MONASH University
Information Technology

www.monash.edu.au

MONASH University
Information Technology

FIT5047 – Intelligent Systems

Solving Problems by Searching
Chapters 3-5, 7

Problem Solving: Learning Objectives

- **Problem formulation**
- **Control strategies**
  - **Tentative**
    - > **Uninformed**:
      - – Backtracking [Chapter 7]
      - – Tree- and Graph search [Chapter 3]
    - > **Informed**: Best-first greedy search, A, A* [Chapter 3]
  - **Irrevocable**
    - > **Informed**: Hill climbing, Local beam search, Simulated annealing, Genetic algorithms [Chapter 4]
- **Adversarial search algorithms [Chapter 5]**
  - Optimal decisions
  - Minimax, α-β pruning

Assumptions about the Environment

- **Observable**
- **Known**
- **Single/multi agent**
- **Deterministic**
- **Sequential**
- **Static/dynamic**
- **Discrete**

Problem-solving Agents

Function Simple-Problem-Solving-Agent(*percept)*
returns *seq*
persistent: *state* – description of current world state
seq – action sequence
goal – a goal
problem – a problem formulation

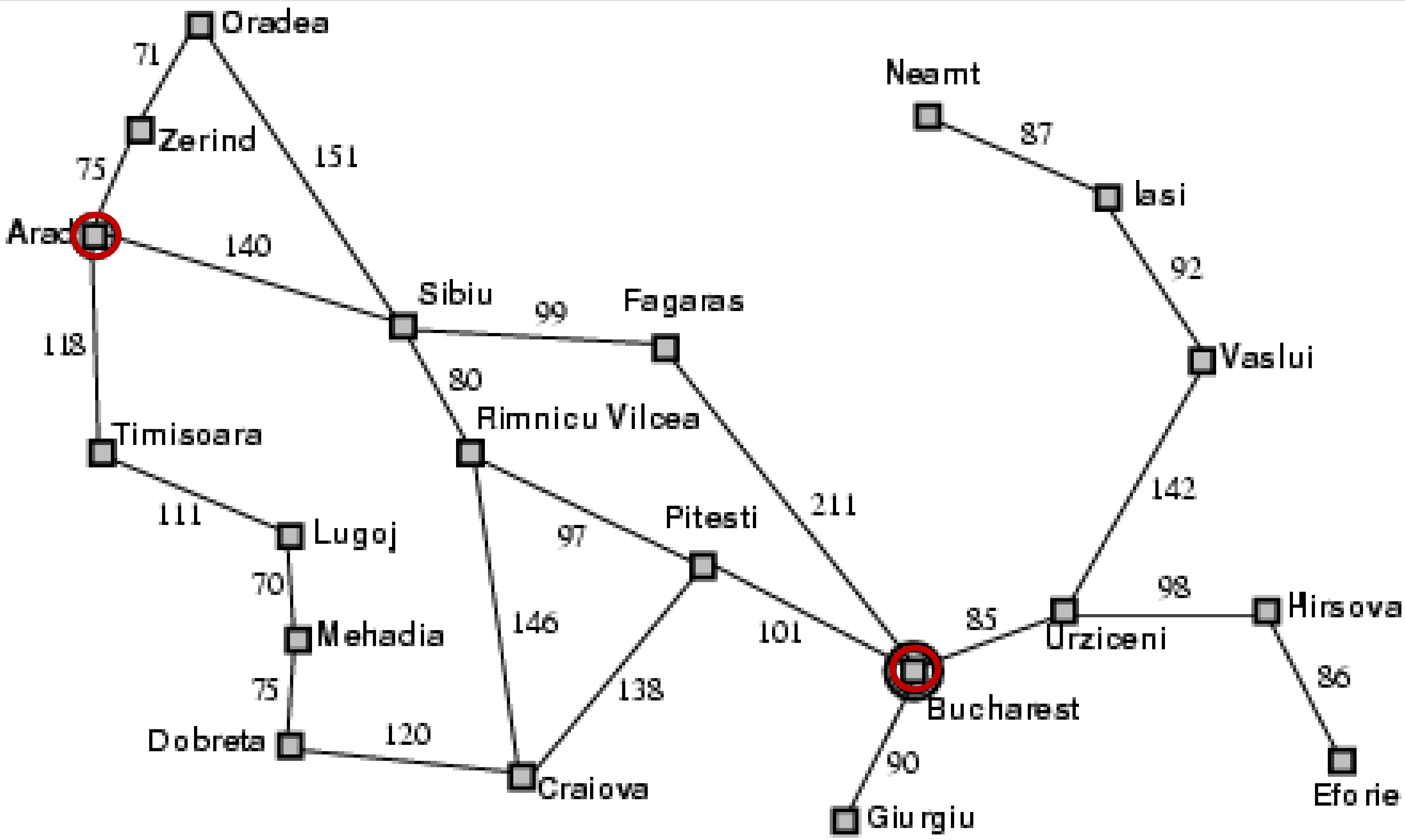
initially null

state ← UpdateState(*state*,*percept*)
goal ← FormulateGoal*(state)*
problem ← FormulateProblem(*state*,*goal*)
seq ← Search*(problem)*
return *seq*

Example: Romania

- ***On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest.***
- **Formulate goal:**
  - be in Bucharest
- **Formulate problem:**
  - states: various cities
  - actions: drive between cities
- **Find solution:**
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



MONASH University
Information Technology

FIT5047 – Intelligent Systems

Problem Formulation

Problem Formulation (I)

- **Problem formulation comprises decisions about:**
  - which properties of the world matter
  - which actions are possible
  - how to represent world states and actions

Abstracting away from unnecessary detail is a key
➔ It can drastically reduce the size of the state/search space

Problem Formulation (II)

- **Basic constituents**
  - States, Goals, Actions, Constraints
- **State space –** the set of all states reachable from the initial state by any sequence of actions
- **Path in the state space –** any sequence of actions leading from one state to another
- **Representing a problem**
  - Initial state
  - Operators (Actions) and transition model
  - Constraints
  - Goal test
  - Path cost function
- **A solution is a sequence of actions leading from the initial state to a goal state**

MONASH University
Information Technology

Problem Formulation: Example

1. **initial state, e.g., "at Arad"**
2. **actions**
   - e.g., {Go(Sibiu),Go(Timisoara), … }

   transition model
   - e.g., *Result(In(Arad),Go(Zerind))*→*In(Zerind)*
3. **constraints – nil**
4. **goal test can be**
   - explicit, e.g., *In(Bucharest)*
   - implicit, e.g., *Checkmate(x)*
5. **path cost (additive)**
   - e.g., sum of distances, number of actions executed
   - *c(s,a,s')* is the step cost of taking action *a* at state *s* to reach state *s'* (assumed to be ≥ 0)

Problem Formulation – 8 Puzzle (I)

Start

5	4	
6	1	8
7	3	2

End

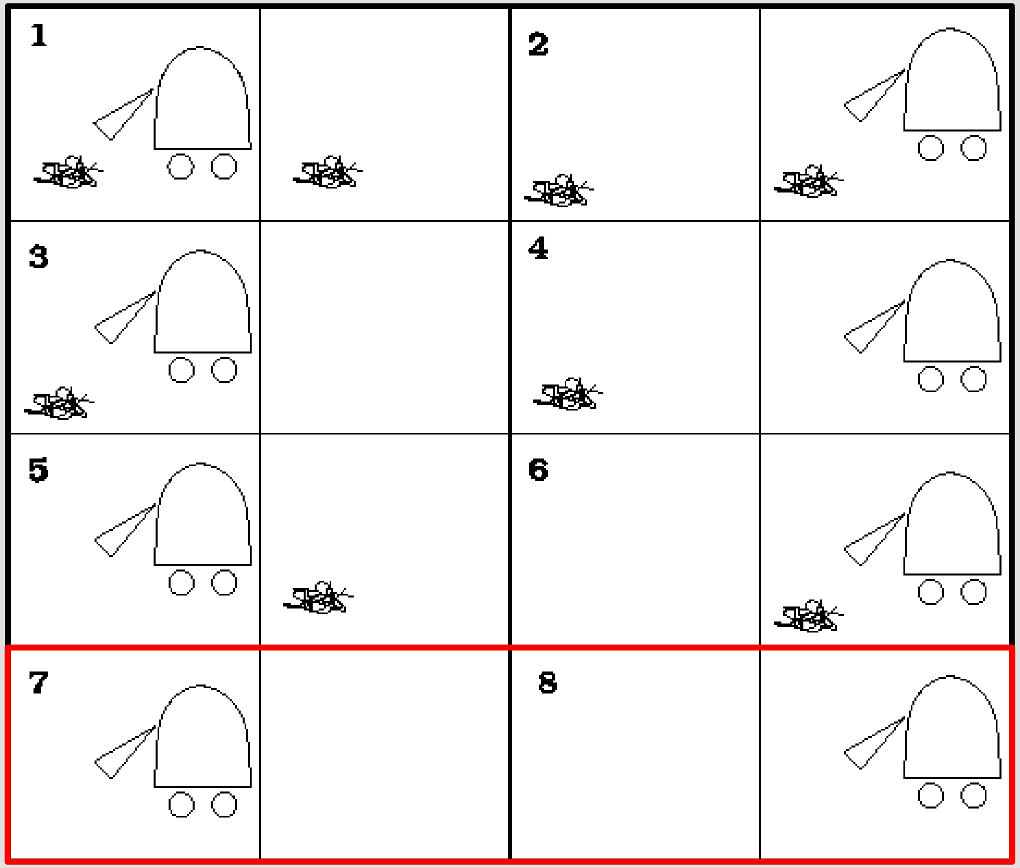
1	2	3
8		4
7	6	5

Problem Formulation – 8 Puzzle (II)

- **States**
  - Location of each of the 8 tiles in one of the 9 squares
- **Operators**
  - Possible moves of blank tile
- **Constraints**
  - A tile cannot move out of bounds
- **Goal test**
  - Have we reached the goal configuration?
- **Path cost**
  - If we want to minimize the number of steps, then cost of 1 per step

Problem Formulation – Vacuum World

- **States**
  - 8 states shown
- **Operators**
  - Left, right, suck
- **Constraints**
  - None
- **Goal test**
  - States 7 and 8
- **Path cost**
  - Each action costs 1



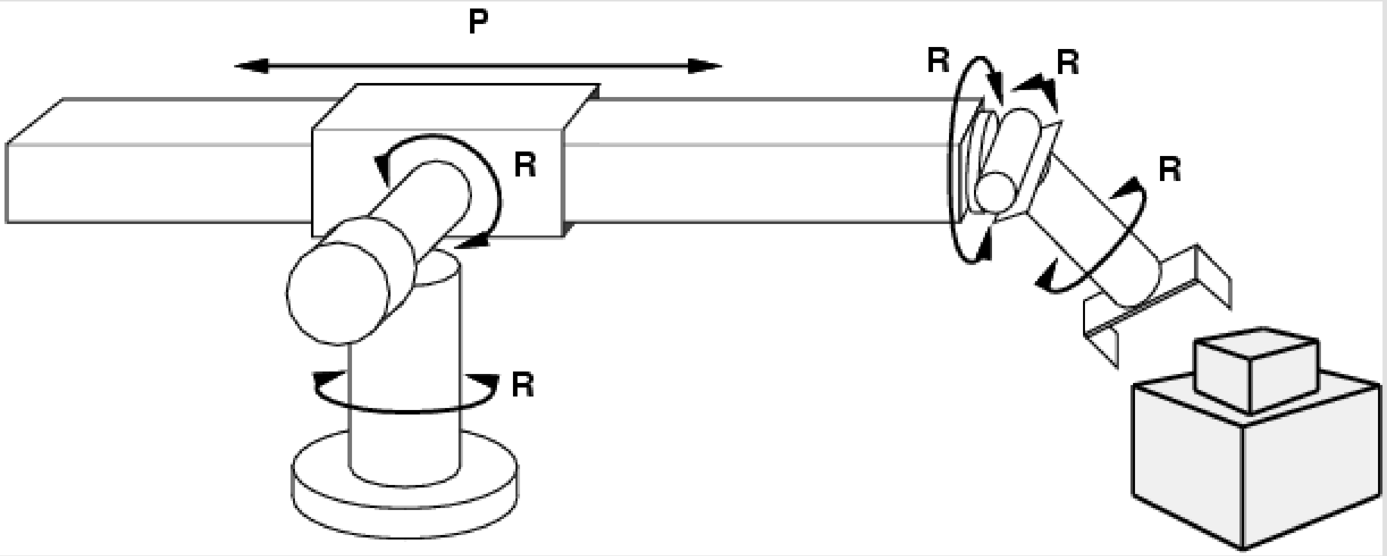
Problem Formulation: Missionaries and Cannibals (I)

- **Start state: 3 missionaries & 3 cannibals on one side of a river**
- **Goal state: 3 missionaries & 3 cannibals on the other side of the river**
- **Constraints:**
  - There is a boat that carries at most 2 people
  - The boat cannot travel empty
  - Cannibals should never outnumber missionaries

Problem Formulation: Missionaries and Cannibals (II)

- **States**
  - 2-digit code (m,c) represents the number of m and c on start bank; 1 digit code represents boat position
  - Initial state (3,3) + boat position
- **Operators**
  - 1m1c, 2m, 2c, 1m, 1c
- **Constraints**
  - $[(c \leq m) \land (3\text{-}c \leq 3\text{-}m)] \lor m=3 \lor m=0$
- **Goal test**
  - (0,0)
- **Path cost**
  - Cost function: Minimize number of crossings

Problem Formulation: Robotic Assembly



- states?: real-valued coordinates of robot joint angles; parts of the object to be assembled
- actions?: continuous motions of robot joints
- constraints?: arm cannot fully rotate up and down
- goal test?: complete assembly
- path cost?: time to execute

MONASH University
Information Technology

Selecting a State Space

- **Real world is complex**

  → state space must be abstracted for problem solving

- **(Abstract) state = set of real states**
- **(Abstract) action = complex combination of real actions**
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc
- **For *guaranteed realizability*, any real state ("in Arad") must get to some real state ("in Zerind")**
- **(Abstract) solution = a solution that can be expanded into a set of real paths in the real world**
- **Each abstract action should be "easier" to perform than solving the original problem**

MONASH University
Information Technology

FIT5047 – Intelligent Systems

Control Strategies

Classification of Control Strategies

- **Tentativeness**
  - Irrevocable – no reconsideration
  - Tentative – with reconsideration
- **Informedness**
  - Uninformed – decide based ***only*** on problem definition
  - Informed – use guidance on where to look for solutions

	Irrevocable	Tentative
Uninformed	--	Backtrack, Tree- and Graph-Search (BFS, DFS, DLS, IDS, UCS)
Informed	Hill climbing, Greedy search, Local beam search, Simulated annealing, Genetic algorithms	Best first (Greedy), A, A*

FIT5047 – Intelligent Systems

Tentative Search Algorithms: Backtrack, Tree- and Graph-search

Tentative Control Strategies

- **Backtracking – at any point in time, we keep one path only**
  – If we fail, we go back to the last decision point and **erase the failed path**
  – Backtracking occurs when
    > we reach a DEADEND state OR
    > there are no more applicable rules OR
    > we generate a previously encountered state description OR
    > an arbitrary number of rules has been applied without reaching the goal
- **Graphsearch – we keep track of several paths simultaneously**
  – Done using a structure called a ***search tree/graph***

Basic Backtrack Algorithm

Procedure Backtrack (State)

1. **If Goal(State) Then return SUCCEED**
2. **If Deadend(State) Then return FAIL**
3. **Operators ← ApplicableOps(State)**
4. **Loop**
    1. **If** null(Operators) **Then** return FAIL
    2. Op ← Pop(Operators)
    3. State’ ← Op(State)
    4. Path ← Backtrack(State’)
    5. **If** Path=FAIL **Then** go Loop
    6. Return {Op, Path}

    End

Backtrack Algorithm

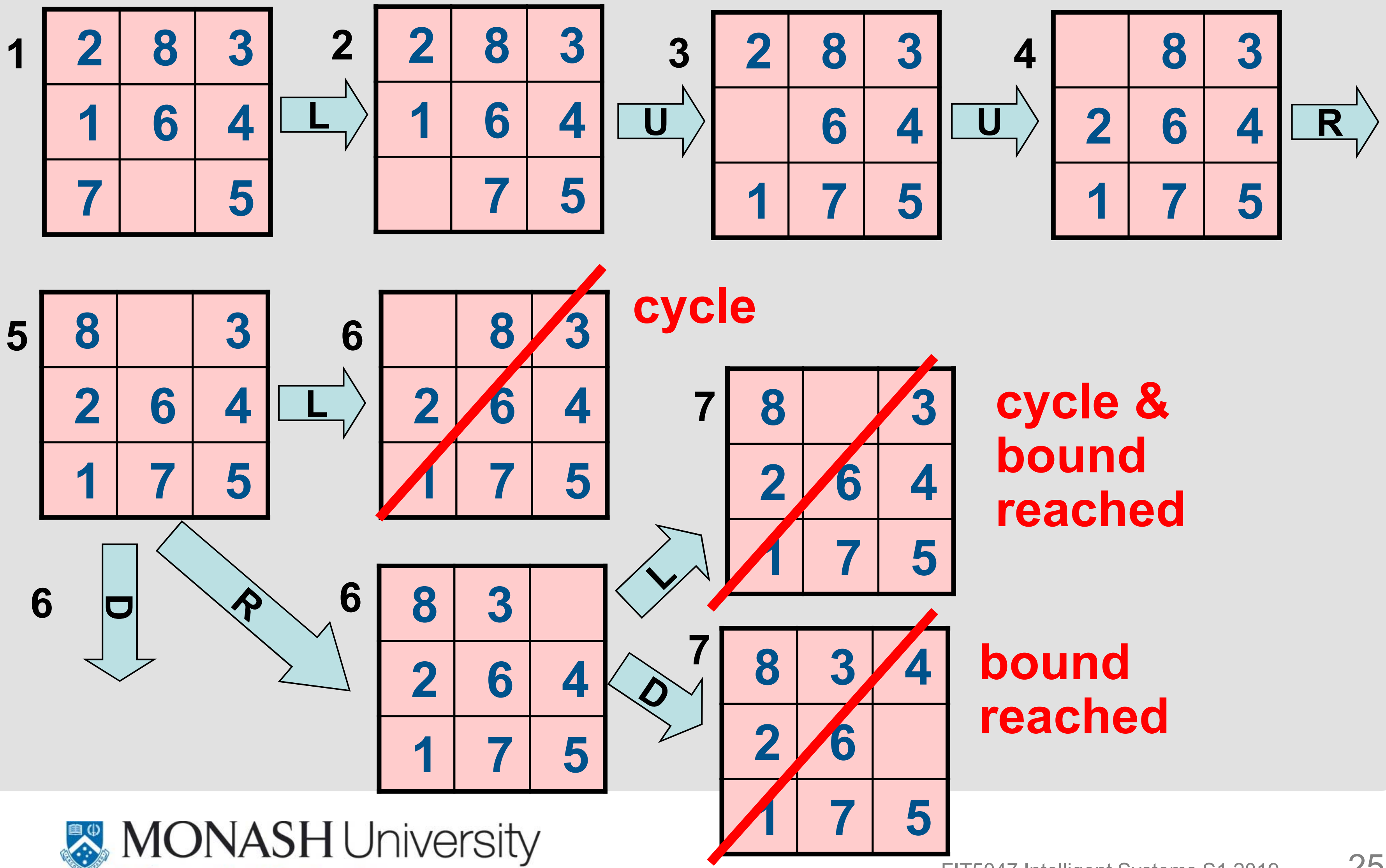
Procedure Backtrack1(StateList)

1. **State ← First(StateList)**
2. **If State ϵ RestOf(StateList) Then return FAIL**
3. **If Goal(State) Then return SUCCEED**
4. **If Deadend(State) Then return FAIL**
5. **If Length(StateList) > Bound Then return FAIL**
6. **Operators ← ApplicableOps(State)**
7. **Loop**
    1. **If** null(Ops) **Then** return FAIL
    2. Op ← Pop(Ops)
    3. State' ← Op(State)
    4. StateList' ← {State', StateList}
    5. Path ← Backtrack1(StateList')
    6. **If** Path=FAIL **Then** go Loop
    7. Return {Op, Path}

    End

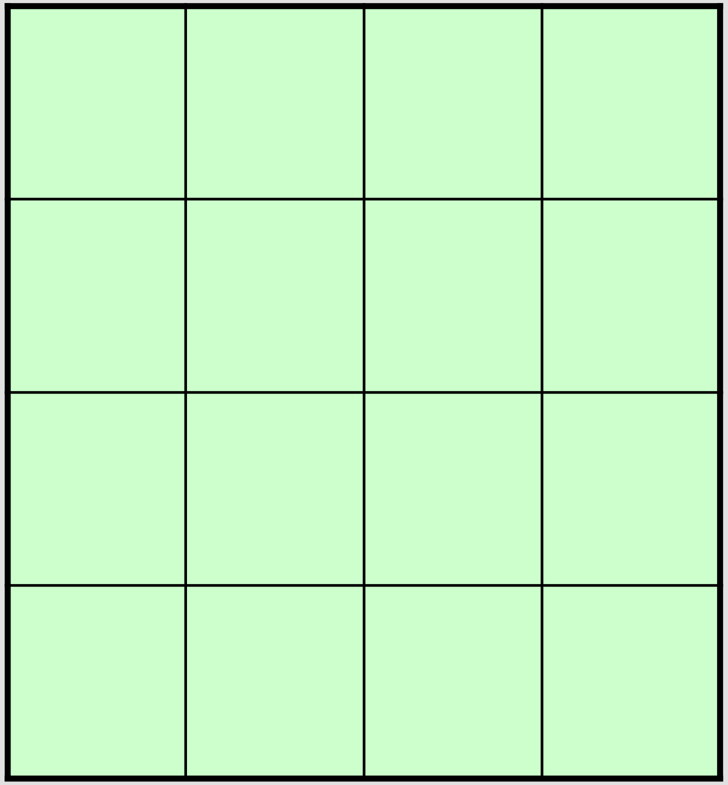
MONASH University
Information Technology

Backtrack Example – Bound = 6



Backtracking Example – 4 Queens Problem

- **Start state:**
  - empty chess board
- **Goal state:**
  - 4 queens placed on chess board
- **Constraints:**
  - queens don't attack each other
- **Operators:**
  - place queen on tile (x,y)
- **Path cost: NA**



Graphsearch – Definitions

- **Graphsearch is a means of finding a path in a graph from a node representing the initial state to a node that satisfies the goal condition**
- **Definitions**
  - **Graph** – set of nodes
  - **Arcs** – connect between certain pairs of nodes
  - **Directed graph** – formed by arcs directed from one node to another
  - n_i is a ***child of*** n_k if $n_k \rightarrow n_i$
  - n_i is ***accessible from*** n_k if there is a path from n_k to n_i
  - **Expanding a node** – finding all its children
  - **Search Problem** – find a path between node *s* and any member of the ***goal set*** $\{t_i\}$ that represents states satisfying the goal condition

Search Tree

- **Tree – each node has at most one parent**
- **Root of search tree is the initial state**
- **Leaves are states without successors (the "fringe" or "frontier")**
- **At each step, choose one leaf node to *expand***

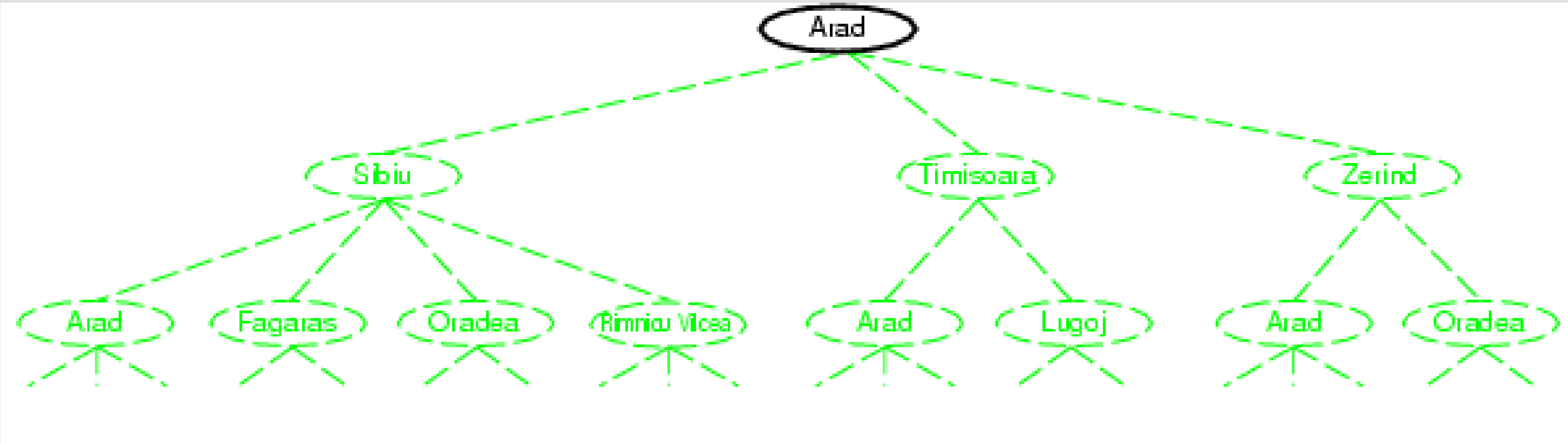
Basic Tree Search Algorithm

function TREE-SEARCH(*problem*) **returns** a solution or failure

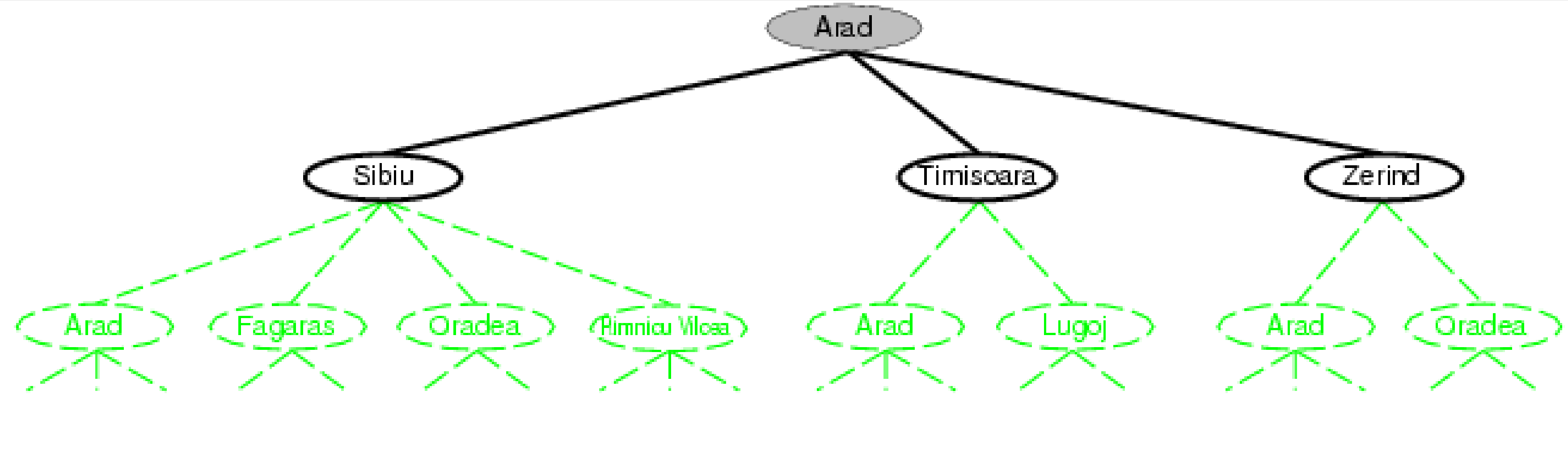
- Initialize the frontier using the initial state of *problem*
- **Loop**
  1. **if** the frontier is empty **then return** failure
  2. ***choose*** a leaf node and remove it from the frontier
  3. **if** the node contains a goal state **then return** the corresponding solution
  4. ***expand*** the chosen node, ***adding*** the resulting nodes to the frontier

  end

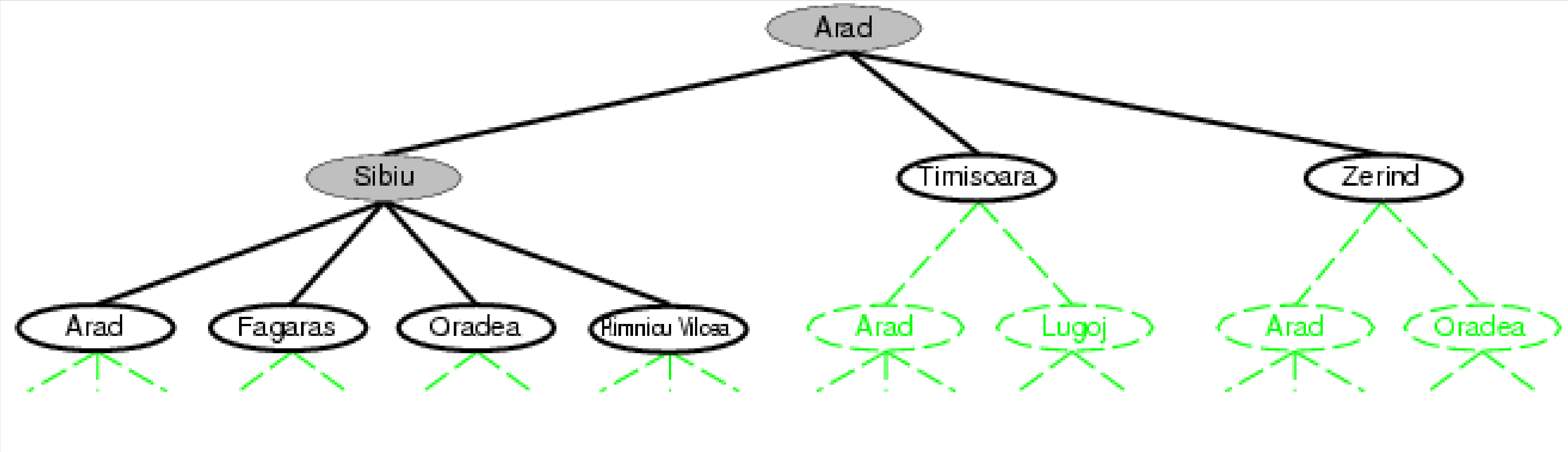
Example: Tree Search (I)



Example: Tree Search (II)

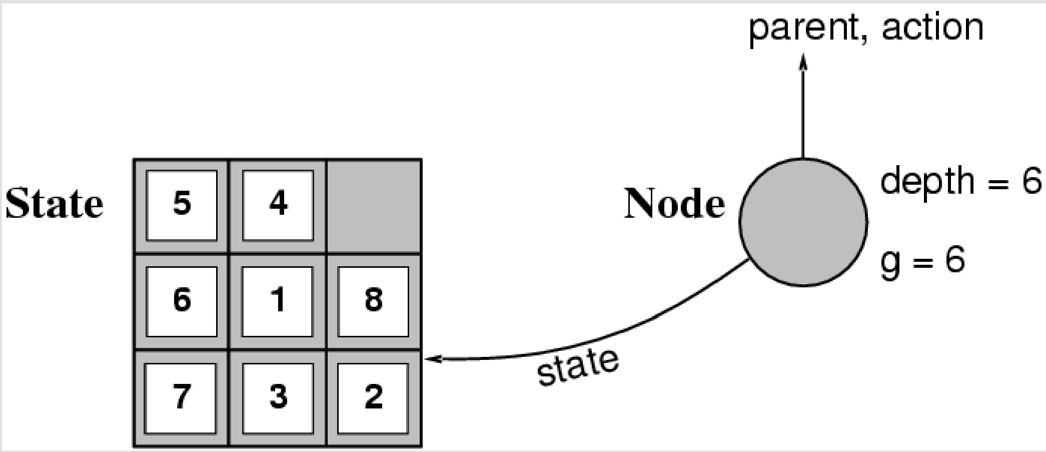


Example: Tree Search (III)



Implementation: States vs. Nodes

- **state – a (representation of a) physical configuration**
- **node – a data structure that is part of a search tree**
  - includes *state, parent node, action, children, path cost g(x), depth*



- **The *Expand* function**
  - creates new nodes, fills in the various fields
  - uses ***SuccessorFn(Operators)*** to create the corresponding states

Searching graphs: Multiple Paths to a Node

- **Often search better is represented via graphs:**
  - There may be multiple paths to the same state
  - Improvement (due to search graph) depends on how costly it is to determine a node has already been visited

Graph Search Algorithm

function GRAPH-SEARCH(*problem*) **returns** a solution or failure

- Initialize the frontier using the initial state of *problem*
- Initialize the *explored set* (***closed***) to empty
- **Loop**
  1. **if** the frontier is empty **then return** failure
  2. ***choose*** a leaf node and remove it from the frontier
  3. **if** the node contains a goal state **then return** the corresponding solution
  4. add the node to the *explored set*
  5. ***expand*** the chosen node, ***merging*** the resulting nodes with the frontier or the *explored set*

  end

Basic Search Algorithm: Key Issues

- **Return a path or a node?**
- **Unboundedness:**
  - Tree search: because of loops
  - Graph/tree search: because the state space is infinite
- **Tree search: Repeated states**
  - Failure to detect repeated states can increase the complexity of a problem
- **How are the nodes ordered? ➔ Search strategy**
  - Is the graph weighted or unweighted?
  - How much is known about the “quality” of intermediate states?
  - Is the aim to find a ***minimal cost path*** or ***any path asap***?

Dealing with Repeated States

- **3 ways to deal with repeated states (ordered by cost and effectiveness):**
  - Do not return to the state you just came from
    → don't generate successors with same state as a node's parent
  - Do not create paths with cycles in them
    → don't generate successors with same state as any ancestor
  - Do not generate any state that was ever generated before
    → Use hashset to check if state has been visited

Implementation of the Graphsearch Algorithm

1. **Create a search graph *G* consisting only of the start node *s***
2. **OPEN ← *s***
3. **CLOSED ← Ø**
4. **Loop**
   1. **If** OPEN = Ø **Then** exit with failure
   2. *n* ← first node in OPEN
      Remove *n* from OPEN, put it in CLOSED
   3. **If** *n* = goal-node **Then** exit successfully with the solution obtained by tracing a path along the pointers from *n* to *s* in *G*
   4. **Expand node *n***, generating a set *M* of its children *that are not ancestors of n*. Put these members of *M* as children of *n* in *G*.
   5. Establish a pointer to *n* from those members of *M* *that were not already in G*. Add these members of *M* to OPEN. For each member of *M* already in *G*, decide whether or not to redirect its pointer to *n*.
   6. Reorder OPEN (according to an arbitrary scheme or merit)

End

MONASH University
Information Technology

FIT5047 – Intelligent Systems

Tree and Graph Search Strategies

Search Strategies

- **A search strategy is defined by picking the order of node expansion**
- **Strategies are evaluated along several dimensions:**
  - **completeness**: does it always find a solution if one exists?
  - **time complexity**: maximum number of nodes generated
  - **space complexity**: maximum number of nodes in memory
  - **optimality**: does it always find a least-cost solution?
- **Time and space complexity are measured in terms of**
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of any path in the search space (may be ∞)

O Notation

- ***n*** **measures the size of the input**
- ***f*(*n*) is a function characterizing the worst-case complexity of an algorithm**
- ***O*(*f*(*n*)) is the set of all functions (eventually, asymptotically) bounded from above by some positive multiple *k* of *f*(*n*)**

Example:

Let *n* be the number of items to be sorted, then

- **Bubble sort has worst case $k_1 n^2$; i.e., $O(n^2)$**
- **Heap sort has worst case $k_2 n \log n$; i.e., $O(n \log n)$**

MONASH University
Information Technology

FIT5047 – Intelligent Systems

Uninformed Search Strategies

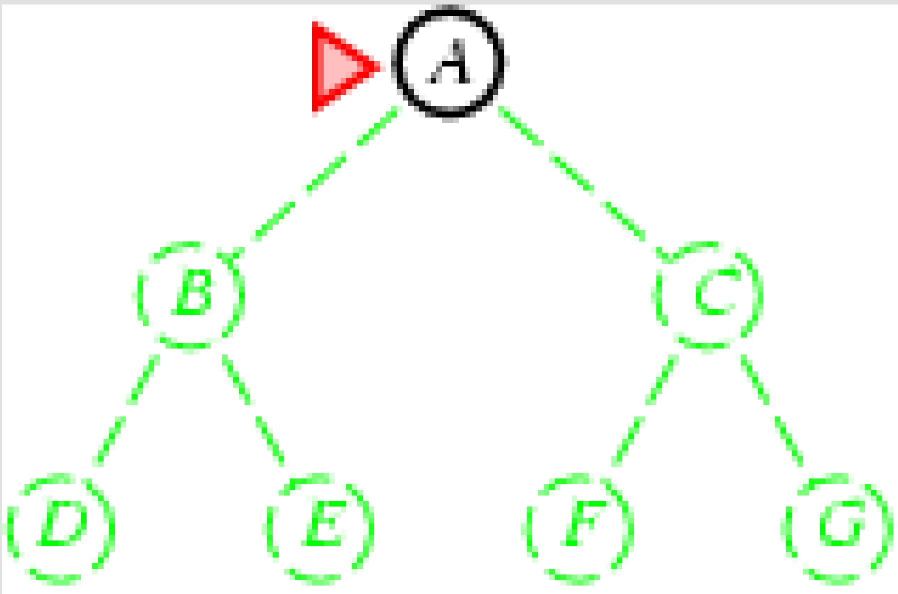
Uninformed Search Strategies

Uninformed search strategies use only the information available in the problem definition

- **Breadth-first search (BFS)**
- **Uniform-cost search (UCS)**
- **Depth-first search (DFS)**
- **Depth-limited search (DLS)**
- **Iterative deepening search (IDS)**

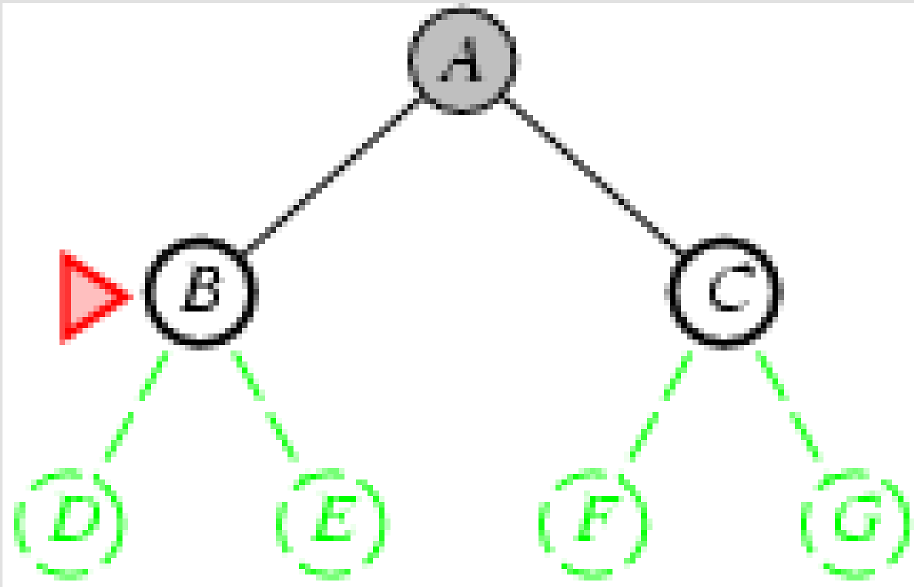
Breadth-first Search (I)

- **Expand shallowest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: FIFO – put successors at end of queue



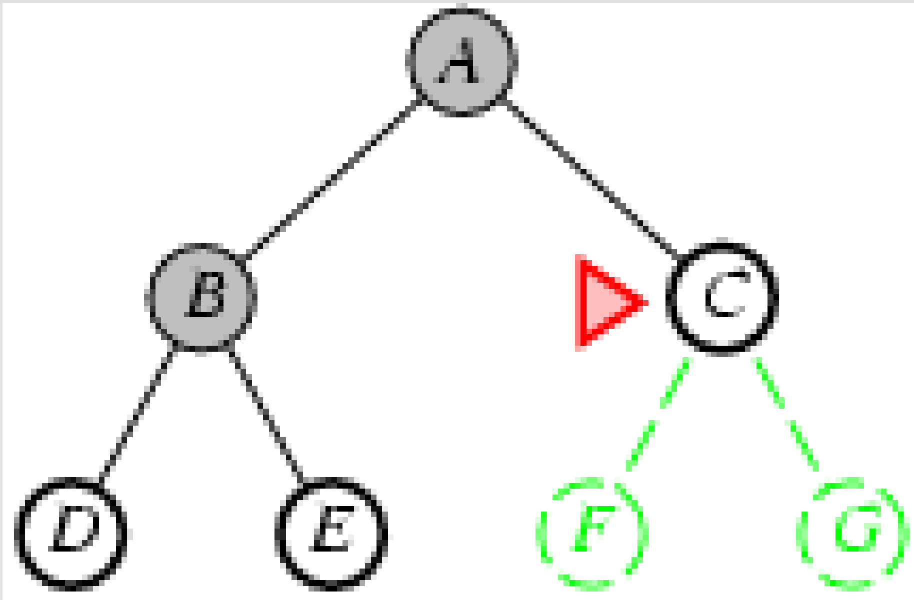
Breadth-first Search (II)

- **Expand shallowest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: FIFO – put successors at end of queue



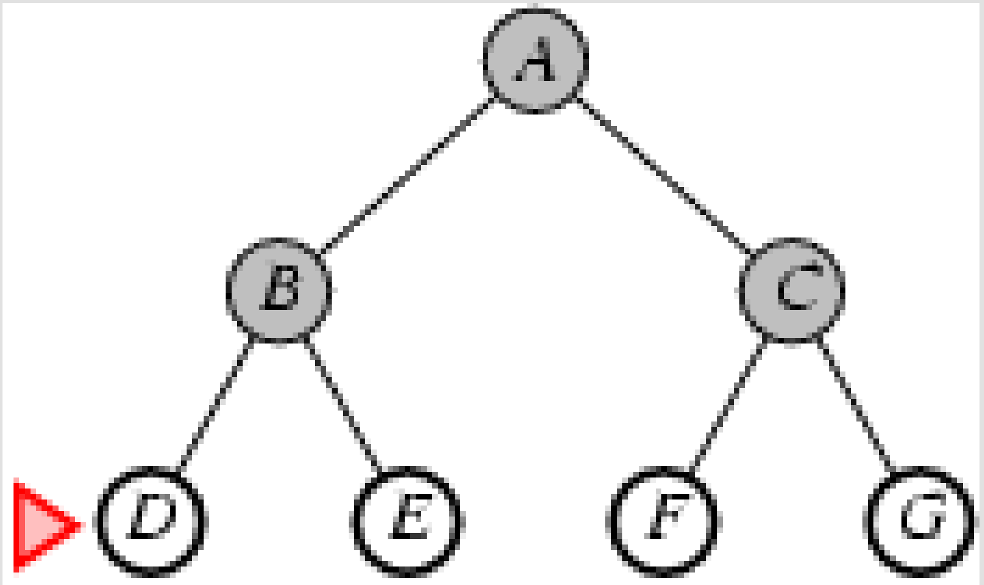
Breadth-first Search (III)

- **Expand shallowest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: FIFO – put successors at end of queue

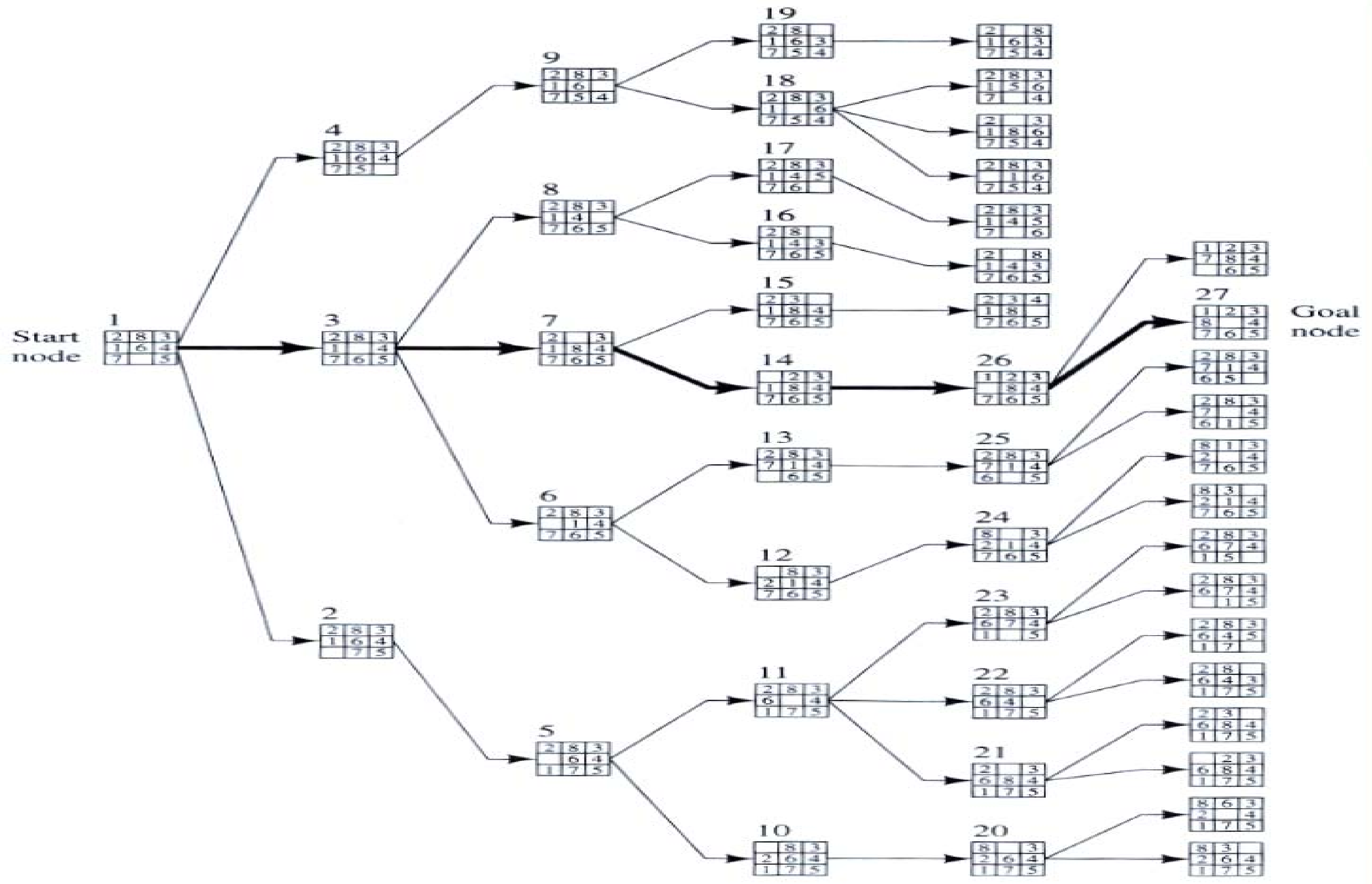


Breadth-first Search (IV)

- **Expand shallowest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: FIFO – put successors at end of queue



BFS – Example



Start node
Goal node
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Properties of Breadth-First Search

- **Complete?** **Yes (if b is finite)**
- **Time?** $b + b^2 + b^3 + \ldots + b^d = b\frac{b^d - 1}{b-1} \rightarrow O(b^d)$
- **Space?** $O(b^d)$ **(keeps every node in memory)**
- **Optimal?** **Yes (if all actions have the same cost)**

Space is the bigger problem

Only tree/graphsearch algorithm that can stop when the goal node is reached

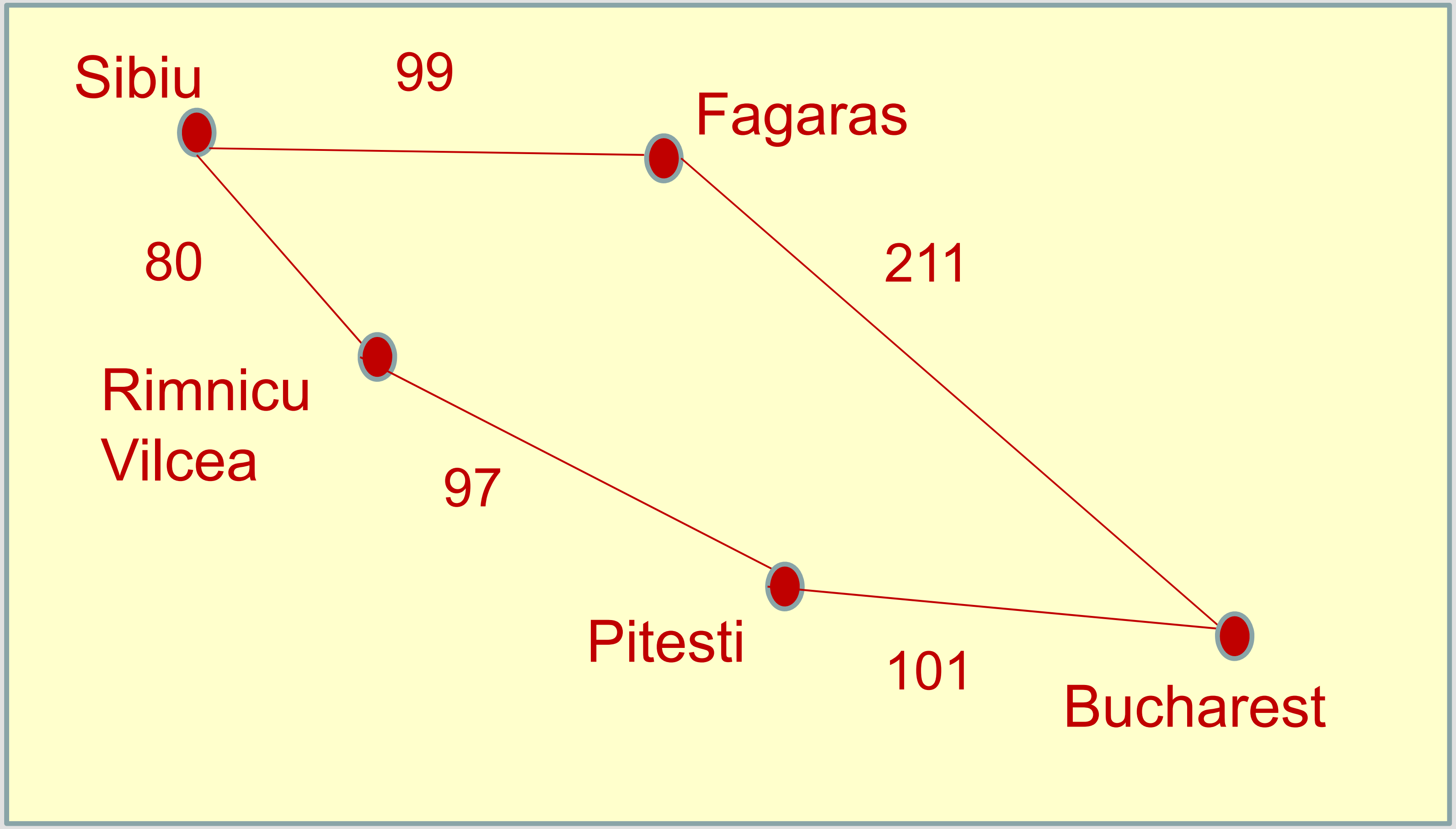
Uniform-Cost Search Algorithm

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

- Initialize the frontier using the initial state of *problem*
- **Loop**
  1. **if** the frontier is empty **then return** failure
  2. ***choose*** the lowest-cost node in the frontier and remove it from the frontier
  3. **if** the node contains a goal state **then return** the corresponding solution
  4. ***expand*** the chosen node
     a. **if** the resulting nodes are not in the frontier **then add** them to the frontier
     b. **else if** the resulting nodes are in the frontier ***with higher path cost*** **then replace** them with the new nodes

end

Uniform-cost Search: Example



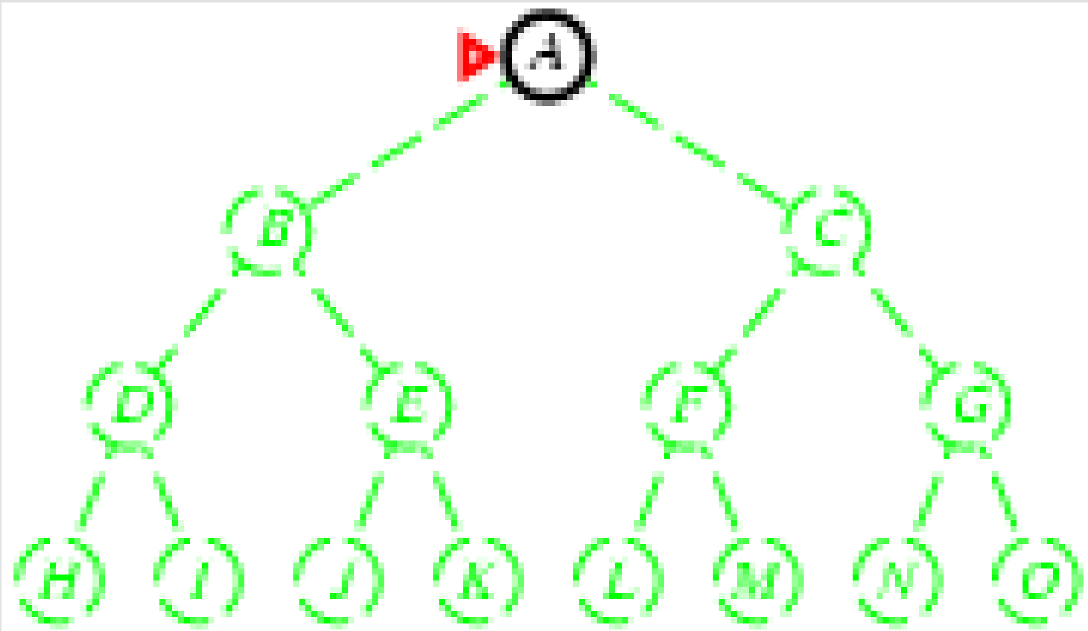
Properties of Uniform-cost Search

- **Almost equivalent to BFS if step costs all equal**
- **Complete? Yes, if step cost ≥ ε**
- **Time?** $O(b^{1+floor(C^*/\varepsilon)})$
  - where C^* is the cost of the optimal solution
- **Space?** $O(b^{1+floor(C^*/\varepsilon)})$
- **Optimal? Yes – nodes expanded in increasing order of *g(n)* = cost of path to node *n***

When all step costs are the same, UCS does more work than BFS. Why?

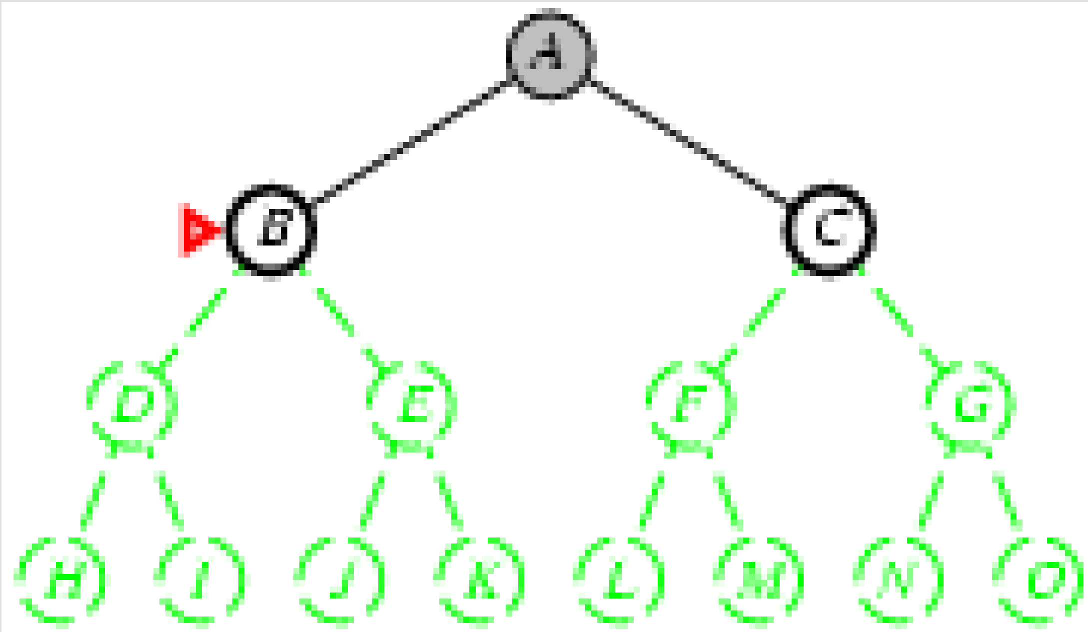
Depth-first Search (I)

- **Expand the deepest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: LIFO – insert successors in front of queue



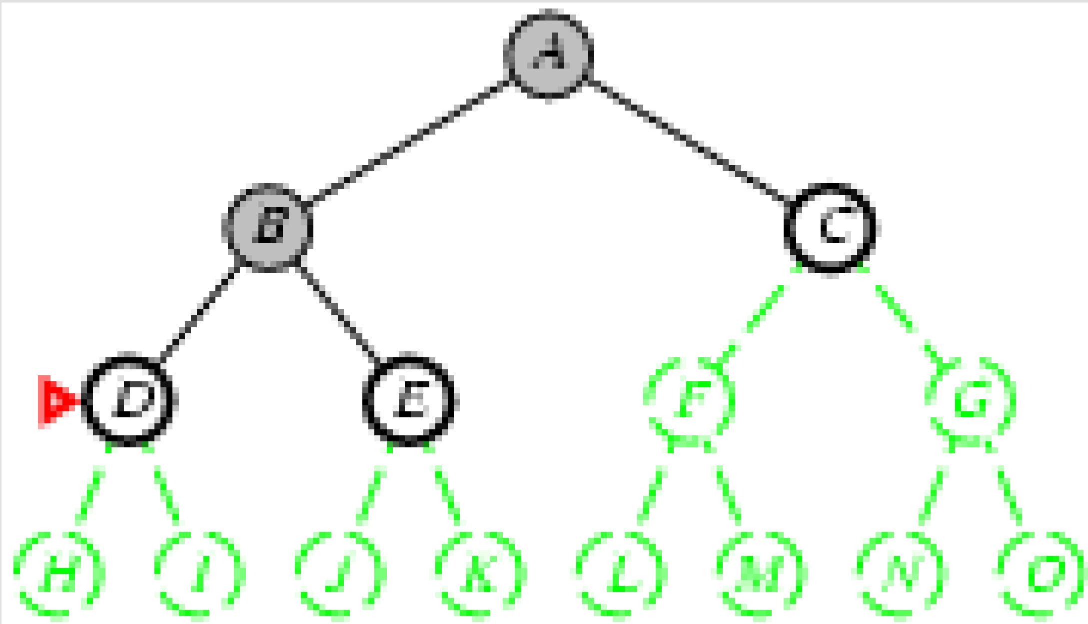
Depth-first Search (II)

- **Expand the deepest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: LIFO – insert successors in front of queue



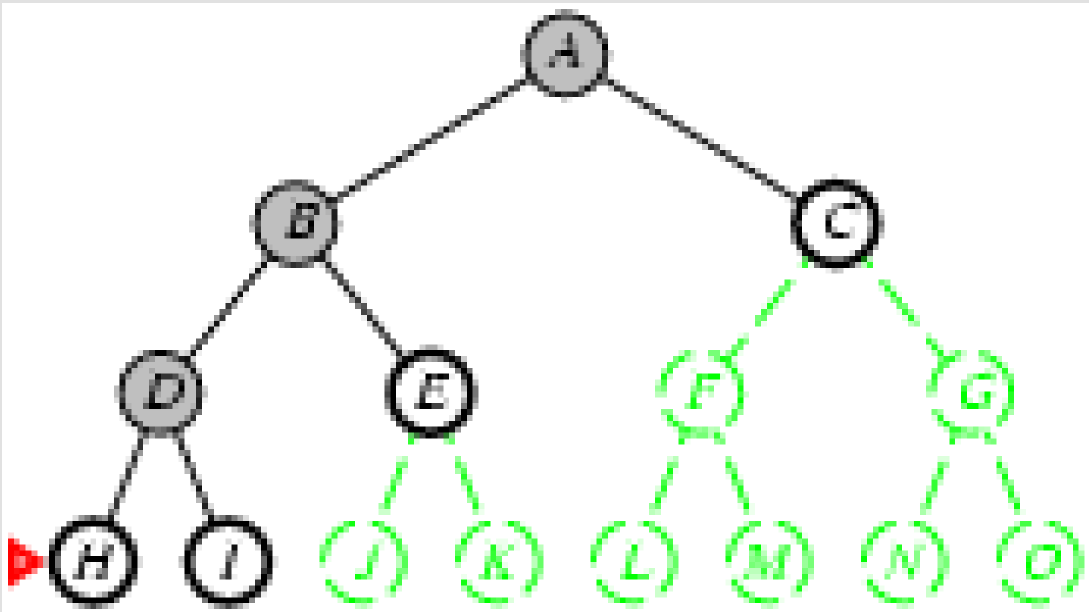
Depth-first Search (III)

- **Expand the deepest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: LIFO – insert successors in front of queue



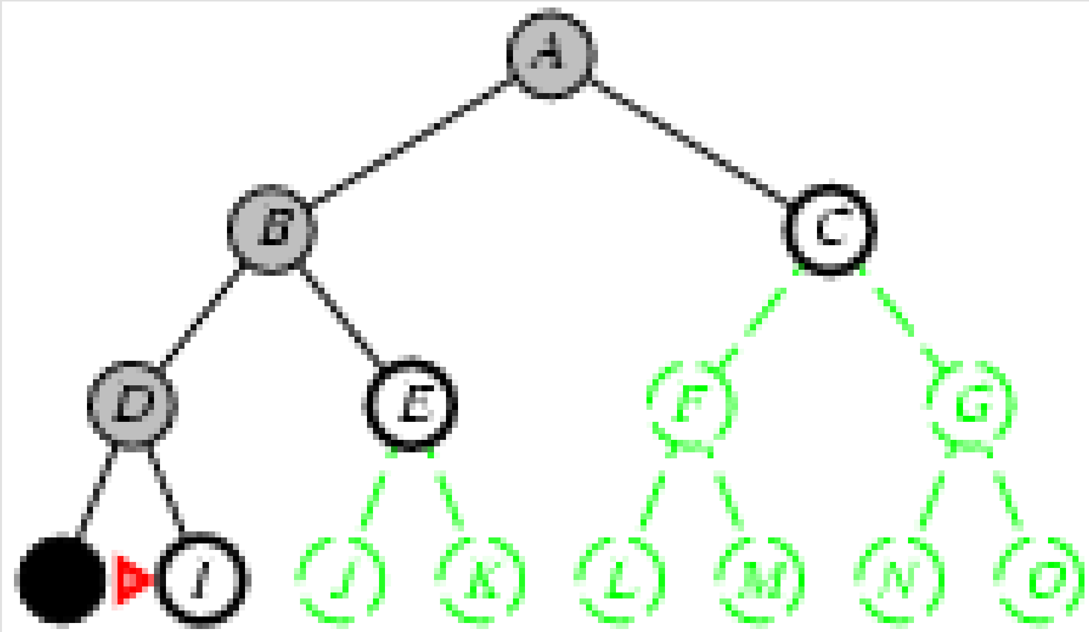
Depth-first Search (IV)

- **Expand the deepest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: LIFO – insert successors in front of queue



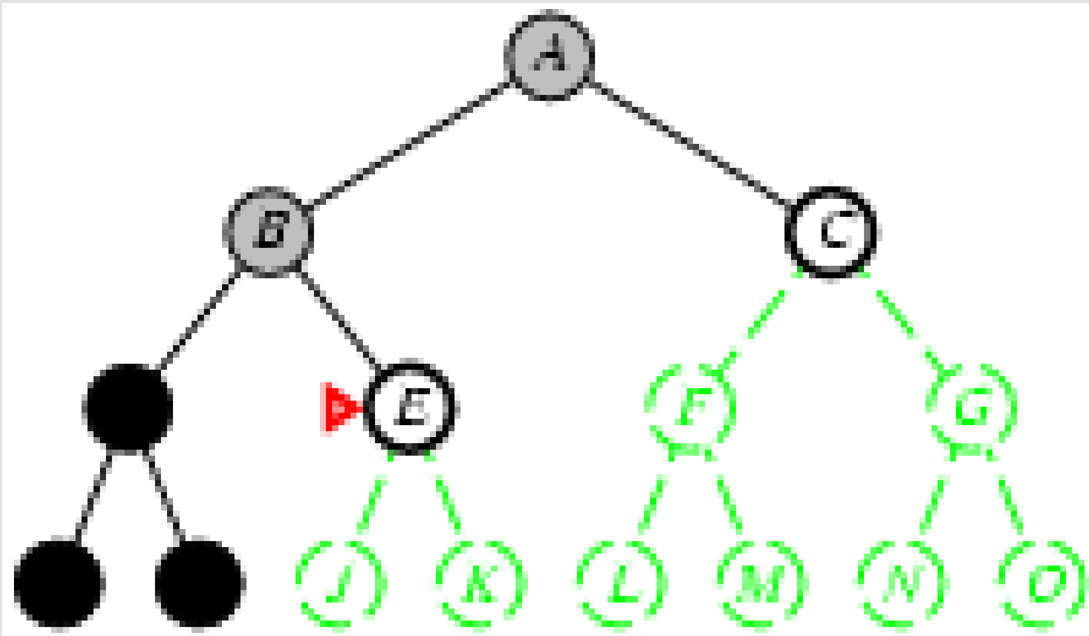
Depth-first Search (V)

- **Expand the deepest unexpanded node**
- **Implementation: managing the frontier**
  – QUEUEING-FN: LIFO – insert successors in front of queue



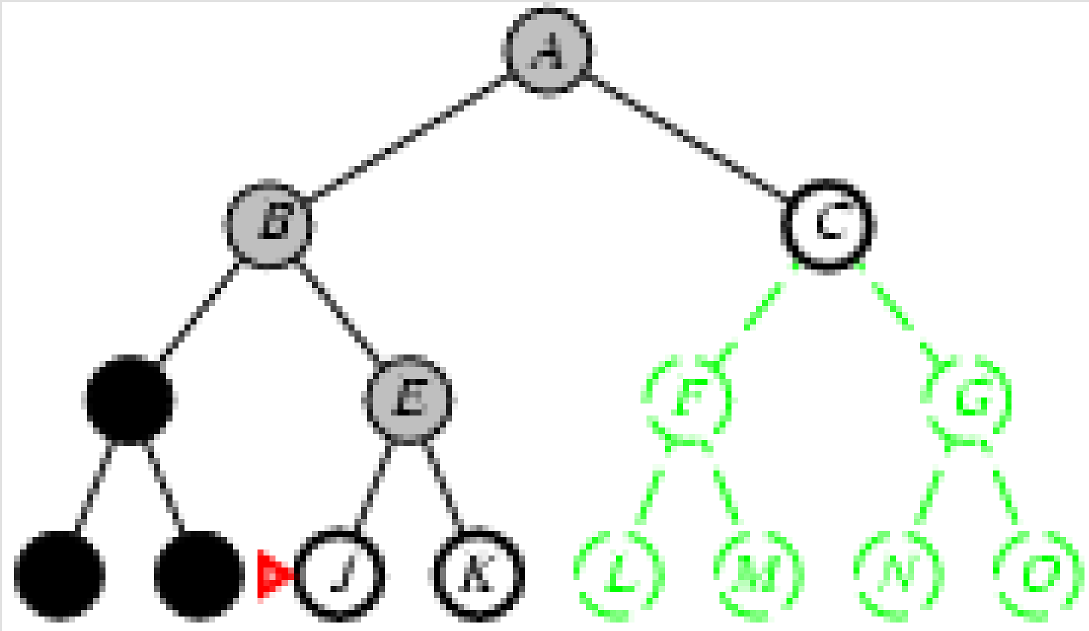
Depth-first Search (VI)

- **Expand the deepest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: LIFO – insert successors in front of queue



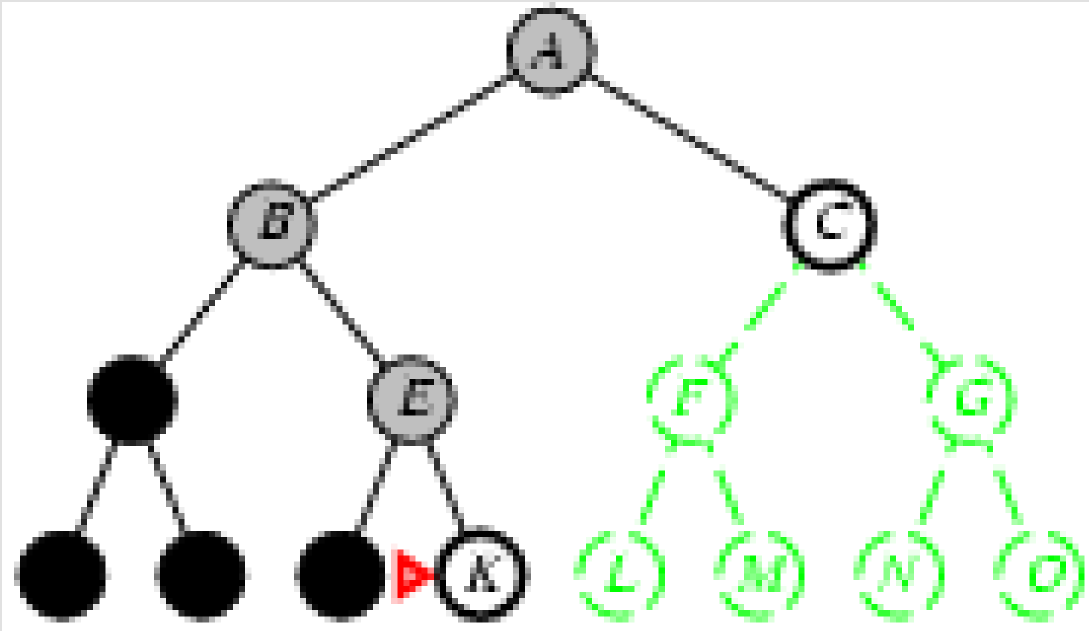
Depth-first Search (VII)

- **Expand the deepest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: LIFO – insert successors in front of queue



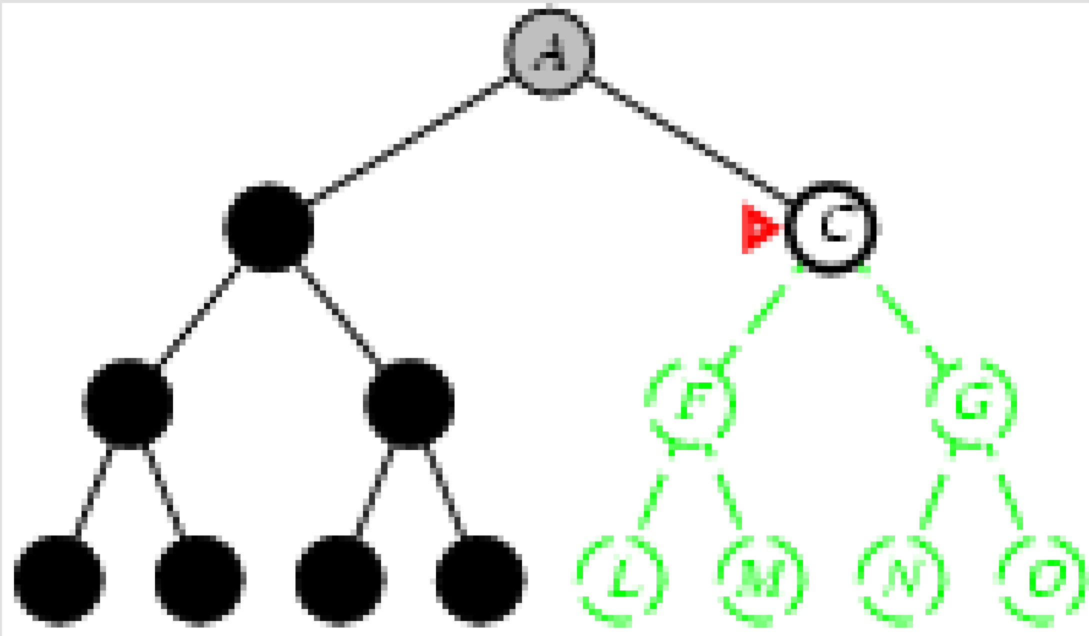
Depth-first Search (VIII)

- **Expand the deepest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: LIFO – insert successors in front of queue



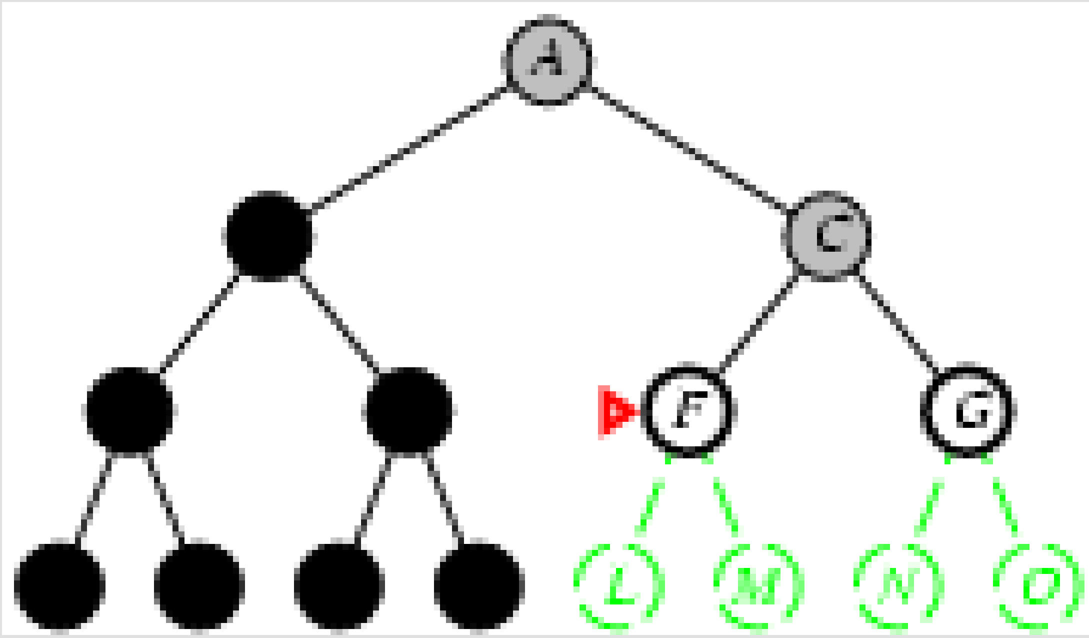
Depth-first Search (IX)

- **Expand the deepest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: LIFO – insert successors in front of queue



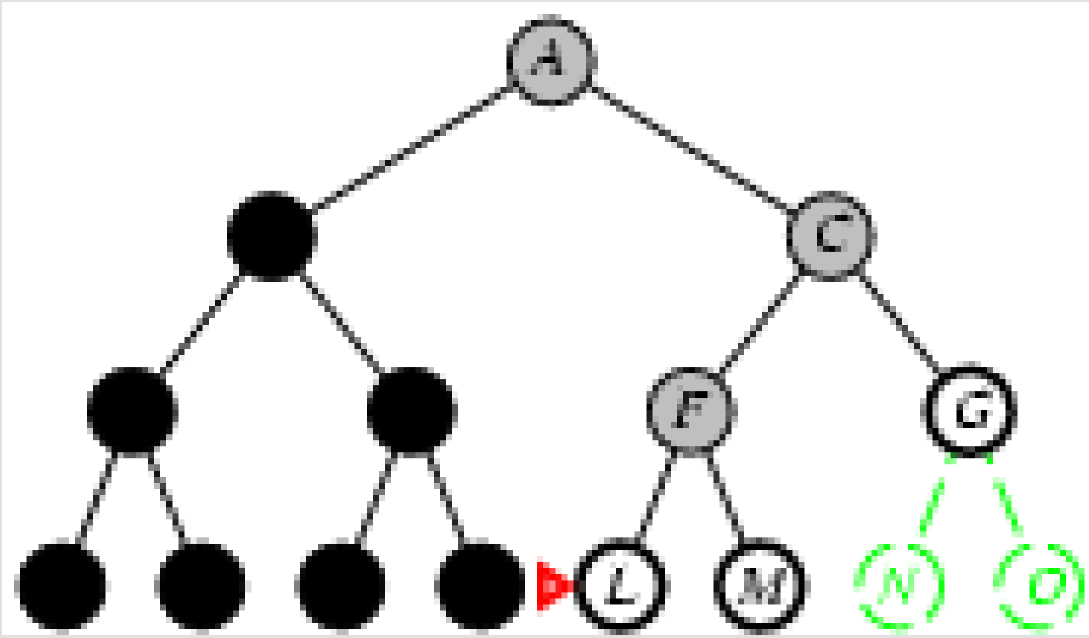
Depth-first Search (X)

- **Expand the deepest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: LIFO – insert successors in front of queue



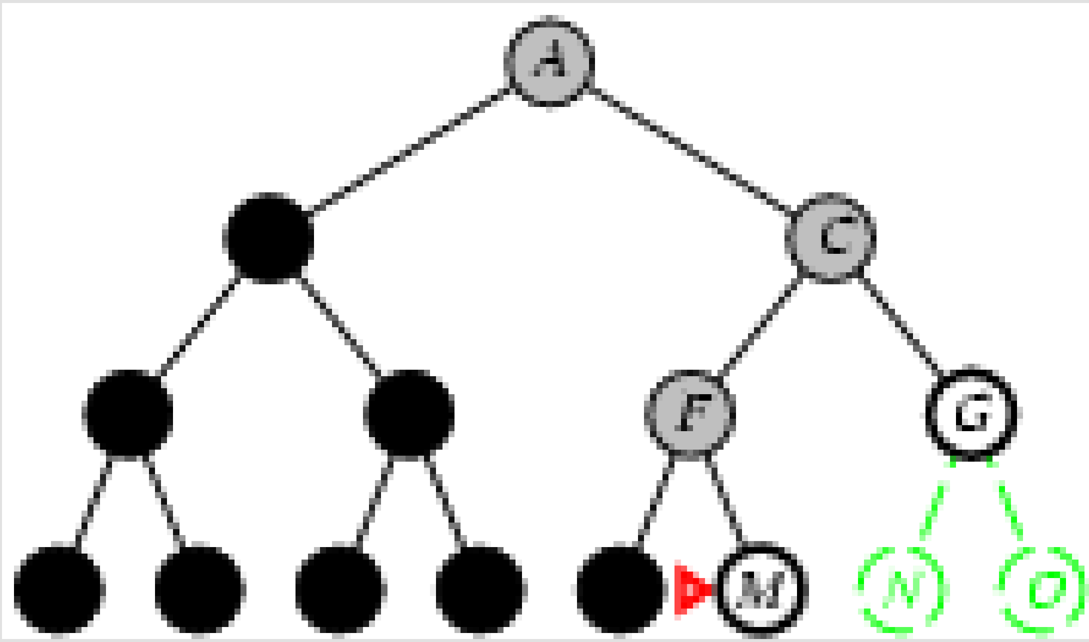
Depth-first Search (XI)

- **Expand the deepest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: LIFO – insert successors in front of queue



Depth-first Search (XII)

- **Expand the deepest unexpanded node**
- **Implementation: managing the frontier**
  - QUEUEING-FN: LIFO – insert successors in front of queue



Properties of Depth-first Search

- **Complete?**
  - Infinite-state spaces: No
  - Finite-state spaces: Yes, if we check for ancestors
- **Time?** **$O(b^m)$, terrible if m is much larger than d**
- **Space?** **$O(bm)$, i.e., linear space**
- **Optimal?** **No**

When all step costs are the same, will DFS find the optimal path?

Depth-limited Search

- **Depth-first search with depth limit *L***
  - nodes at depth *L* have no successors
  - returns *cut-off* if no solution is found
- **Complete? No if $d > L$**
- **Time?** $b + b^2 + b^3 + \ldots + b^L = b\frac{b^L - 1}{b - 1} \rightarrow O(b^L)$
- **Space?** $O(bL)$
- **Optimal? No**

When all step costs are the same, will DLS find the optimal path?

Iterative Deepening DF Search

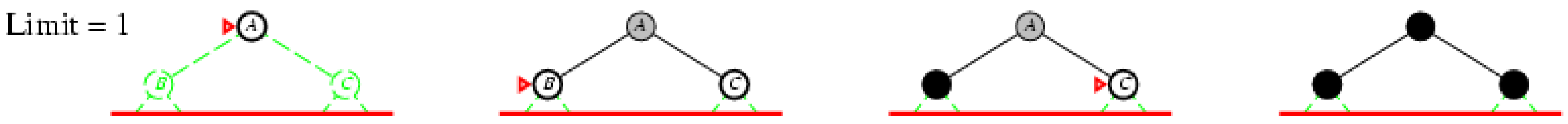
function ITERATIVE-DEEPENING-DF-SEARCH(*problem*) **returns** a solution or failure

- Initialize the frontier using the initial state of *problem*
- **For** *depth* ← 0 **to** ∞
  - *result* ← DEPTH-LIMITED-SEARCH(*problem*,*depth*)
  - **if** *result* ≠ *cut-off* **then return** *result*
- **end**

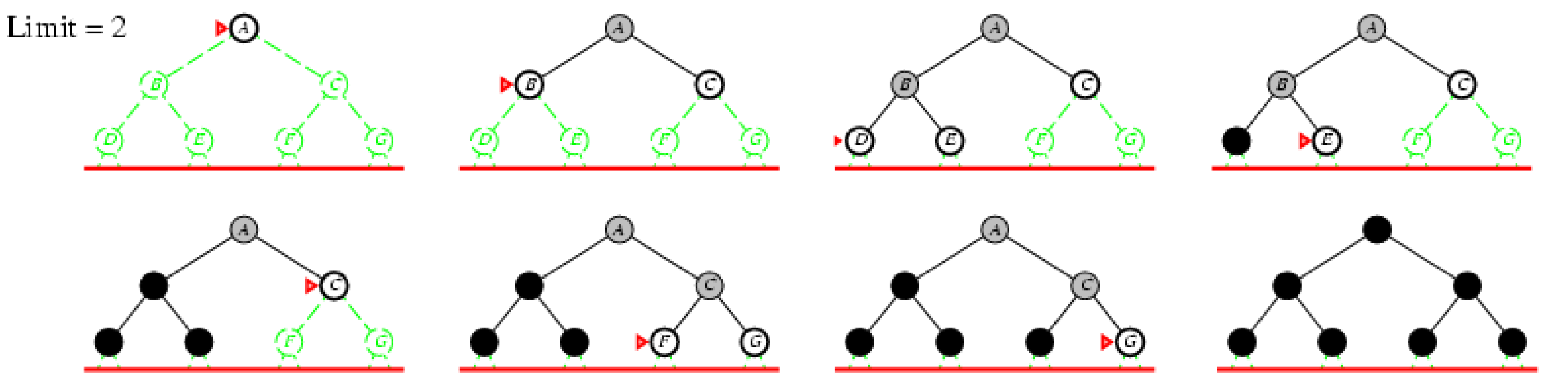
indicates failure

Iterative Deepening Search *depth*=0

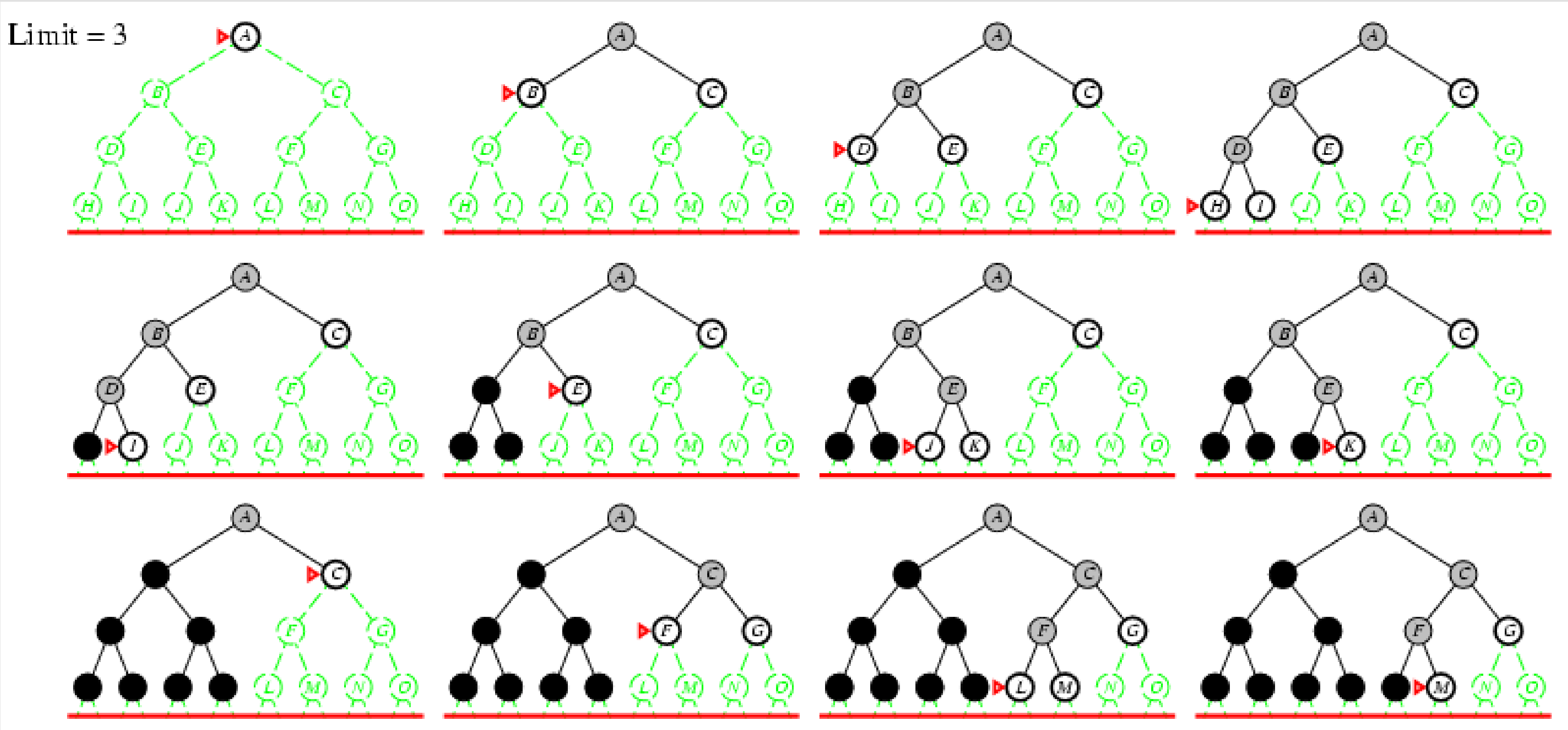
Iterative Deepening Search *depth*=1



Iterative Deepening Search *depth*=2



Iterative Deepening Search *depth*=3



Iterative Deepening Search – Generated Nodes

- **Number of nodes generated in a depth-limited search to depth d with branching factor b:**

$$N_{DLS} = b + b^2 + b^3 + \ldots + b^d = b\frac{b^d - 1}{b - 1} \to O(b^d)$$

- **Number of nodes generated in an iterative deepening search to depth d with branching factor b:**
  $N_{IDS} = db + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + b^d \to O(b^d)$
- **Example: For $b = 10$, $d = 6$,**
  - N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
  - N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
  - Overhead = $\frac{123{,}456 - 111{,}111}{111{,}111}$ = 11%

Properties of Iterative Deepening Search

- **Complete?** Yes
- **Time?**
  $db + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + b^d \rightarrow O(b^d)$
- **Space?** $O(bd)$
- **Optimal?** Yes, if step costs are identical

MONASH University
Information Technology

FIT5047 – Intelligent Systems

Informed Search Strategies:
Best-first Search

Heuristic (Informed) Graphsearch Procedures

- **Use *Heuristic Information* (domain dependent information) to help reduce the search**
  - Evaluation function – a real valued function used to compute the "promise" of a node

Heuristic Graphsearch: Definitions (I)

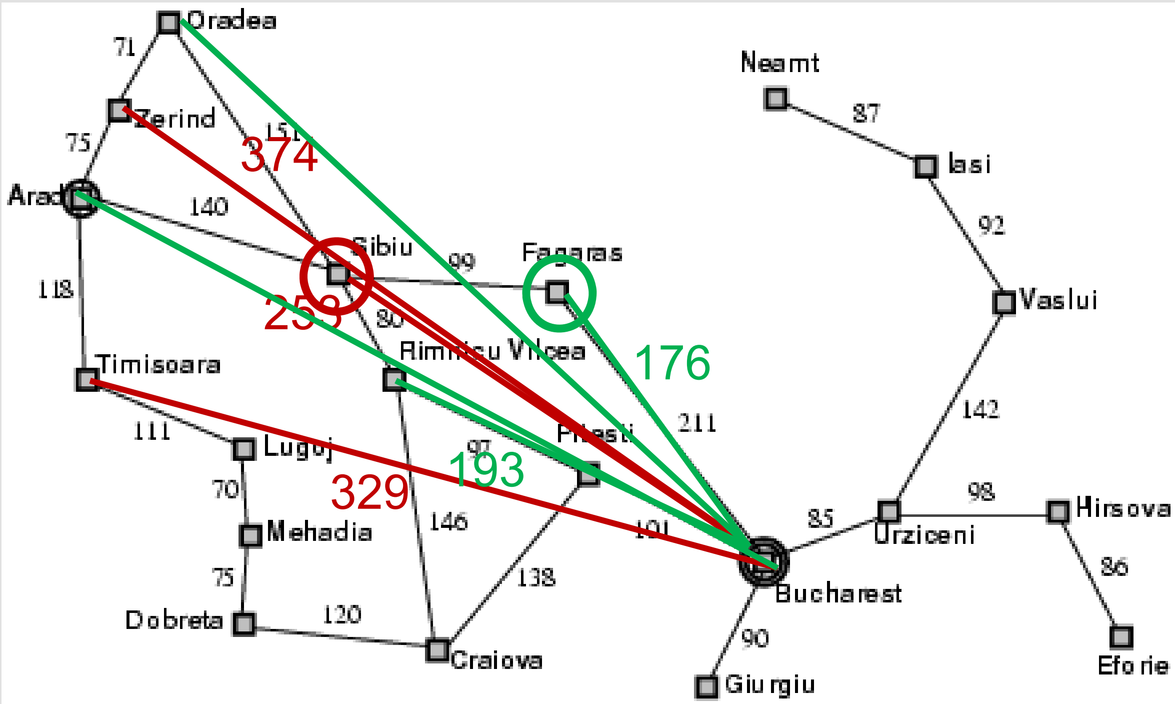
- $k(n_i,n_j)$ – actual cost of minimal cost path between n_i and n_j
- $h^*(n) = min\{k(n,t_i)\}$
  minimum of all the $k(n,t_i)$ over the entire set of goal nodes $\{t_i\}$
- $g^*(n) = k(s,n)$
  minimum cost from the start node s to n
- $f^*(n)=g^*(n)+h^*(n)$
  cost of an optimal path constrained to go through n
- $f^*(s)=h^*(s)$
  cost of an unconstrained optimal path from s to goal

Heuristic Graphsearch: Definitions (II)

- **$f(n)$ – <u>*estimate*</u> of the minimal cost path constrained to go through node n**
- **$g(n)$ – <u>*estimate*</u> of $g^*(n)$ ($g(n) \geq 0$)**
  Usual choice: Cost of the path in the search tree/graph from s to n ➔ $g(n) \geq g^*(n)$
- **$h(n)$ – heuristic function**
  Estimate of $h^*(n)$ ($h(n) \geq 0$)

Best-first Greedy Search

- **Expands the node that is closest to the goal among the *current options***
  - $f(n) = h(n)$
  - Example: $h_{SLD}(n)$ = Straight-Line Distance to the goal



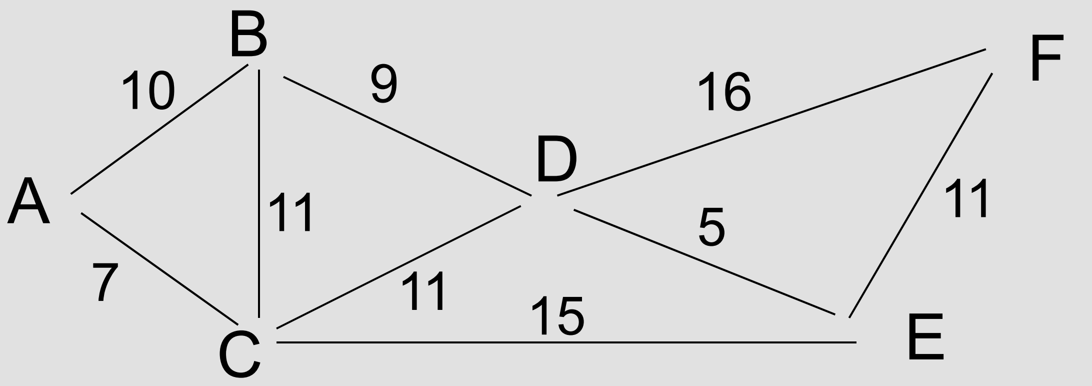
Properties of Best-first Greedy Search

- **Complete?**
  - Infinite-state spaces: No
  - Finite-state spaces: Yes, if we check for ancestors
- **Time?** $O(b^m)$
- **Space?** $O(b^m)$
- **Optimal?** **No**

Algorithm A

- **Graphsearch using the evaluation function** $f(n) = g(n) + h(n)$
- $g(n) \geq g^*(n)$; $h(n) \geq 0$
- **Expands next the node in the frontier with the smallest value of** $f(n)$

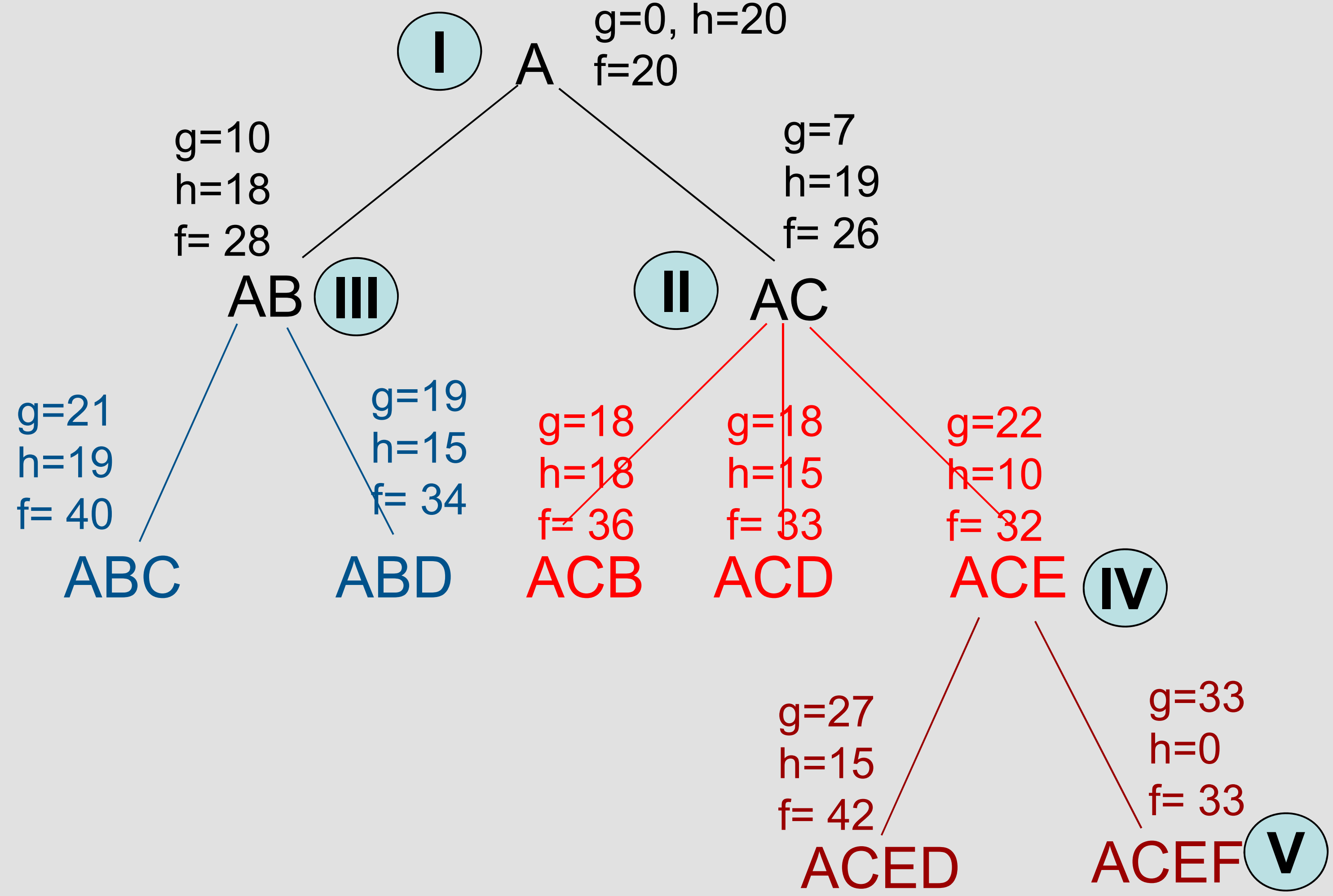
Algorithm A* Example – Shortest Path (I)



ROAD DISTANCES						
	A	B	C	D	E	F
A		10	7			
B			11	9		
C				11	15	
D					5	16
E						11

AIR DISTANCES						
	A	B	C	D	E	F
A		4	3	8	12	20
B			6	5	9	18
C				7	10	19
D					5	15
E						10

Algorithm A* Example – Shortest Path (II)



MONASH University
Information Technology

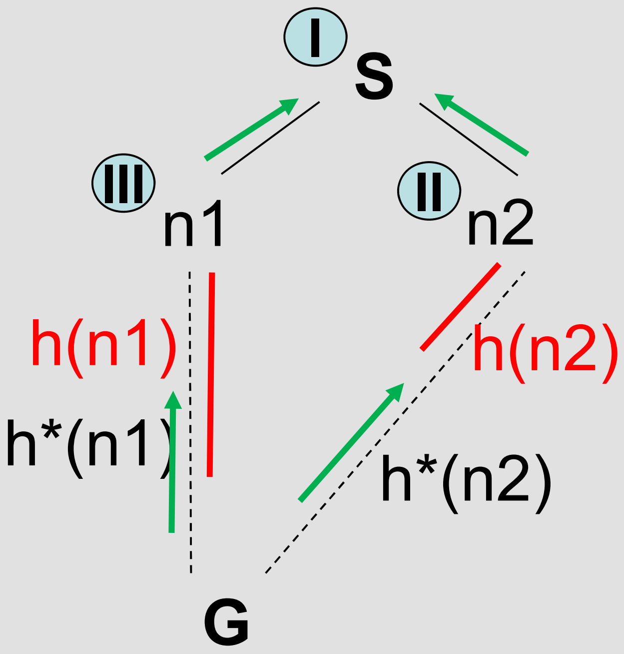
Algorithm A* = A + Admissible *h*

- ***Admissibility* of h:**
  If $\forall n \;\; h(n) \leq h^*(n)$
  Then A* is guaranteed to find the optimal solution (if it exists)
- ***Monotonicity (Consistency)* of h:**
  If $\forall n \;\; h(n) \leq c(n,m) + h(m)$
  where m is a child of n
  Then A* has found the optimal path to any node it selects for expansion
- **Optimality of A***
  – General graphsearch (Nilsson and classnotes) is optimal and terminates (if there is a solution) if $h(n)$ is <u>admissible</u>
  – Restricted graphsearch (Russell & Norvig) is optimal and terminates if $h(n)$ is <u>consistent</u>

h(n) – Admissibility

If $\forall n \;\; h(n) \leq h^*(n)$

Then A* is guaranteed to find the optimal solution (if it exists)



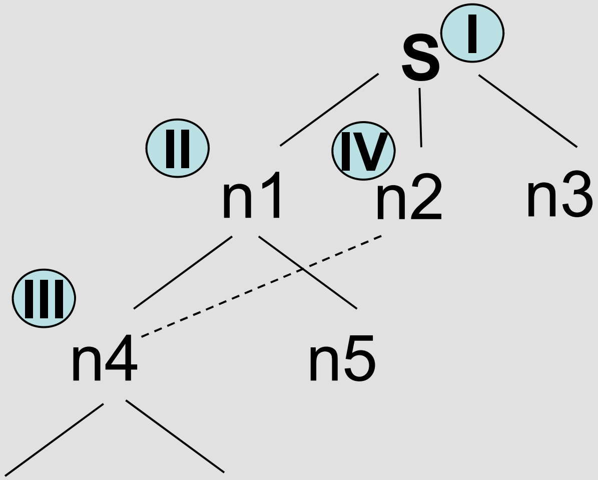
g(n1) = g(n2), h(n2) < h(n1)
OPEN={n2,n1}
n2 is expanded: g(G)=g(n2)+h*(n2)

If h(n1) ≥ h*(n2) then
  OPEN={G,n1}, G is expanded
Else
  OPEN = {n1,G}
  n1 expanded: g(G)=g(n1)+h*(n1)
but h*(n1) < h*(n2), so
recalculate f(G) and redirect pointer

h(n) – Monotonicity

If $\forall n \quad h(n) \leq c(n,m) + h(m)$, where m is a child of n

Then A* has found the optimal path to any node it selects for expansion



n1 is expanded: f(n1) ≤ f(ni) for i=2,3

n4 is expanded: f(n4) ≤ f(ni) for i=2,3,5

n2 is expanded: do we recalculate f(n4)?

$$f(n4) = g(n4) + h(n4) \leq g(n2)+h(n2) = f(n2)$$

monotonicity

$$= \underbrace{g(n1)+c(n1,n4)}_{\text{n4 through n1}} + h(n4) \leq g(n2)+h(n2) \leq \underbrace{g(n2)+c(n2,n4)}_{\text{n4 through n2}}+h(n4)$$

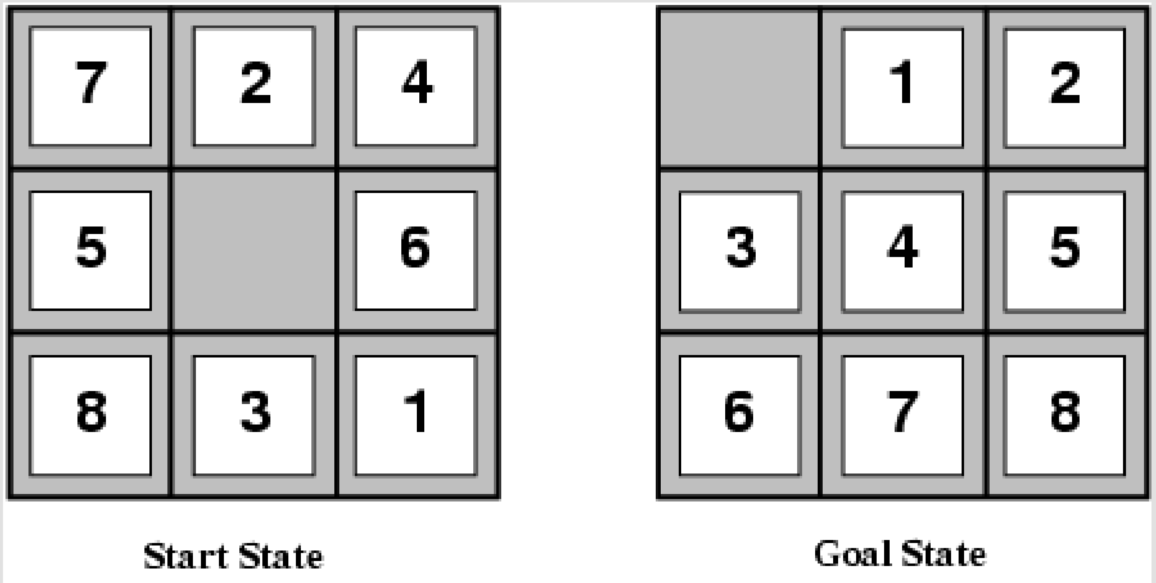
Properties of A and A*

	A	A*
Complete?	Yes	Yes
Time?	$O(b^d)$	$O(b^\Delta)$, where $\Delta \alpha \max\|h-h^*\|$
Space?	$O(b^d)$	$O(b^\Delta)$
Optimal?	No	Yes

Admissible heuristics: 8 Puzzle

- *$h_1(n)$* = **number of misplaced tiles**
- *$h_2(n)$* = **total *Manhattan distance* (# of squares from desired location of each tile)**

- $h_1(S) = ?$
- $h_2(S) = ?$



Relaxed problems

- **A problem with fewer restrictions on the actions is called a *relaxed problem***
- **The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem**
- **Examples:**
  - If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
  - If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Dominance

- **Given two <u>admissible heuristics</u> h_1 and h_2, if $h_2(n) \geq h_1(n)$ for all n then h_2 <u>dominates</u> h_1**

  → h_2 is better for search

- **If we have several admissible heuristics h_1, h_2, ..., h_n, none of which dominates, we can take the maximum:**

$$h(i) = \max\{h_1(i), h_2(i), ..., h_n(i)\}$$

Measuring Performance

Performance is often measured by _effective branching factor (EBF) b^*_

- **if N nodes are generated, this is the branching factor that a uniform tree of depth d would need to have in order to contain N+1 nodes, i.e.,**

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d = \frac{(b^*)^{d+1} - 1}{b^* - 1}$$

  – **Example:** N=52, d=5 → $b^* \approx 1.9$

→ **Experimental measurements of b^* on a small set of problems can provide an idea of a heuristic's usefulness**

  – A good heuristic yields $b^* \approx 1$

Summary: Tree- and Graph-Search

- **When an agent is not clear on which immediate action is best, it can consider possible sequences of actions: search**
- **Before solutions can be found, the agent must formulate a goal and a problem, which consist of:**
  - the initial state; a set of operators; a set of constraints; a goal test function; a path cost function
- **A single general search algorithm can be used to solve any search problem**
- **Different search strategies yield different algorithms, which are judged on the basis of:**
  - completeness; optimality; time complexity; space complexity

MONASH University
Information Technology

FIT5047 – Intelligent Systems

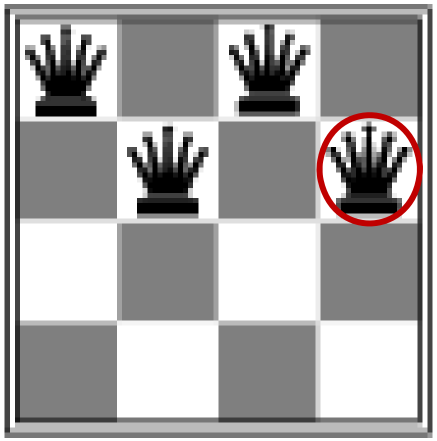
Irrevocable Search Algorithms

Local Search Algorithms

- **In many optimization problems, the goal state is the solution**
- **State space = set of "complete" configurations**
- **Find configuration satisfying constraints, e.g., n-queens problem**
- **In such cases, we can use local search algorithms**
  - keep a single "current" state, try to improve it

Example: *n*-Queens Problem

- **Put *n* queens on an *n* × *n* board with no two queens on the same row, column or diagonal**



Hill Climbing Algorithm

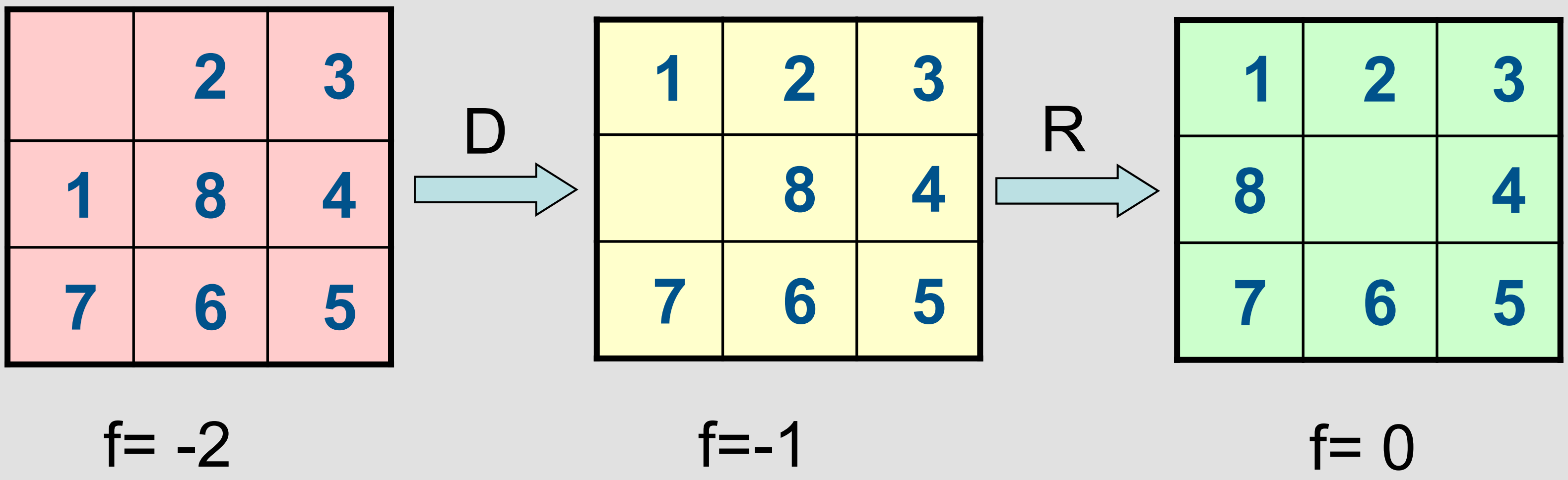
Procedure Hill Climbing(current-state)

1. **If current-state = goal-state Then return it**
2. **Else until a solution is found or no more operators can be applied do**
   a. Select an operator that has not been applied yet to current-state and apply it to generate new-state
   b. Evaluate new-state:
      i. **If** new-state = goal-state **Then** return it and quit
      ii. **Elseif** new-state is better than current-state **Then** current-state ← new-state

Steepest ascent hill-climbing: select the best operator

Hill Climbing – Example 8 Puzzle (I)

- **f = - { number of tiles out of place }**



Hill Climbing – Example 8 Puzzle (II)

- **f = - { number of tiles out of place }**

Current

1	2	5
	8	4
7	6	3

f= -2

Goal

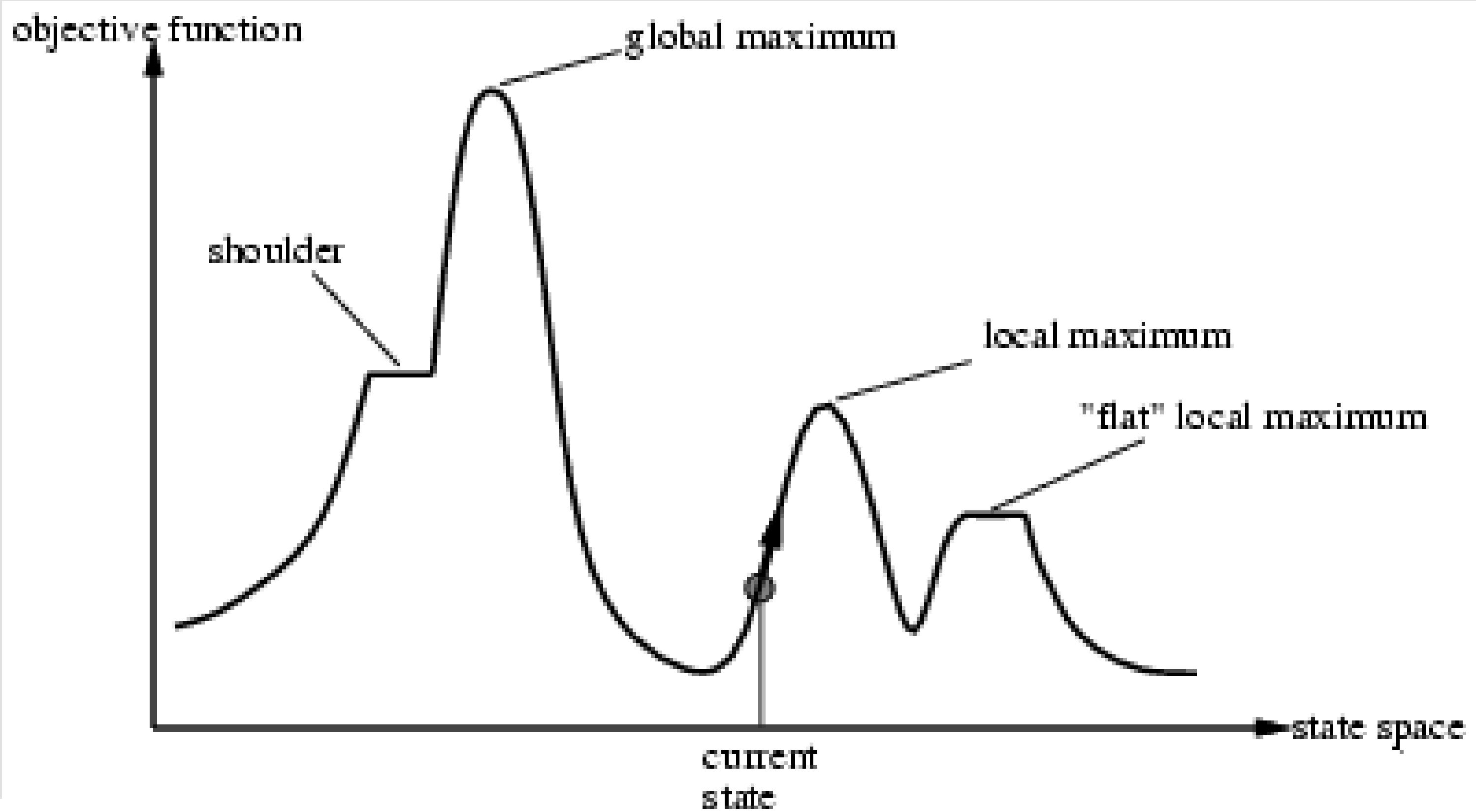
1	2	3
	8	4
7	6	5

f= 0

Stuck in local maximum

Hill-climbing Search

- **Problem: depending on initial state, can get stuck in local maxima**



Local Beam Search

- **Keep track of *k* states rather than just one**
- **Start with *k* randomly generated states**
- **At each iteration, all the successors of all *k* states are generated**
  - If any one is a goal state, stop
  - Else select the *k* best successors from the complete list and repeat

Simulated Annealing

- Based on the physical process of annealing
- Idea: escape local maxima/minima by allowing some "bad" moves, but gradually decrease their frequency
- *Temperature* (*T*) – the temperature at which the annealing takes place
- *Annealing schedule* – the rate at which the temperature is lowered

Simulated Annealing Algorithm

Procedure Simulated Annealing(current-state)

1. **If** current-state = goal-state **Then** return it and quit
2. BestSoFar←current-state
3. Initialize ***T*** according to the annealing schedule
4. Until no more operators can be applied do
   a. Select an operator that has not been applied yet to current-state and apply it to generate new-state
   b. Evaluate new-state. Compute:
   ΔE = Value(current-state) - Value(new-state)
      i. **If** new-state = goal-state **Then** return it and quit
      ii. **Elseif** ΔE<0 (new-state is better than current-state) **Then**
      current-state←new-state
      If new-state is better than BestSoFar **Then** BestSoFar←new-state
      iii. **Else** with probability Pr=$e^{-\Delta E/T}$ current-state←new-state
   c. Revise ***T*** according to the annealing schedule
   d. If ***T***=0 Then return BestSoFar

Maximization problem

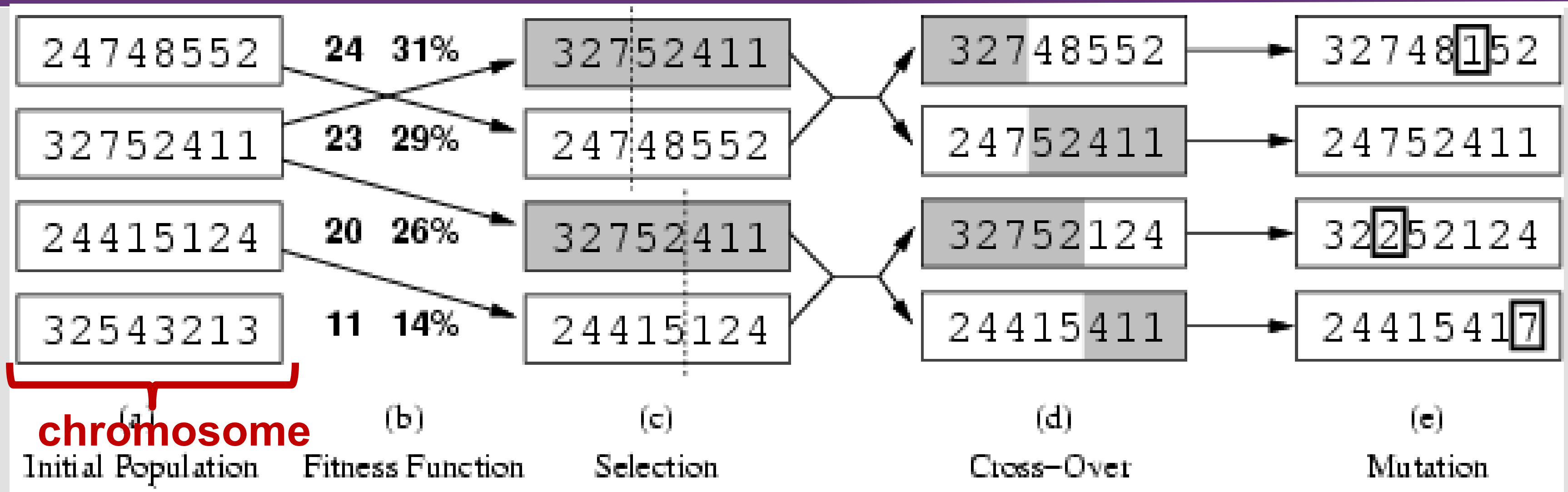
Properties of Simulated Annealing Search

- **One can prove: If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1**
- **It is widely used in VLSI layout and airline scheduling**

Genetic Algorithms

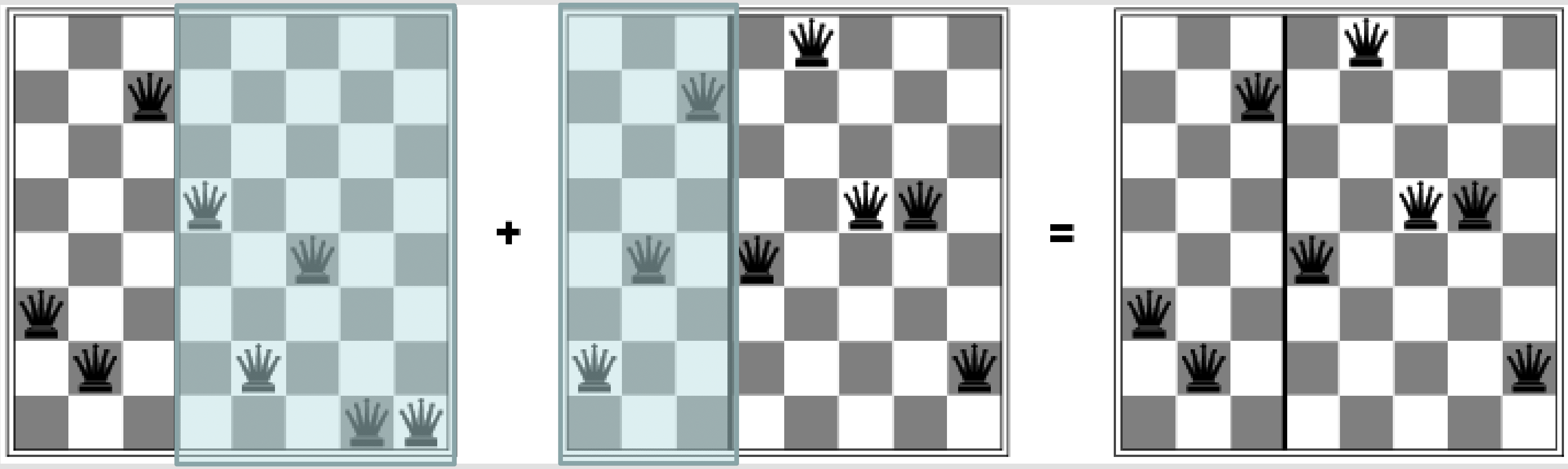
- **Start with a *population* of *k* randomly generated states**
- **A state (*chromosome*) is represented as a string over a finite alphabet of *genes* (often a string of 0s and 1s)**
- **A successor state is generated by combining two parent states**
- **Evaluation function (*fitness function*):**
  - Higher values for better states
- **Produce the next generation of states by *selection, crossover* and *mutation***

GAs: Example 8-Queens Problem (I)

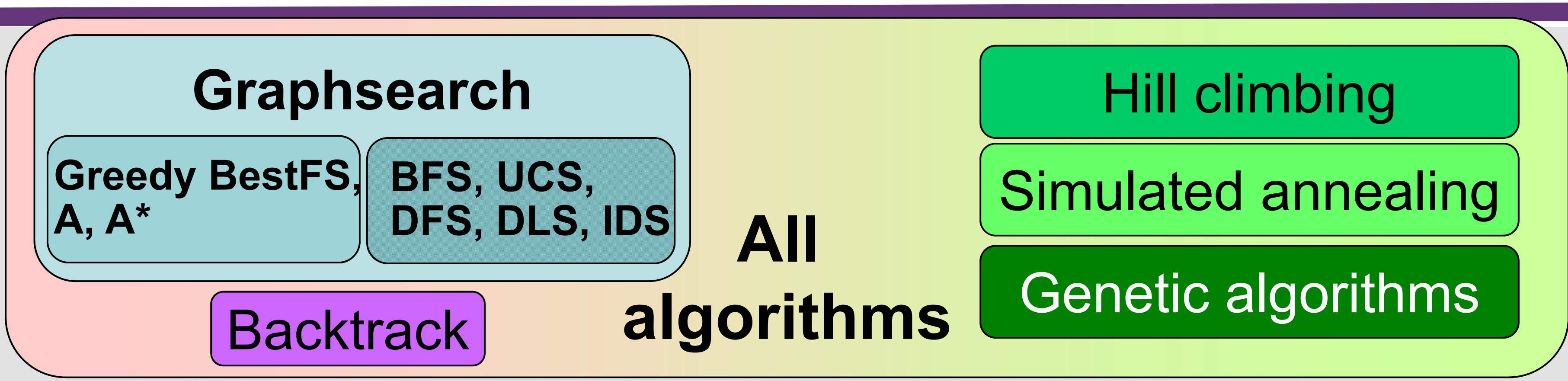


- **Representation:**
  - **Gene**: row # (between 1 and 8) of the queen that is in column i
  - **Chromosome**: 1 gene per column (8 genes per chromosome)
- **Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)**
  - Probability of selection: $\frac{24}{24+23+20+11}$=31%, $\frac{23}{24+23+20+11}$=29%

GA Crossover: Example 8-Queens Problem



Search Algorithms – A Perspective



- **Informedness in Graphsearch depends on g and h**
  - A $\quad f(n) = g(n)+h(n) \quad (g(n) \geq g^*(n), h(n) \geq 0)$
  - A* $\quad (g(n) \geq g^*(n), h(n) \leq h^*(n))$
- **Uninformed Graphsearch**
  - BFS є A* when $g(n)$=depth and $h(n)=0$
  - UCS є A* with $g(n) \geq 0$ and $h(n)=0$
  - DFS, DLS, IDS є Graphsearch, DFS, DLS, IDS $\not\in$ A
- **Informed Graphsearch**
  - BestFirst Greedy with $g(n)=0$ and $h(n) \geq 0$ $\not\in$ A

MONASH University
Information Technology

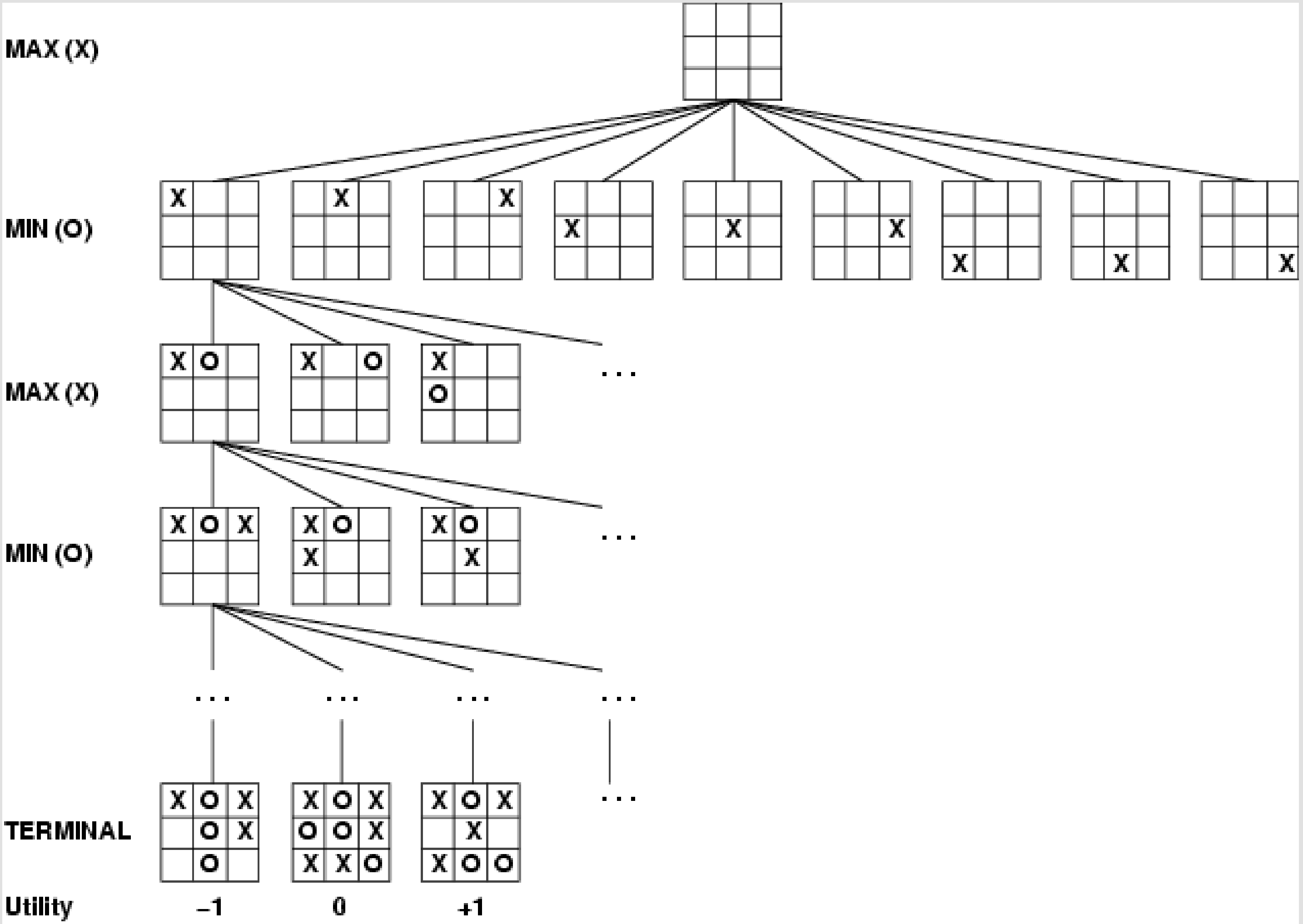
FIT5047 – Intelligent Systems

Adversarial Search Algorithms

Searching Game Trees

- **Two person, perfect information games**
- **Conventions:**
  - Players are MAX and MIN
    - A position favourable to MAX has a value > 0 (winning is often ∞)
    - A position favourable to MIN has a value < 0 (winning is often $-\infty$)
  - Goal: find a winning strategy for MAX
    - For all nodes representing a game situation where it is MIN's move next, show that MAX can win from **every** position to which MIN might move
    - For all nodes representing a game situation where it is MAX's move next, show that MAX can win from **just one** position to which MAX might move

Game Tree (2-player, Deterministic, Turns)

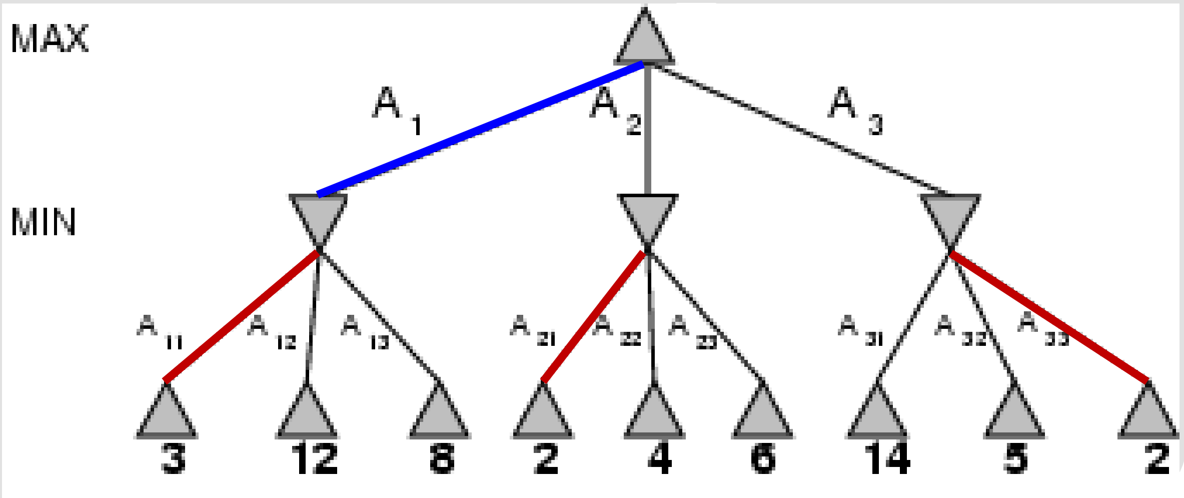


Games versus Search Problems

- **Unpredictable opponent → must specify a move for every possible opponent reply**
- **Time limits: not all games can be searched to the end → find a good first move**

Minimax Ideas

- **If MAX were to choose among tip nodes, s/he would take the node with the largest value**
- **If MIN were to choose among tip nodes, s/he would take the node with the smallest value**
- **Choose move to the position with highest *minimax value*: best achievable payoff against best play**
- **E.g., 2-ply game:**



Minimax Algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(*s*, *a*)))
  return v

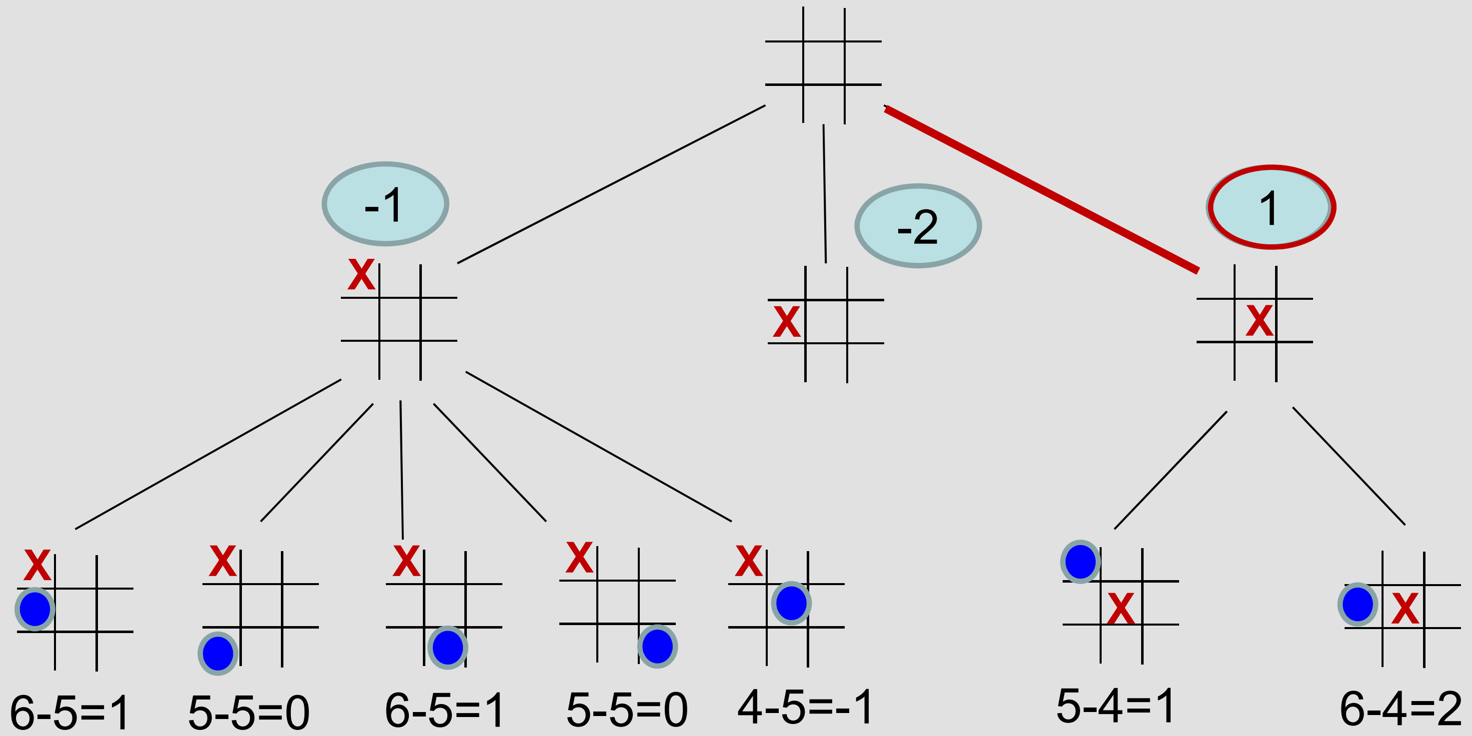
---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(*s*, *a*)))
  return v

Minimax Example: Tic-Tac-Toe

- **Evaluation function:**

  { # of rows, columns, diagonals available to MAX –
  # of rows, columns, diagonals available to MIN }



Properties of Minimax

Minimax performs a <u>complete depth first exploration</u> of the game tree

- **<u>Complete?</u>** **Yes (if tree is finite)**
- **<u>Optimal?</u>** **Yes (against an optimal opponent)**
- **<u>Time complexity?</u>** $O(b^m)$
- **<u>Space complexity?</u>** $O(bm)$ **(depth-first exploration)**

- **For chess, b ≈ 35, m ≈100 for "reasonable" games**
  → exact solution completely infeasible

Resource Limits

- **Suppose we have 100 secs per move, and we explore 10^4 nodes/sec**
  → 10^6 nodes per move
- **Standard approach:**
  - **Cutoff test –** depth limit (perhaps add quiescence search)
  - **Evaluation function –** estimates the desirability of a position
    - E.g., for chess typically a linear weighted sum of features
      $\mathrm{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$,
      where $w_1 = 9$ and
      $f_1(s) = (\text{\# of white queens}) - (\text{\# of black queens})$
  - **Forward pruning**
    - beam search that looks only at n-best moves

Definitions: α and β Values

- **α-value of a MAX node – current <u>largest</u> final backed-up value of its successors**
  - α-value is the <u>lower</u> bound for a MAX backed-up value
- **β-value of a MIN node – current <u>smallest</u> final backed-up value of its successors**
  - β-value is the <u>upper</u> bound for a MIN backed-up value

α-β Procedure

- **Rules for discontinuing the search:**
  - **α cut-off**: search can be discontinued below any **MIN** node having a β-value ≤ **α-value** of **any** of its MAX node ancestors
    > The **final backed-up value** of this MIN node is set to its β-value
  - **β cut-off**: search can be discontinued below any **MAX** node having an α-value ≥ **β-value** of **any** of its MIN node ancestors
    > The **final backed-up value** of this MAX node is set to its α-value

Termination Condition

- **All the successors of the start node are given final backed-up values**
- **The best first move is that which creates the successor with the highest backed-up value**

The α-β Algorithm (I)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

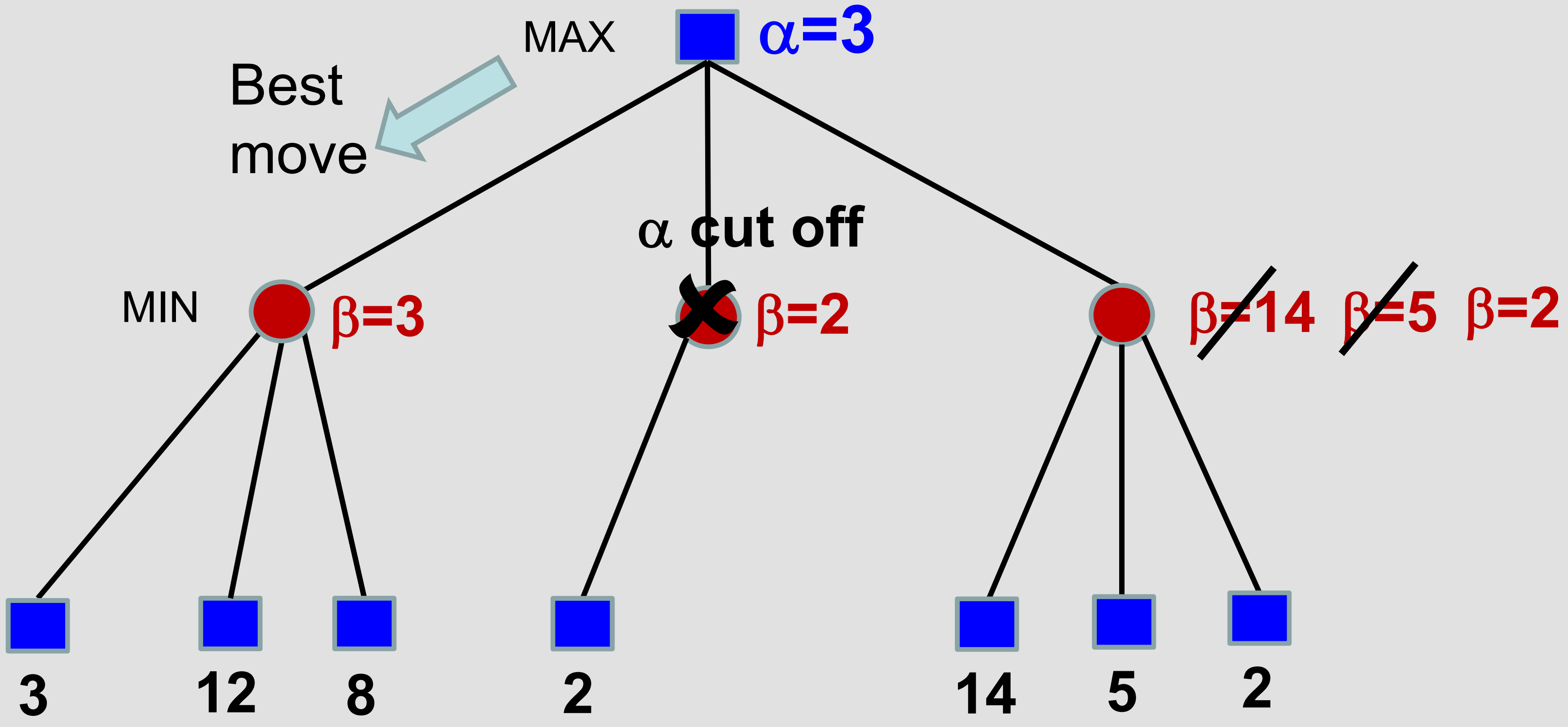
---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    $\boxed{\textbf{if } v \geq \beta \textbf{ then return } v}$ β cut-off
    $\alpha \leftarrow$ MAX(α, v)
  return v

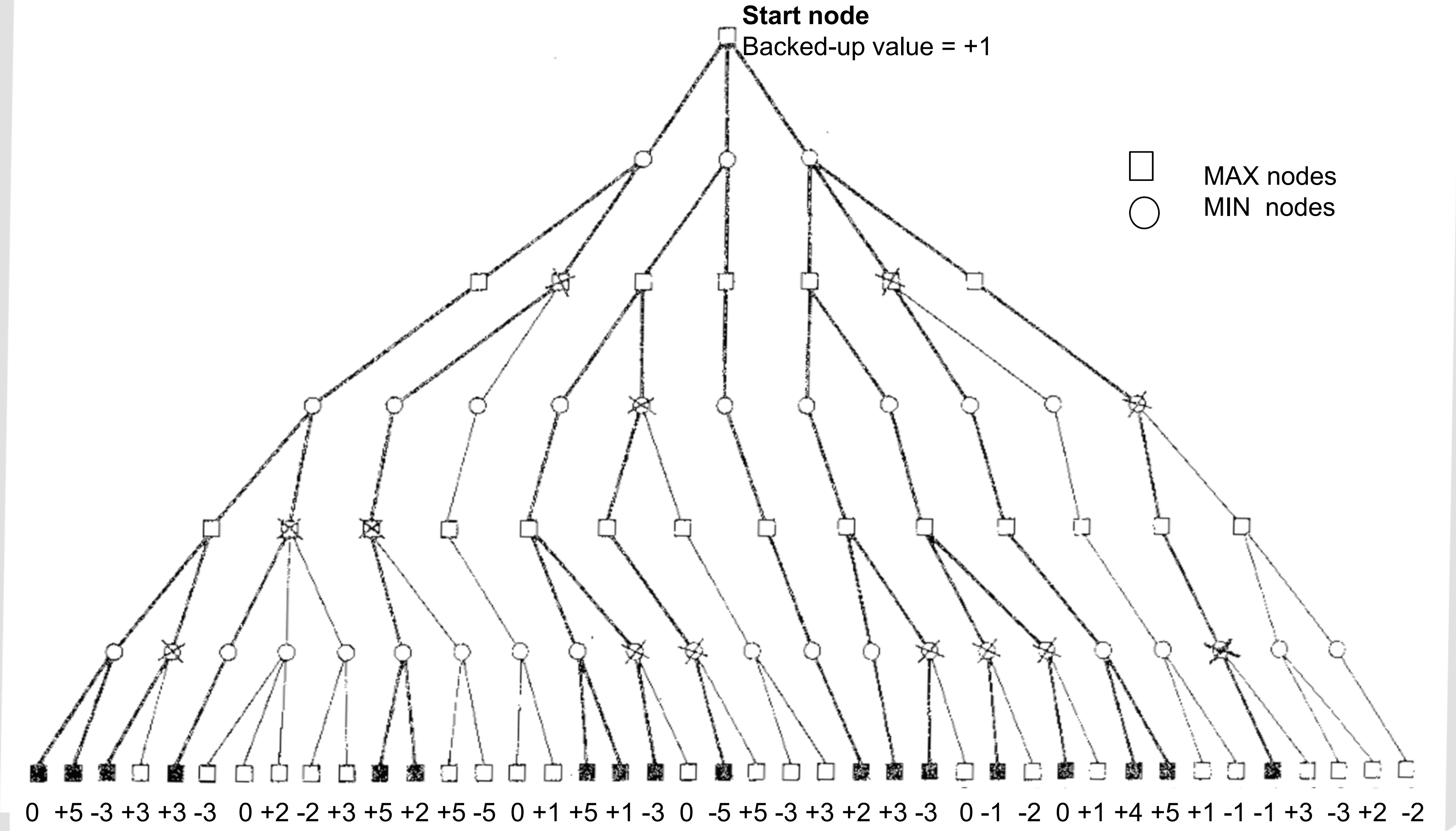
---

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    $\boxed{\textbf{if } v \leq \alpha \textbf{ then return } v}$ α cut-off
    $\beta \leftarrow$ MIN(β, v)
  return v

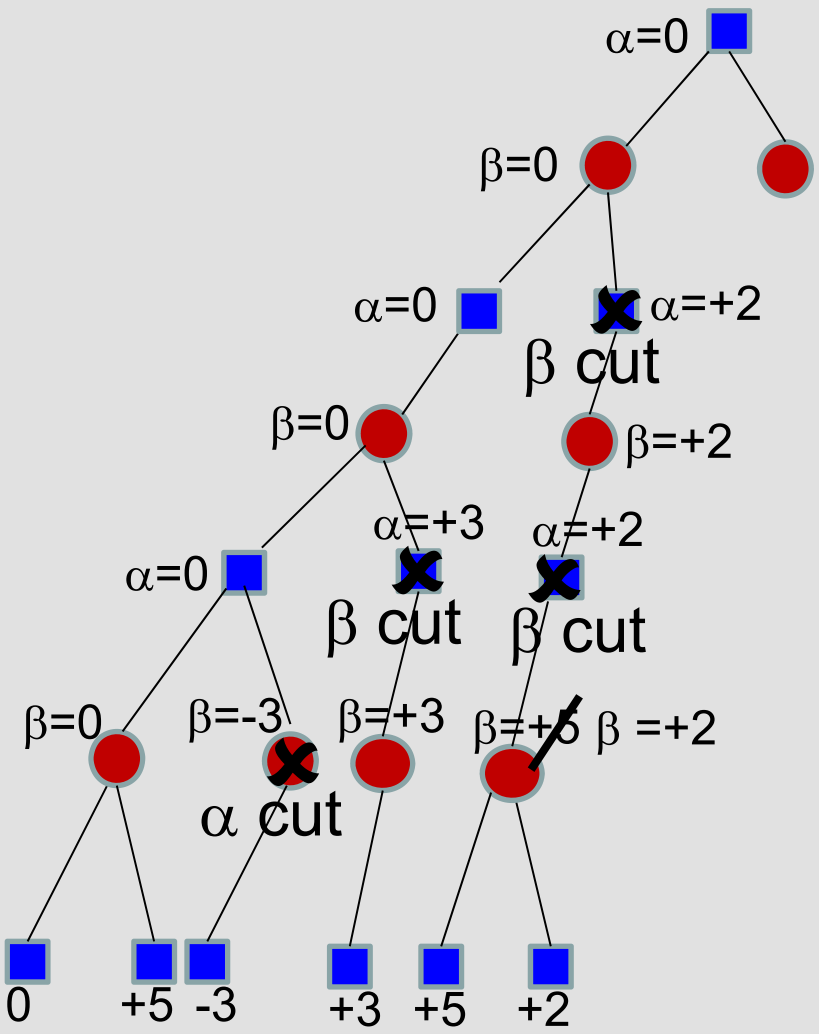
α-β Pruning – Example



α-β Pruning – Large Example



α-β Pruning – Part of Large Example



MONASH University
Information Technology

Move Ordering

- **The effectiveness of the αβ algorithm depends on the order in which states are examined**
- **With perfect ordering, time complexity = $O(b^{m/2})$**

  →depth of search can be doubled

- **Adding dynamic ordering schemes brings us close to the theoretical limit**

Deterministic Games in Practice

- **Checkers**: Chinook defeated the world champion in an abbreviated game in 1990. It uses αβ search combined with a pre-computed database defining perfect play for 39 trillion endgame positions.
- **Chess**: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 30 billion positions per move (200 million per second), normally searching to depth 14, and extending the search up to depth 40 for promising options. Heuristics reduce the EBF to about 3.
- **Othello**: In 1997, a computer defeated the world champion 6-0. Humans are no match for computers.
- **Go**: $b > 361$, which is too large for αβ. In 2016, AlphaGo, which uses Deep Learning, beat the world champion 4-1.

Summary: Adversarial Search

Games illustrate important points about AI

- **Perfection is unattainable → must approximate**
- **Force us to think about what to think about, e.g., nodes to keep/discard**

Reading

- **Russell, S. and Norvig, P. (2010), *Artificial Intelligence – A Modern Approach* (3rd ed), Prentice Hall**
  - Chapter 7, Sections 7.1, 7.3 (only backtrack algorithm)
  - Chapter 3 (excluding 3.5.3, 3.5.4, 3.6.3, 3.6.4)
  - Chapter 4, Section 4.1
  - Chapter 5, Sections 5.1-5.4

Next Lecture Topic

- **Lecture Topic 4**
  - Knowledge representation