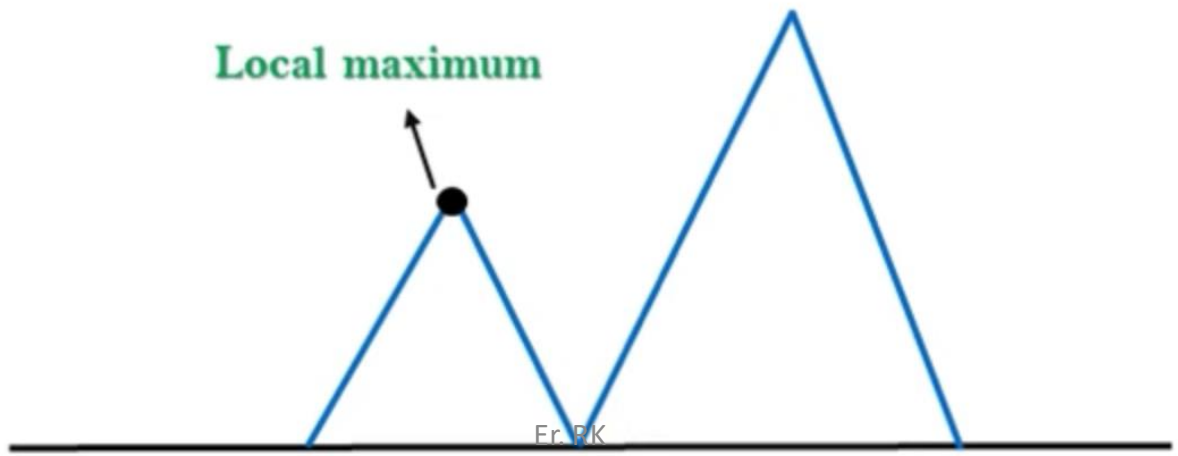
Problems in Hill Climbing Algorithm

- **1. Local Maximum:** A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.
- **Solution:** Backtracking technique can be a solution of the local maximum in state space landscape.
- Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



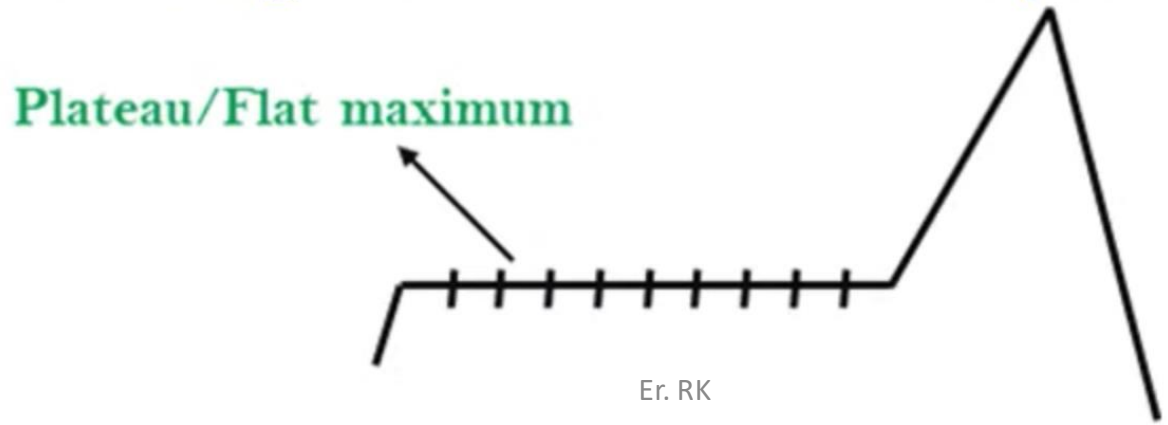
Problems in Hill Climbing Algorithm

- **1. Local Maximum:** A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.
- **Solution:** Backtracking technique can be a solution of the local maximum in state space landscape.
- Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



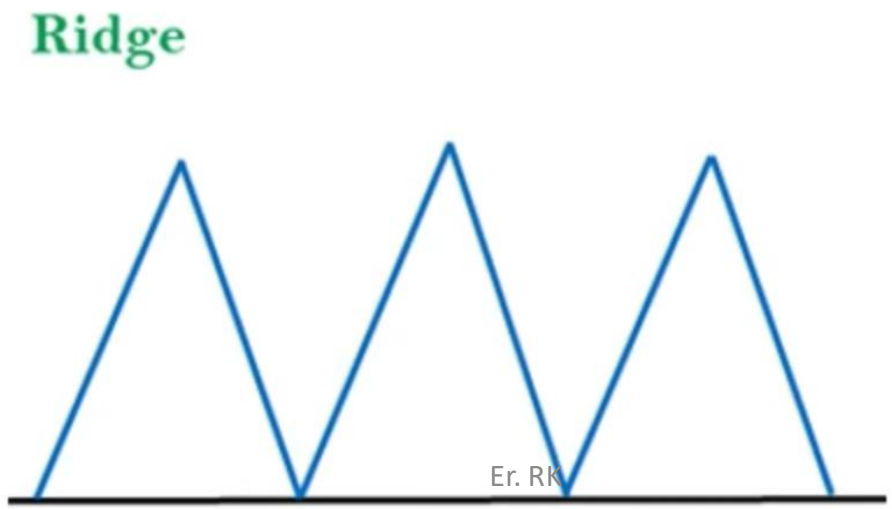
Problems in Hill Climbing Algorithm...

- **2. Plateau:** A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move.
- A hill-climbing search might be lost in the plateau area.
- **Solution:** The solution for the plateau is to take big steps while searching, to solve the problem.
- Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



Problems in Hill Climbing Algorithm...

- **3. Ridges:** A ridge is a special form of the local maximum.
- It has an area, which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.
- **Solution:** With the use of bidirectional search, or by moving in different directions, we can improve this problem.



Physical Annealing

- The Simulated Annealing algorithm is based upon Physical Annealing in real life.
- Physical Annealing is the process of heating up a material until it reaches an **annealing temperature** and then
- it will be **cooled down** slowly in order to change the material to a desired structure.
- When the material is hot, the molecular structure is weaker and is more susceptible to change.
- When the material cools down, the molecular structure is harder and is less susceptible to change.

Physical Annealing...

- Thermal Dynamics Equation calculates the probability that the Energy Magnitude will increase.

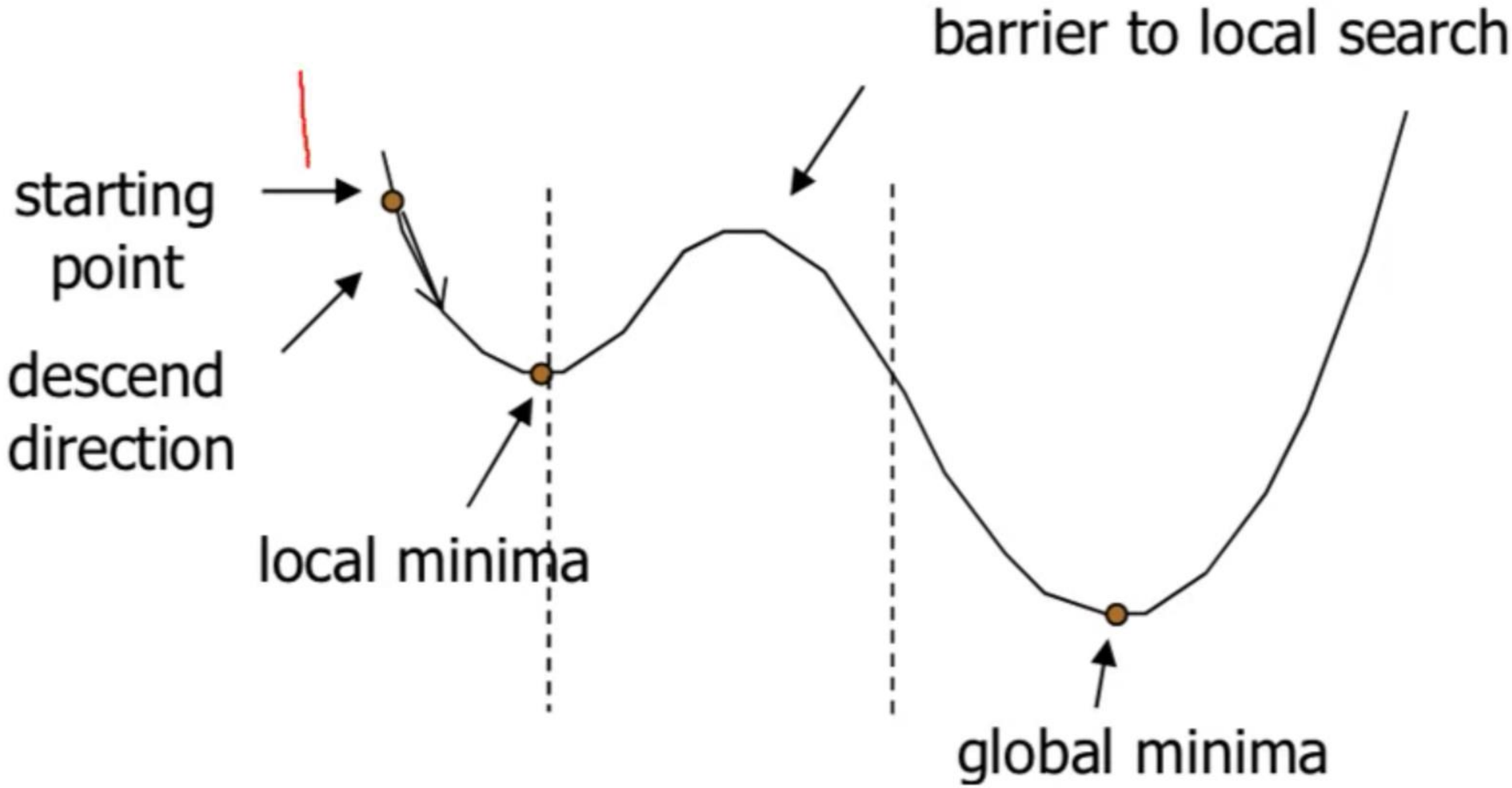
$$P(\Delta E) = e^{-\frac{\Delta E}{k*t}}$$

- Where ΔE - Energy Magnitude
- t - temperature
- k - Boltzmann constant.

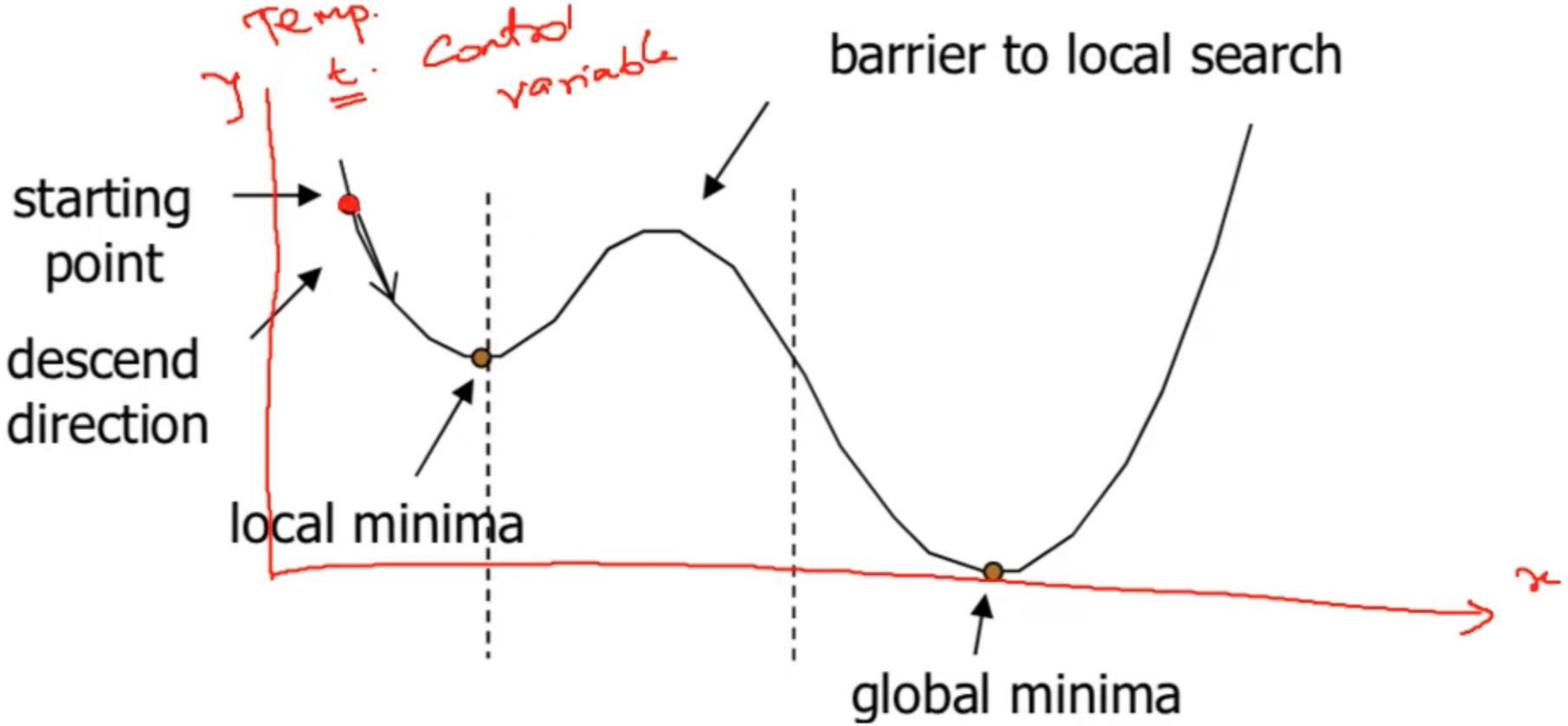
Simulated Annealing

- **Simulated Annealing** is a stochastic global search optimization algorithm and it is modified version of stochastic hill climbing.
- This algorithm appropriate for nonlinear objective functions where other local search algorithms do not operate well.
- **The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and**
- **then gradually reduce the intensity of the shaking (i.e., lower the temperature).**
- Simulated Annealing (SA) is very useful for situations where there are a lot of local minima.

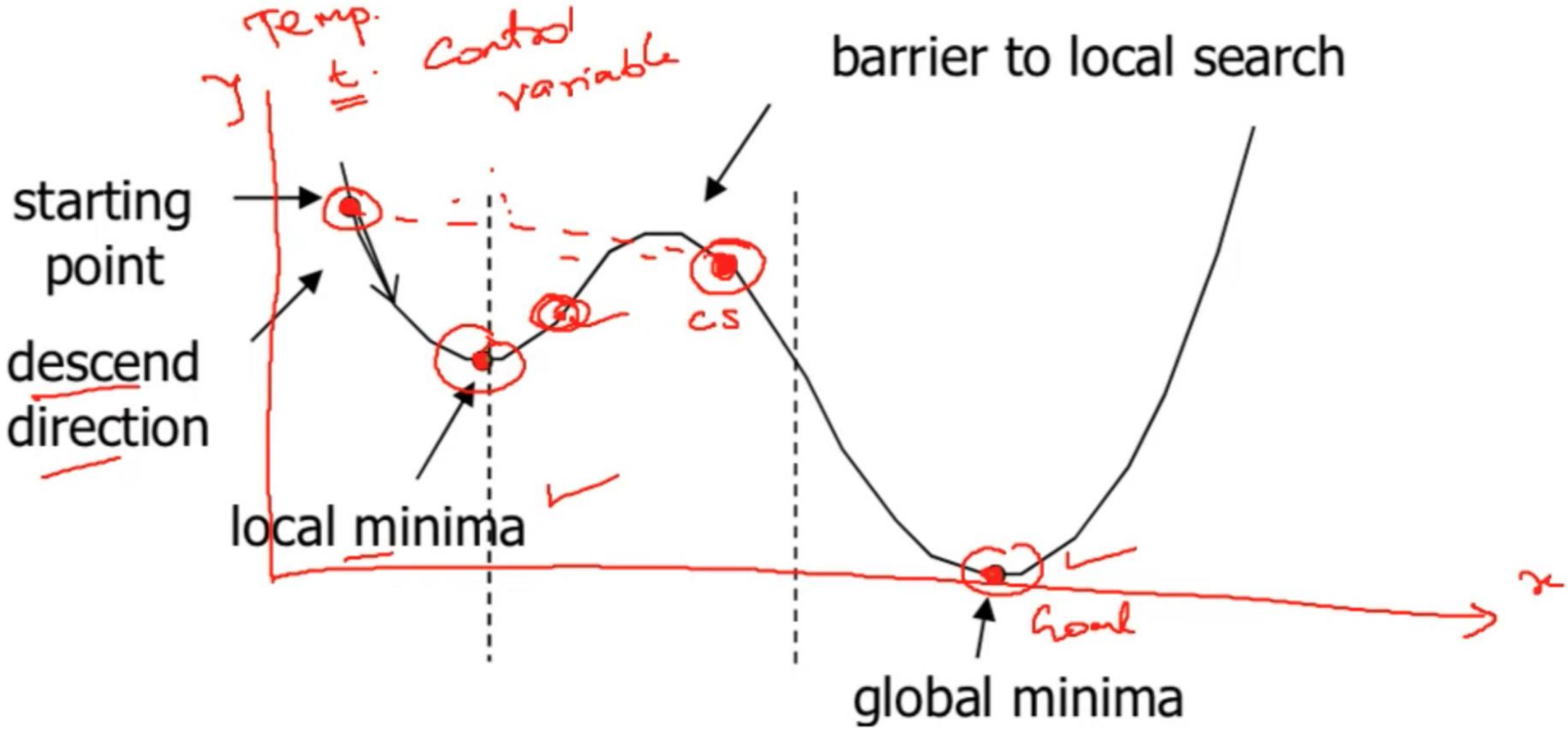
Simulated Annealing - State Space Diagram



Simulated Annealing - State Space Diagram

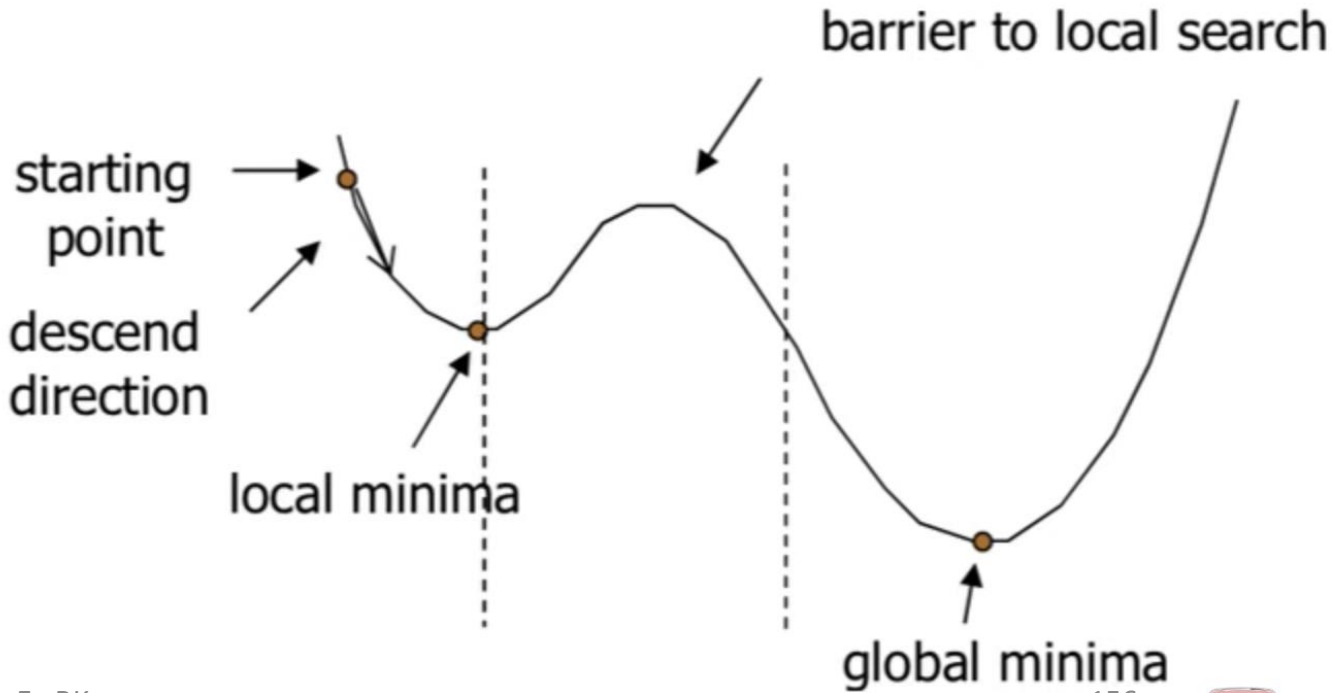


Simulated Annealing - State Space Diagram



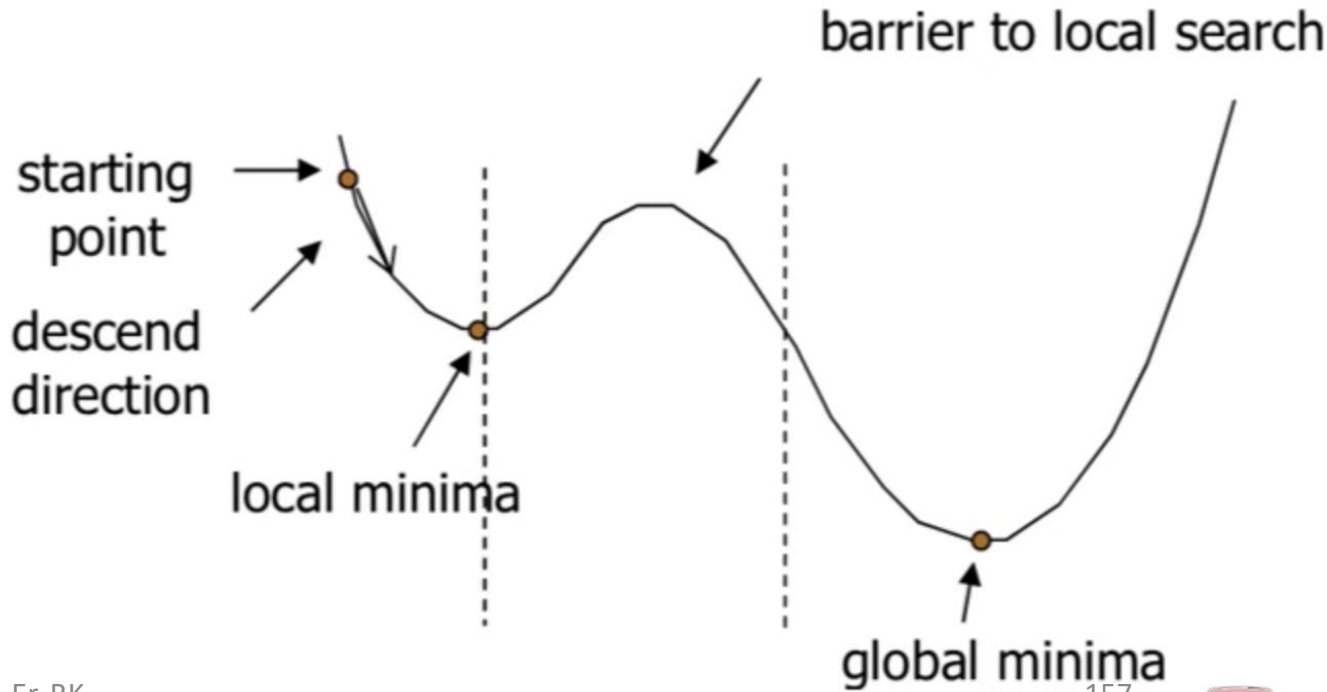
Simulated Annealing- Example- ping-pong ball

- Imagine the task of getting a ping-pong ball into the deepest crevice in a very bumpy surface.
- If we just let the ball roll, it will come to rest at a local minimum.
- The trick is to shake just **hard** enough, to bounce the ball out of local minima then the ball will reach the global minimum.



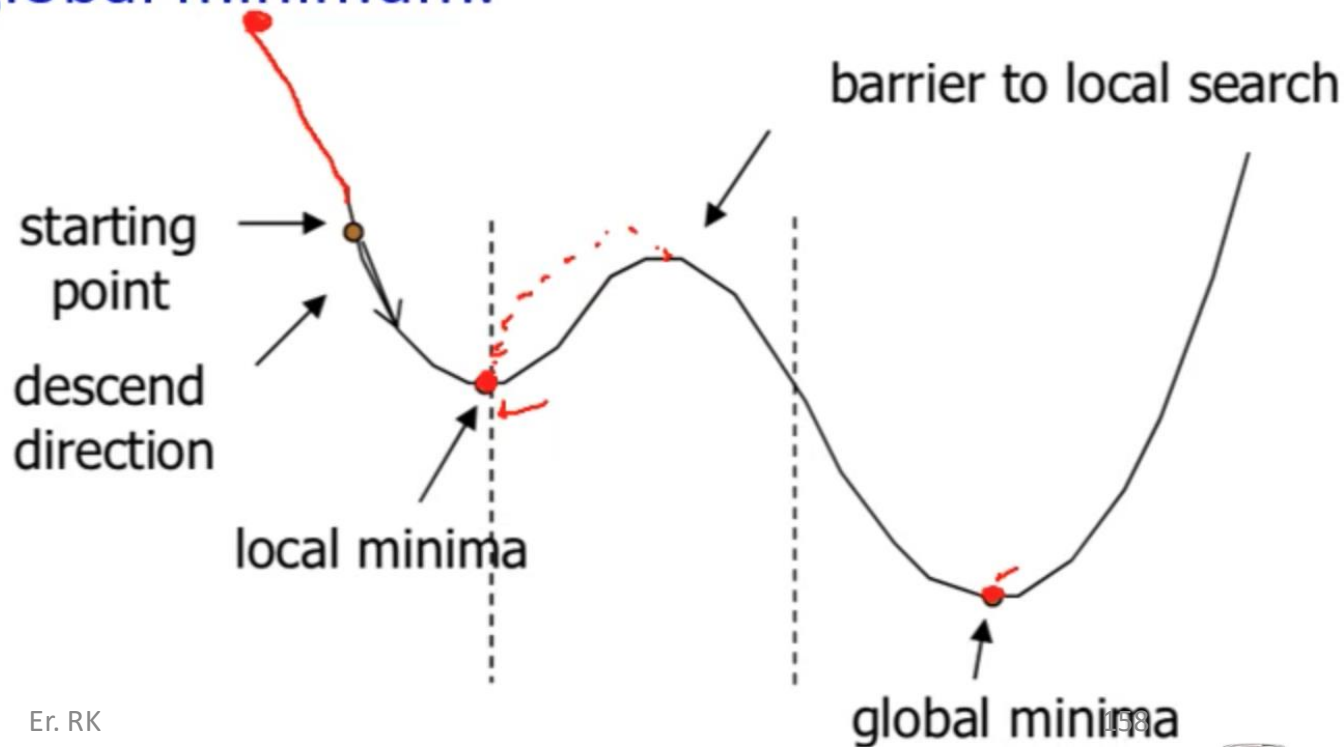
Simulated Annealing- Example- ping-pong ball

- Imagine the task of getting a ping-pong ball into the deepest crevice in a very bumpy surface.
- If we just let the ball roll, it will come to rest at a local minimum.
- The trick is to shake just **hard** enough, to bounce the ball out of local minima then the ball will reach the global minimum.



Simulated Annealing- Example- ping-pong ball

- Imagine the task of getting a ping-pong ball into the deepest crevice in a very bumpy surface.
- If we just let the ball roll, it will come to rest at a local minimum.
- The trick is to shake just **hard** enough, to bounce the ball out of local minima then the ball will reach the global minimum.



Simulated Annealing…

- Simulated annealing improves this strategy through the introduction of two tricks.
- This algorithm picks a *random* move instead of picking the best move.
- If the move improves the result then it accepts this random move, otherwise it accepts the move with some probability less than 1.

Simulated Annealing…

- Based on annealing, the algorithm uses the equation

$$e^{-\Delta D/T} > R\,(0,\,1),$$

- Where ΔD is Change of distance,
- T is synthetic temperature (control Variable) and
- R(0,1) is a random number in the interval [0,1].

Simulated Annealing Algorithm

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state

current ← *problem*.INITIAL

for $t = 1$ **to** ∞ **do**

$T \leftarrow schedule(t)$

if $T = 0$ **then return** *current*

next ← a randomly selected successor of *current*

$\Delta E \leftarrow$ VALUE(*current*) – VALUE(*next*)

if $\Delta E > 0$ **then** *current* ← *next*

else *current* ← *next* only with probability $e^{-\Delta E/T}$

Simulated annealing…

- Simulated annealing was used to solve VLSI layout problems
- It has been applied widely to factory scheduling and
- other large-scale optimization tasks.

CryptArithmetic Problem:-

↳ Type of Constraint Satisfaction Problem (CSP)

↳ Constraints → No two letters have same value. ≠
→ Sum of digits must be as shown in problem.
→ There should be only one carry forward.

Digits that can be assigned to a word/alphabet (0–9)

0 1 2 X

Range = 0–9

```
  T O
+ G O
-------
O U T
```

Left most digit = 1 ≠0

9+8 → 17 → Carry = 1

Letter	Digit
T	2
O	1
G	8 ≠
U	0

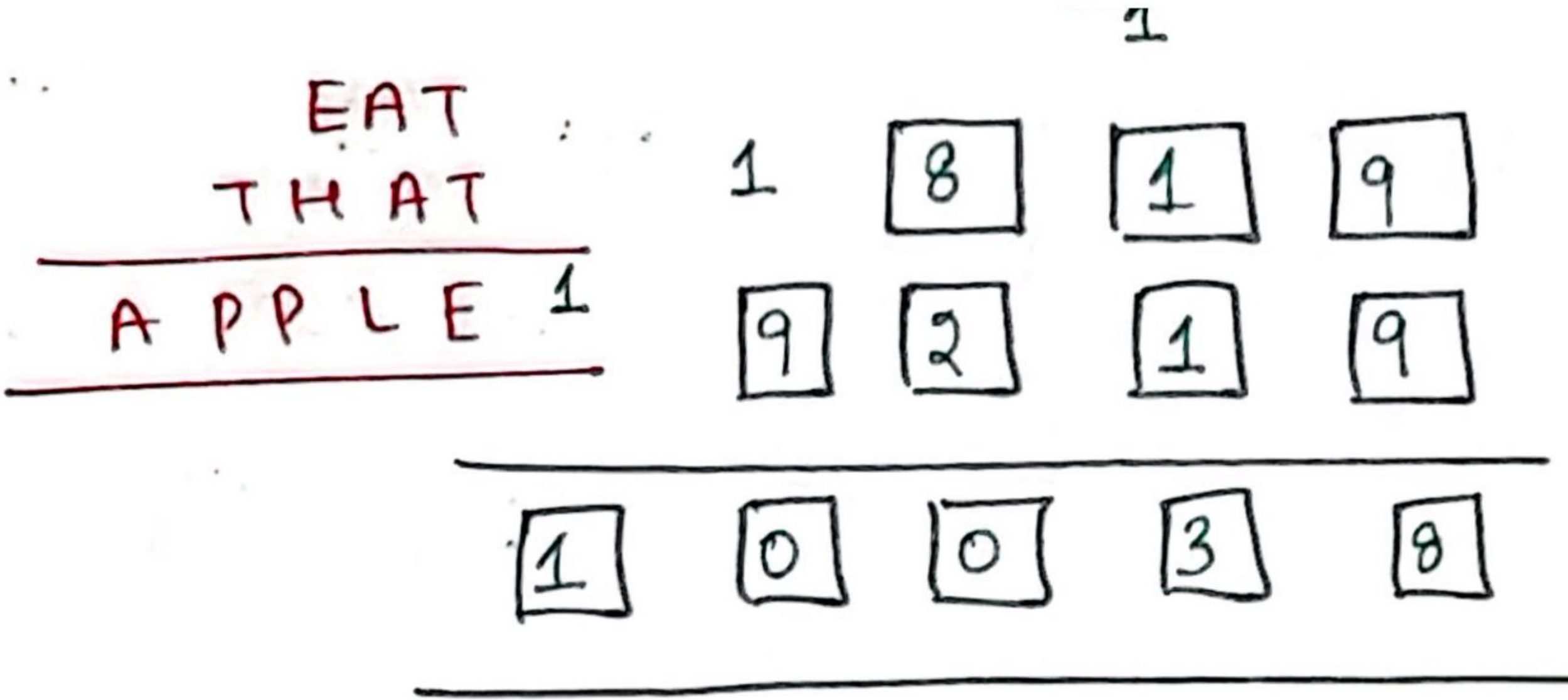
2+G = U+10

2+9 = 11 X

2+8 = 10

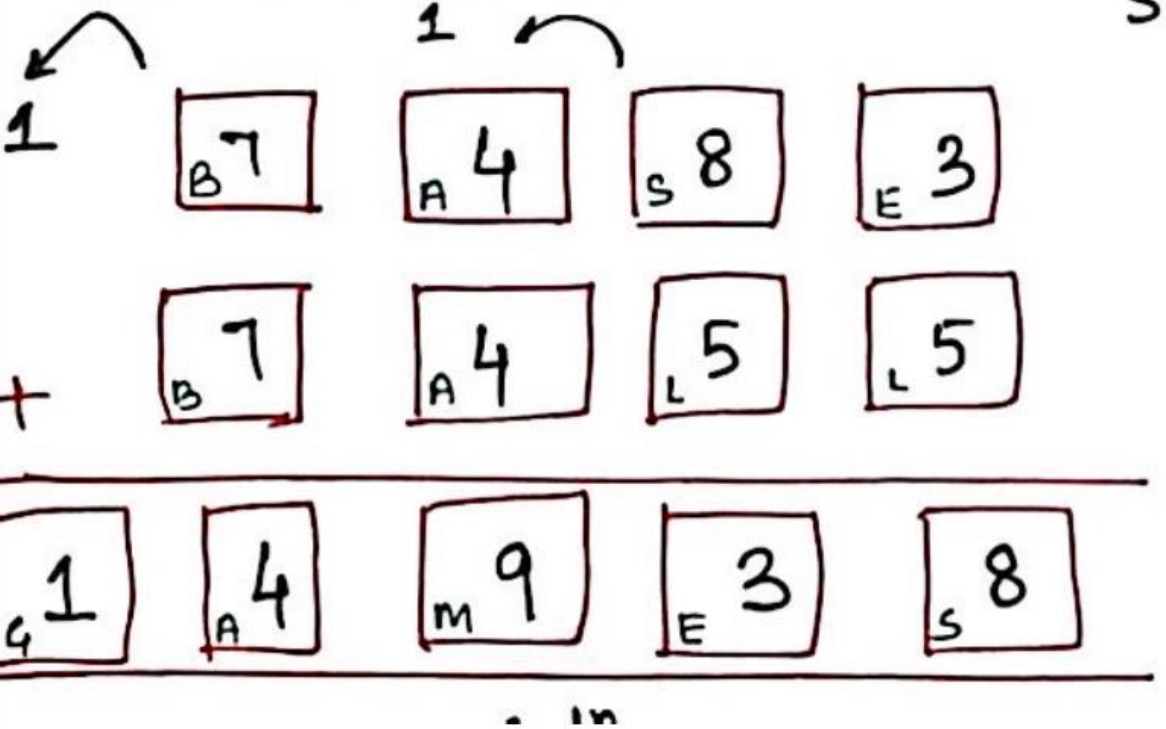
```
  2   1
+ 8   1
-------
  1   0   2
```

≠


EAT
THAT
APPLE
1
1
1 8 1 9
9 2 1 9
1 0 0 3 8

Cryptarithmetic Problem:-

```
  B A S E
+ B A L L
---------
G A M E S
```



$E+L=S$ —(a) [no Carry]

$E+L=S+10$ —(b) [Carry]

↳ $E=S-L+10$

$S+L=E$

$S+L=S-L+10$

or $2L=10$

$L=5$

$B+B=A+Carry$

B — 5 X, 6, 7, 8

$E+L=S$

or $S-E=L$

or $S-E=5$

(S,E) — (5,0) X, (6,1) X, (7,2) X, (8,3), (9,4)

Let (S,E) = (7,2) and B = 6.

↳ (S,E) = (8,3) 6+6 = 12 — A, C

SEND
+ MORE
MONEY

S+M, if m=1, s+m ≥ 10., S=9.

E+O=N
↳ zero. } if $C_2 = 0$ ✗
↳ $C_2 = 1$.

Let E=5

$E+O+C_2=N$

N=6.

N+R=E
6+R=5 → $C_1 = 1$

Let R=9 } S=9.
6+9=15 → ✗
↳ S

D+E=Y.

D+5=Y, 7+5=Y

Y=12

D≥5

D — 5 — E
6 — N
7
8 — R
9 — S

Carry

6+8+1=15

R.

C_4	C_3	C_2	C_1	
	S 9	E 5	N 6	D 7
+	M 1	O 0	R 8	E 5
1	O 0	N 6	E 5	Y 2

Soln =