Part 2

9.2 Problem solving and searching techniques: Definition, Problem as a state space search, Problem formulation, Well-defined problems, Constraint satisfaction problem, Uninformed search techniques (Depth First Search, Breadth First Search, Depth Limited Search, Iterative Deepening Search, Bidirectional Search), Informed Search (Greedy Best first search, A* search, Hill Climbing, Simulated Annealing), Game playing, Adversarial search techniques, Mini-max Search, and Alpha-Beta Pruning. (ACtE0902)

Problem solving through search

- In which we see **how an agent can find a sequence of actions that achieves its goals when no single action will do**.
- **Problem solving is a agent based system that finds sequence of actions that lead to desirable states from the initial state.**

Problem-solving agents

More details on "states" soon.

Problem solving agents are goal-directed agents:

1. **Goal Formulation: Set of one or more (desirable) world states (e.g. checkmate in chess).**
2. **Problem formulation: What actions and states to consider given a goal and an initial state.**
3. **Search for solution: Given the problem, search for a solution --- *a sequence of actions to achieve the goal starting from the initial state*.**
4. **Execution of the solution**

Note: Formulation feels somewhat "contrived," but was meant to model very general (human) problem solving process.

Example: Path Finding problem

Formulate goal:

- be in Bucharest (Romania)

Formulate problem:

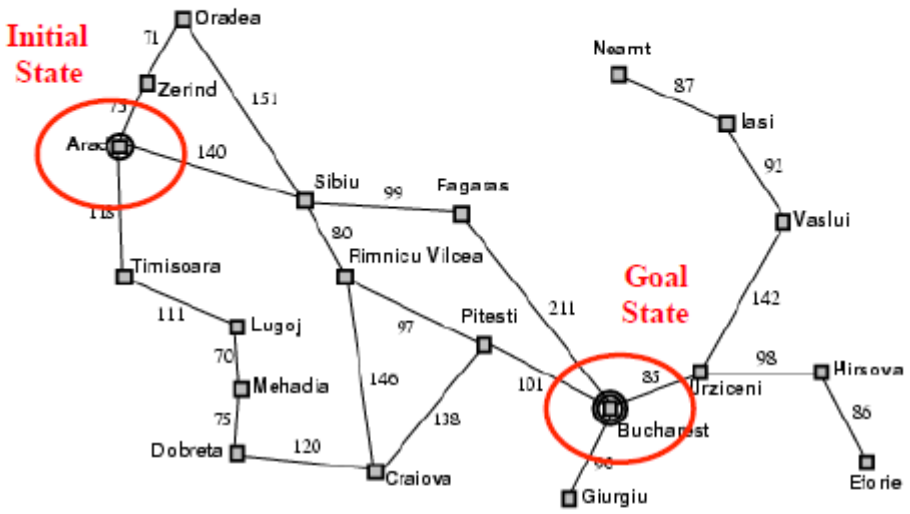
- action: drive between pair of connected cities (direct road)
- state: be in a city (20 world states)

Find solution:

- sequence of cities leading from start to goal state, e.g., Arad, Sibiu, Fagaras, Bucharest

Execution

- drive from Arad to Bucharest according to the solution



Environment: fully observable (map), deterministic, and the agent knows effects of each action. Is this really the case?

Note: Map is somewhat of a "toy" example. Our real interest: *Exponentially large spaces*, with e.g. 10^100 or more states. Far beyond full search. Humans can often still handle those!
One of the mysteries of cognition.

State space: the set of all states reachable from the initial state by *any* sequence of actions

Formalizing Search

A search problem has five components:

S, I, G, actions, cost

1. State space S : all valid configurations
2. Initial states $I \subseteq S$: a set of start states $I = \{(FCDS,)\} \subseteq S$
3. Goal states $G \subseteq S$: a set of goal states $G = \{(,FCDS)\} \subseteq S$
4. An action function *successors*(*s*) $\subseteq S$: states reachable in one step (one arc) from *s*

   successors((FCDS,)) = {(CD,FS)}
   successors((CDF,S)) = {(CD,FS), (D,FCS), (C,FSD)}
5. A cost function cost(*s*, *s'*): The cost of moving from *s* to *s'*

- The goal of search is to find a solution path from a state in *I* to a state in *G*

1. All of the following are true except:

- A. A State is a representation of all necessary information about the environment
- B. A state captures all the relevant information
- C. A state captures all the information
- D. The number of actions needed depends on how the world states are represented

2. An action is also known as :

A. Operator
B. Move
C. Both A and B
D. All of above

A well defined problem consist of

- A. Initial state, actions, and transition model,
- B. Goal test
- C. Path Cost
- D. All of the above

A successor is

- A. any state reachable from a given state by a single action.
- B. set of all state reachable from a given state
- C. None
- D. All of the above

Abstraction

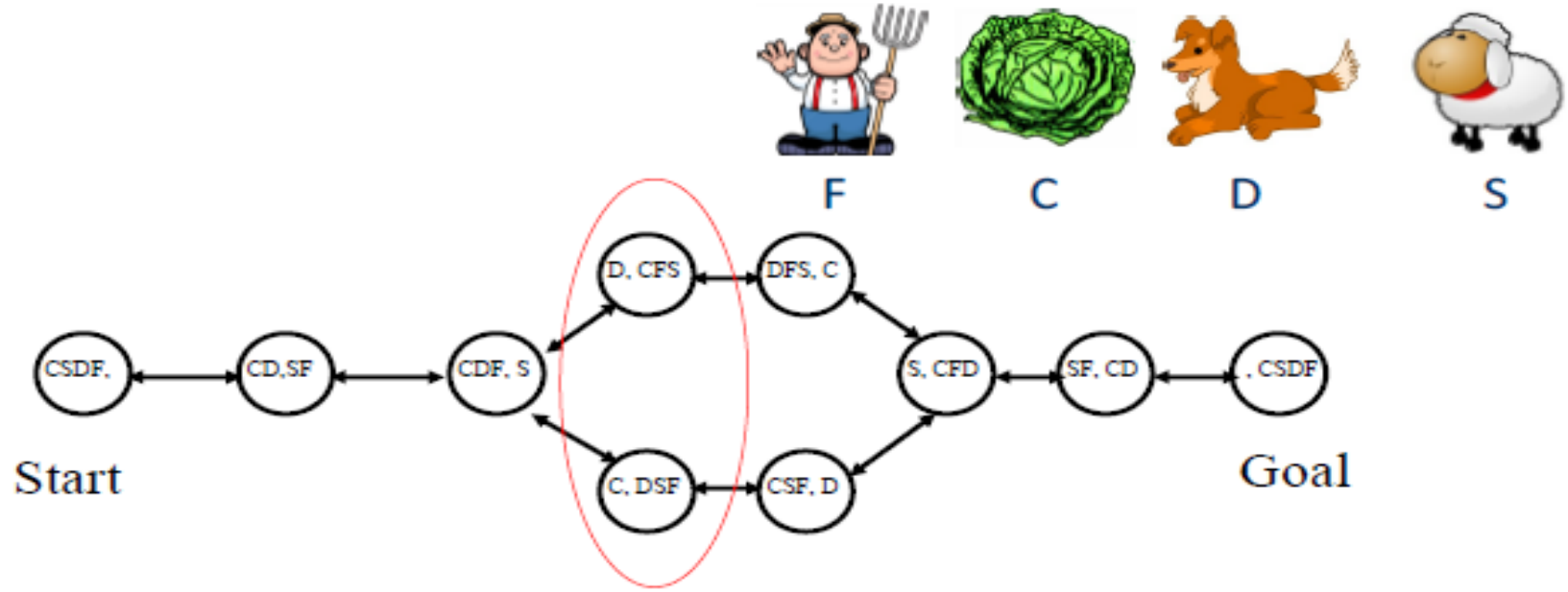
- For a taxi driving the state of the world includes so many things:
- the traveling companions,
- the current radio program,
- the scenery out of the window, the proximity of law enforcement
- officers, the distance to the next rest stop, the condition of the road, the weather, and so on.
- All these considerations are left out of our state descriptions **because they are irrelevant to the problem of finding a route to Bucharest. Hence many things are abstracted**

The process of removing detail from a representation

- A. Encapsulation
- B. Polymorphism
- C. Abstraction
- D. Goal Test

State Space

State Space = A Directed Graph

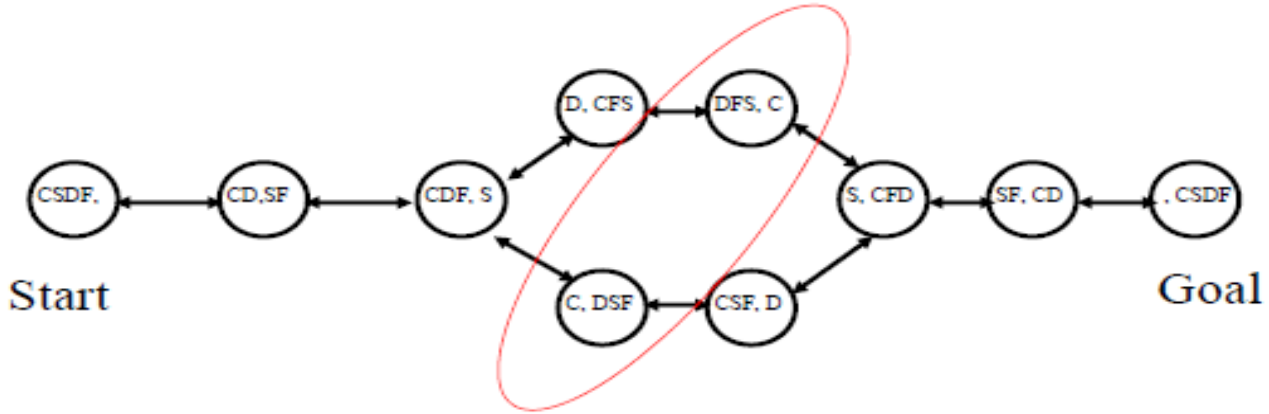


- In general, there will be many generated, but un-expanded, states at any given time during a search
- One has to choose which one to "expand" next

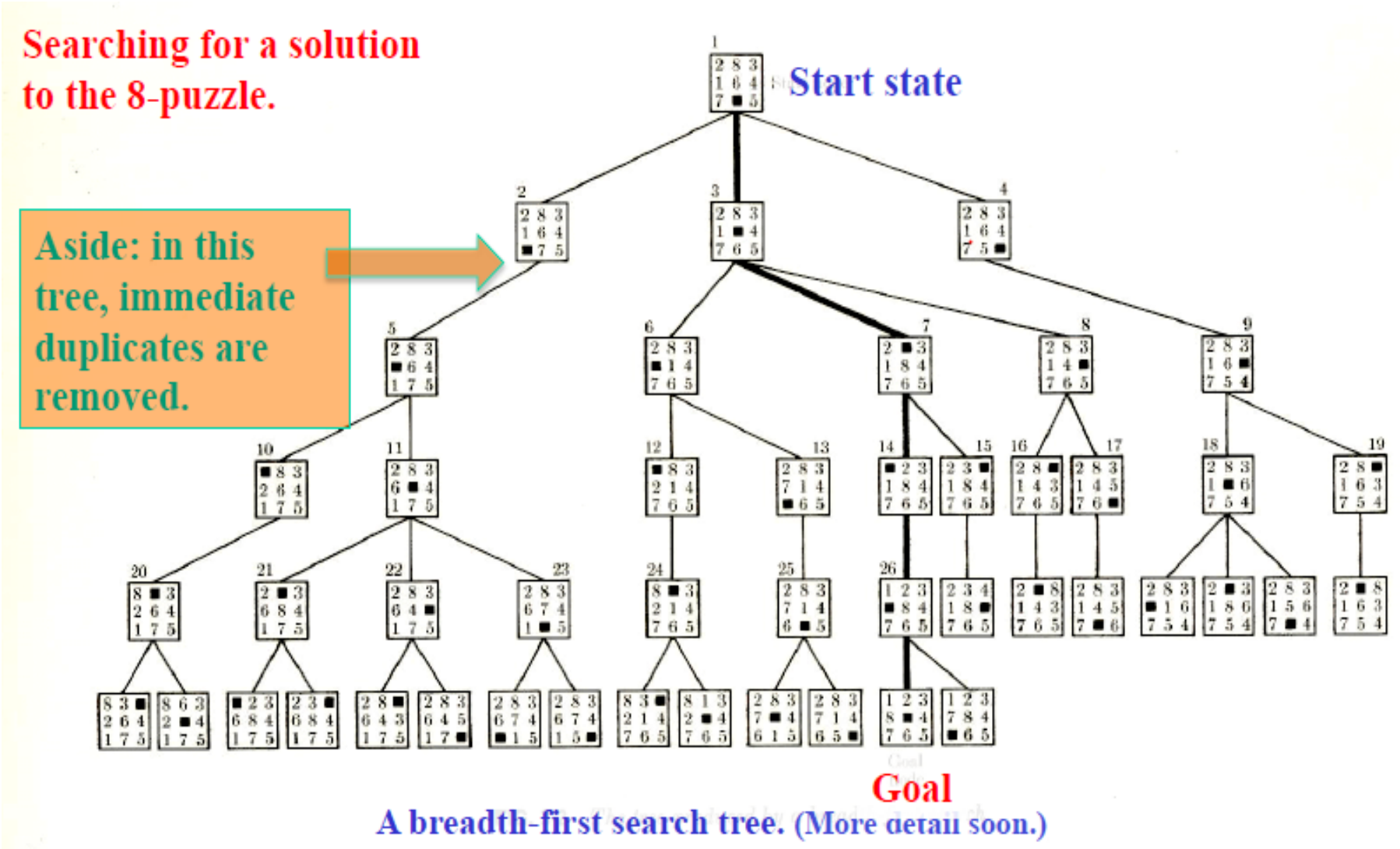
Frontier –The set of all nodes that are available for expansion

Different Search Strategies

- The generated, but not yet expanded, states define the *Frontier* (aka *Open* or *Fringe*) set
- The essential difference is, which state in the Frontier to expand next?

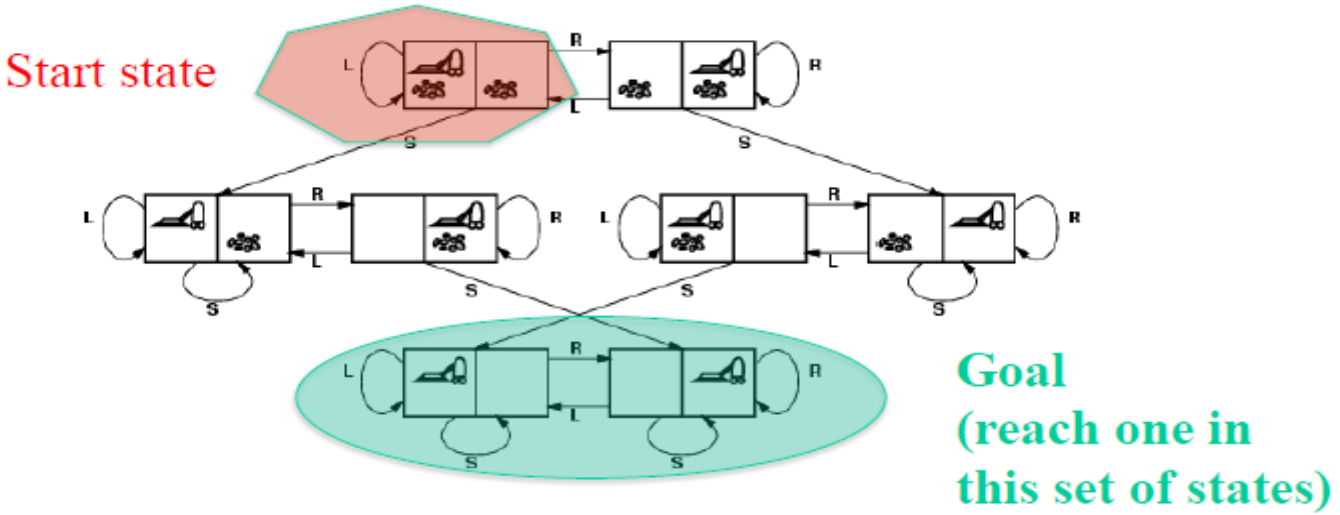


The essence of any Search strategy is → Which path to follow or which frontier node to expand →depending on it we have various search algorithms and it's the essence of any search **algorithm-Below search strategy is the Bread First Search (BFS)**



Observe the state space in below figure

Example: Vacuum world state space graph



states? The agent is in one of 8 possible world states.

actions? *Left, Right, Suck [simplified: left out No-op]*

goal test? No dirt at all locations (i.e., in one of bottom two states).

path cost? 1 per action

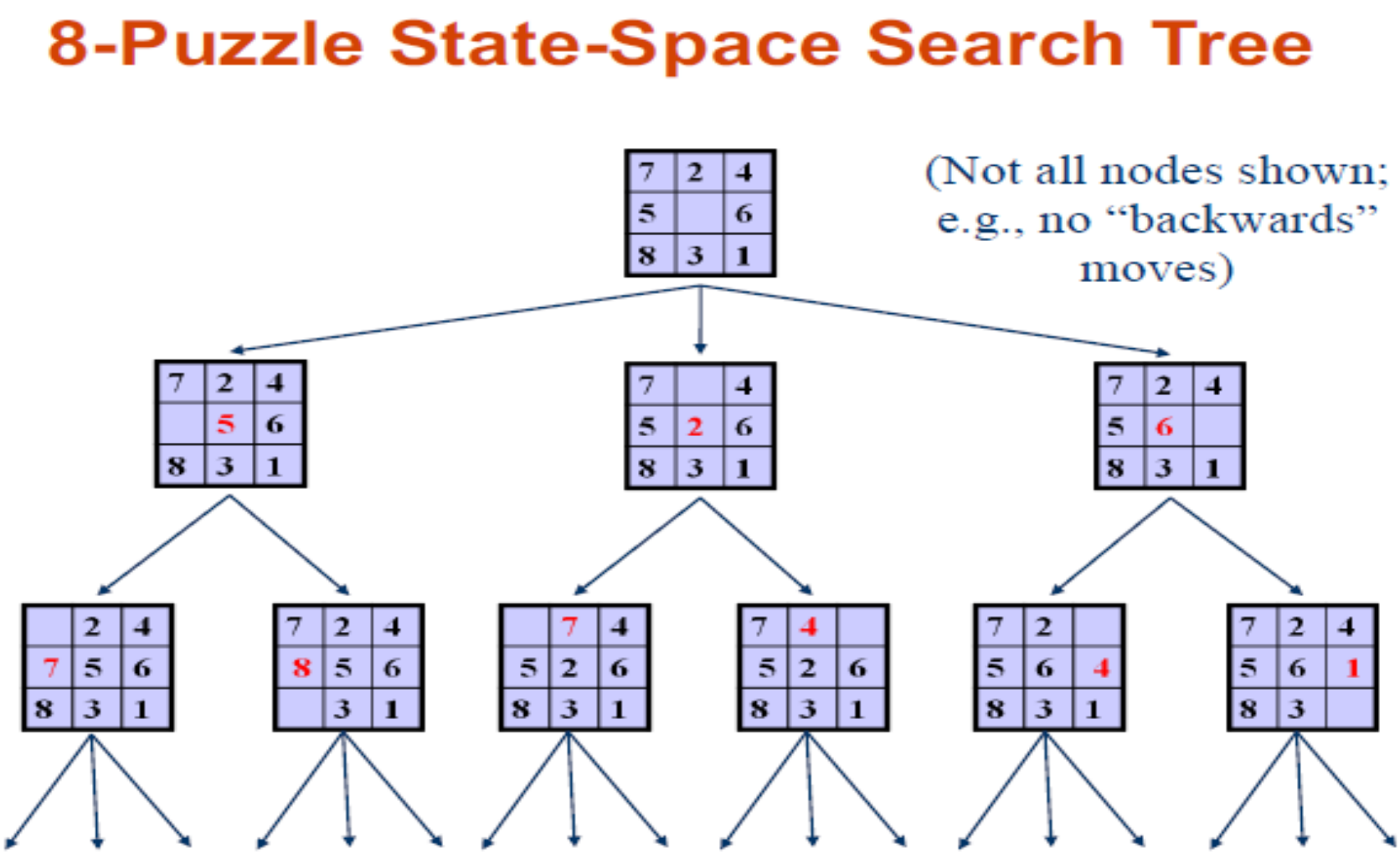
Minimum path from Start to Goal state: 3 actions

Alternative, longer plan: 4 actions

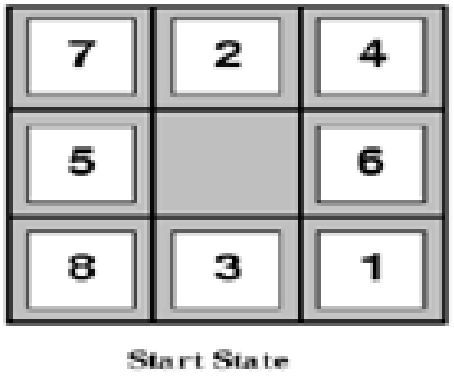
Note: path with thousands of steps before reaching goal also exist.

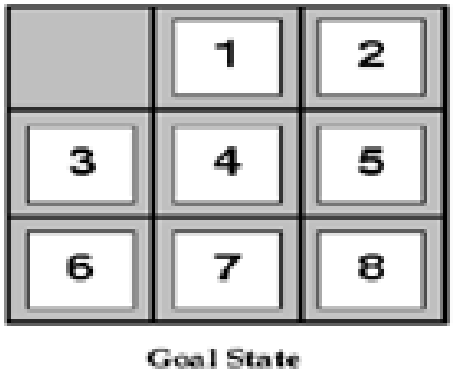
- In a vaccum world which of the following is required to reach a goal for cleaning purpose?
- A. current location and status of all rooms
- B. Move Left, Right, Suck
- C. Wait if the room is clean
- D. Both A and B

The search Process explores all State Spaces in a given scenario



Find the total number of states the 8 square puzzle grid can be in

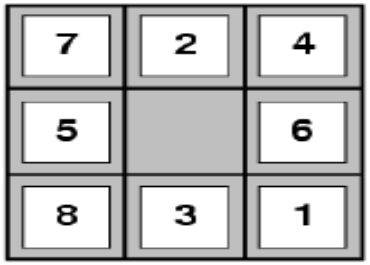




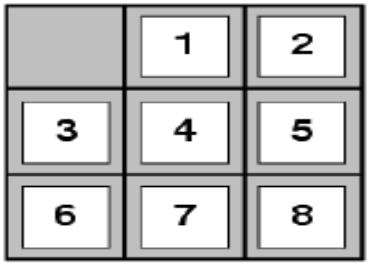
- Its 9!

- States?? List of 9 locations- e.g., [7,2,4,5,-,6,8,3,1]
- Initial state?? [7,2,4,5,-,6,8,3,1]
- Actions?? *{Left, Right, Up, Down}*
- Goal test?? Check if goal configuration is reached
- Path cost?? Number of actions to reach goal

Example: The 8-puzzle "sliding tile puzzle"



Start State



Goal State

Aside: variations on goal state. eg empty square bottom right or in middle.

states? the boards, i.e., locations of tiles
actions? move blank left, right, up, down
goal test? goal state (given; tiles in order)
path cost? 1 per move

Note: finding optimal solution of *n*-puzzle family is NP-hard!
Also, from certain states you can't reach the goal.
Total number of states 9! = 362,880 (more interesting space; not all connected... only half can reach goal state)

Water Jugs Problem

Given 4-liter and 3-liter pitchers, how do you get exactly 2 liters into the 4-liter pitcher?

State: (*x*, *y*) for # liters in 4-liter and 3-liter pitchers, respectively

- Which of the following are the empty state and goal state to reach the required goal?
- **A. Initial state: (0, 0) Goal state: (2, *)**
- **B. Initial state: (2, *) Goal state: (2, *)**
- **C. Initial state: (2, 0) Goal state: (0, 0)**
- **D. Initial state: (4, 3) Goal state: (4, 2)**

Slon to Water Jug problem

Useful Concepts

- ***State space***: **the set of all states reachable from the initial state by *any* sequence of actions**
  - *When several operators can apply to each state, this gets large very quickly*
  - *Might be a proper subset of the set of configurations*
- ***Path***: **a sequence of actions leading from one state s_j to another state s_k**
- ***Frontier:*** **those states that are available for *expanding* (for applying legal actions to)**
- ***Solution***: **a path from the initial state s_i to a state s_f that satisfies the goal test**

Some additional terms

- A **solution** to a problem is an action sequence that leads from the initial state to a goal state.
- Solution quality is measured by the path cost function,
- and an **optimal solution** has the lowest path cost among all solutions.

Some more MCQS

1. The essence of search strategy/algorithm is

- A. Explore all paths to reach a goal
- B. Choose a path breadth wise first
- C. Choose a path depth wise first
- D. Determine which frontier node to expand at a given state.

2. All the nodes that are to be expanded are ……

- A. Frontiers
- B. Leaf nodes
- C. both A and B
- D. none of above

Formalizing Search in a State Space

- **A state space is a directed *graph*: (V, E)**
  - V is a set of nodes (vertices)
  - E is a set of arcs (edges)
    each arc is *directed* from one node to another node
- **Each node is a data structure that contains:**
  - a **state** description
  - other information such as:
    - link to parent node
    - name of action that generated this node (from its parent)
    - other bookkeeping data

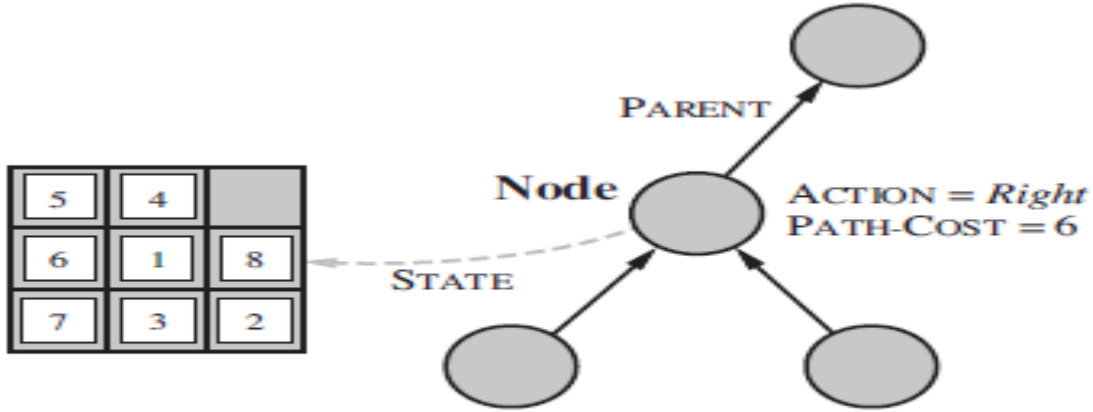


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:

- n.STATE: the state in the state space to which the node corresponds;
- n.PARENT: the node in the search tree that generated this node;
- n.ACTION: the action that was applied to the parent to generate the node;
- n.PATH-COST: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers.

Formalizing Search in a State Space

- **Each arc corresponds to one of the finite number of actions:**
  - when the action is applied to the state associated with the arc's source node
  - then the resulting state is the state associated with the arc's destination node

- **Each arc has a fixed, positive cost:**
  - corresponds to the cost of the action

Formalizing Search in a State Space

- **Each node has a finite set of successor nodes:**
  - corresponding to ***all*** the legal actions that can be applied at the source node's state

- **Expanding a node means:**
  - generate ***all*** successor nodes
  - add them and their associated arcs to the *state-space search tree*

Formalizing Search in a State Space

- **One or more nodes are designated as start nodes**
- **A goal test is applied to a node's state to determine if it is a goal node**
- **A solution is a sequence of actions associated with a path in the state space from a start to a goal node:**
  - just the goal state (e.g., cryptarithmetic)
  - a path from start to goal state (e.g., 8-puzzle)
- **The cost of a solution is the sum of the arc costs on the solution path**

Search Summary

- **Solution is an ordered sequence of primitive actions (steps)**
  $f(x) = a_1, a_2, \ldots, a_n$ where x is the input
- **Model task as a graph of all possible states and actions, and a solution as a path**
- **A state captures all the relevant information about the past**

Formalizing Search in a State Space

State-space search is the process of searching through a state space for a solution by **making *explicit* a sufficient portion of an implicit *state-space graph*, in the form of a *search tree*, to include a goal node:**

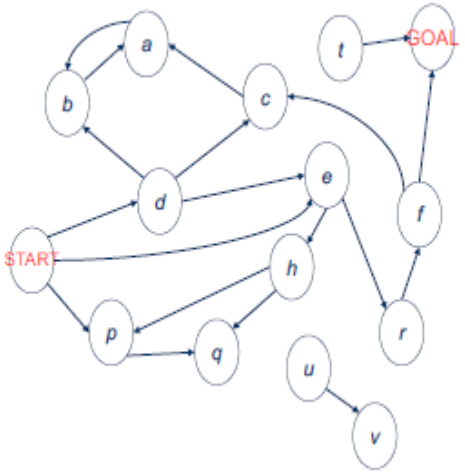
TREE SEARCH Algorithm:

Frontier = {*S*}, where *S* is the start node
Loop do
  if *Frontier* is empty **then return** failure
  pick a node, *n*, from *Frontier*
  if *n* is a goal node **then return** solution
  Generate all *n*'s successor nodes and add them all to *Frontier*
  Remove *n* from *Frontier*

called "**expanding**" node *n*

- **Each node implicitly represents**
  - a partial solution path from the start node to the given node
  - cost of the partial solution path
- **From this node there may be**
  - many possible paths that have this partial path as a prefix
  - many possible solutions

A State Space Graph



What is the corresponding search tree?

Key Issues of State-Space Search Algorithm

- Search process constructs a "**search tree**"
  - root is the start state
  - leaf nodes are:
    - unexpanded nodes (in the Frontier list)
    - "dead ends" (nodes that aren't goals and have no successors because no operators were possible)
    - goal node is last leaf node found
- Loops in graph may cause "search tree" to be infinite even if state space is small
- **Changing the *Frontier* ordering leads to different search strategies**

Basic search algorithms: *Tree Search*

Enumerate in some order all possible paths from the initial state

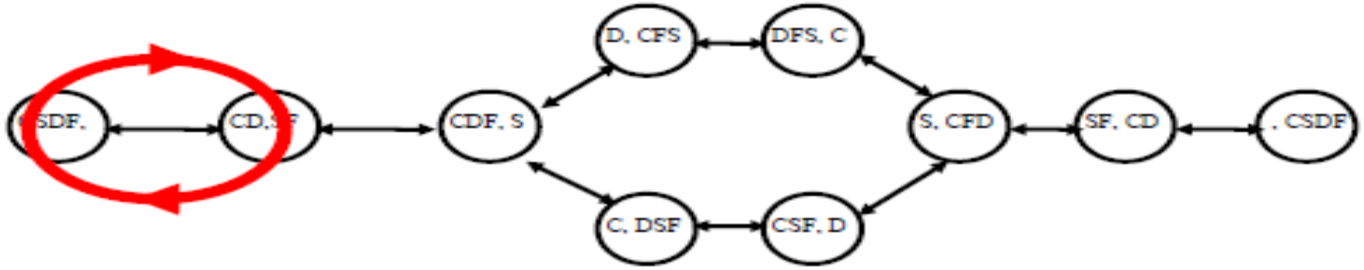
Here: search through *explicit tree generation*

- —ROOT= initial state.
- —Nodes in search tree generated through *transition model*
- n general search generates a *graph*(same state through multiple paths),
- —Tree search treats different paths to the same node as distinct

When does Repeated states occur and How to Handle them??

If State Space is *Not* a Tree

- **The problem: *repeated states***



- **Ignoring repeated states: wasteful (BFS) or impossible (DFS). Why?**
- **How to prevent these problems?**

If State Space is *Not* a Tree

- **We have to remember already-expanded states (called *Explored* (aka *Closed*) set) too**

- **When we pick a node from *Frontier***
  - Remove it from *Frontier*
  - Add it to *Explored*
  - Expand node, generating all successors
  - For each successor, *child*,
    - If *child* is in *Explored* or in *Frontier*, throw *child* away // for BFS and DFS
    - Otherwise, add it to *Frontier*

- Called **Graph-Search algorithm** in Figure 3.7 and **Uniform-Cost-Search** in Figure 3.14

The reason for Explored set in Graph Search

The need for Explored set

- *Algorithms that forget their history are doomed to repeat it.*
- The way to avoid exploring redundant paths is to remember where one has been. To do this, we use a data structure called the **explored set /closed list** which remembers every expanded node.
- **Newly generated nodes that match previously generated nodes**—ones in the explored set or the frontier—can be discarded instead of being added to the frontier.
- This new algorithm is called GRAPH-SEARCH

Graph Search vs Tree Search

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

The basic difference between graph search and tree search algorithm is in:

- A. Handling loopy path/repeated states using explored set
- B. Frontier expansion
- C. Both A and B
- D. none

Measuring problem-solving performance/How do we compare which search strategy is better among given options →Based In Following criteria

- Completeness: Is the algorithm guaranteed to find a solution when there is one?
- OPTIMALITY : Does the strategy find the optimal solution?
- TIME COMPLEXITY : How long does it take to find a solution?
- SPACE COMPLEXITY : How much memory is needed to perform the search?

Search strategies

A search strategy is defined by picking the order of node expansion.

Strategies are evaluated along the following dimensions:

- completeness: does it always find a solution if one exists?
- time complexity: 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 the state space (may be ∞)

Search Strategies

- ***Review: Strategy*** **= order of tree expansion**
  - Implemented by different queue structures (LIFO, FIFO, priority)

- **Dimensions for evaluation**
  - *Completeness-* always find the solution?
  - *Optimality* - finds a least cost solution (lowest path cost) first?
  - *Time complexity* - # of nodes generated *(worst case)*
  - ***Space complexity*** **- # of nodes in memory** *(worst case)*

- **Time/space complexity variables**
  - *b, maximum branching factor* of search tree
  - *d, depth* of the shallowest goal node
  - *m*, maximum length of any path in the state space (potentially ∞)

Introduction to *space* complexity

- **You know about:**
  - "Big O" notation
  - *Time complexity*

- ***Space complexity* is analogous to time complexity**

- **Units of space are arbitrary**
  - Doesn't matter because Big O notation ignores constant multiplicative factors
  - Space units:
    - —One Memory word
    - —Size of any fixed size data structure
      - eg Size of fixed size node in search tree

Uninformed search strategies

Uninformed (blind) search strategies use only the information available in the problem definition:

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

Key issue: type of queue used for the fringe of the search tree (collection of tree nodes that have been generated but not yet expanded)

Uninformed/Blind Search And Informed Search → The Difference

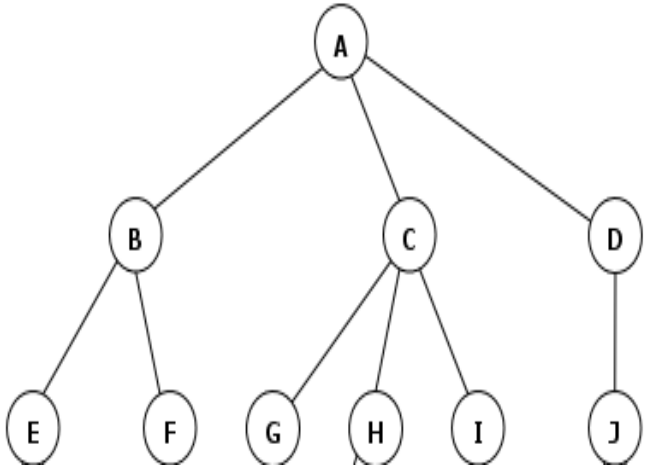
- **Uninformed means we *only* know:**
  - The goal test
  - The *successors*() function
- But ***not*** which non-goal states are better

Informally:

- ***Uninformed search:*** **All non-goal nodes in frontier look equally good**
- ***Informed search*****: Some non-goal nodes can be ranked above others.**

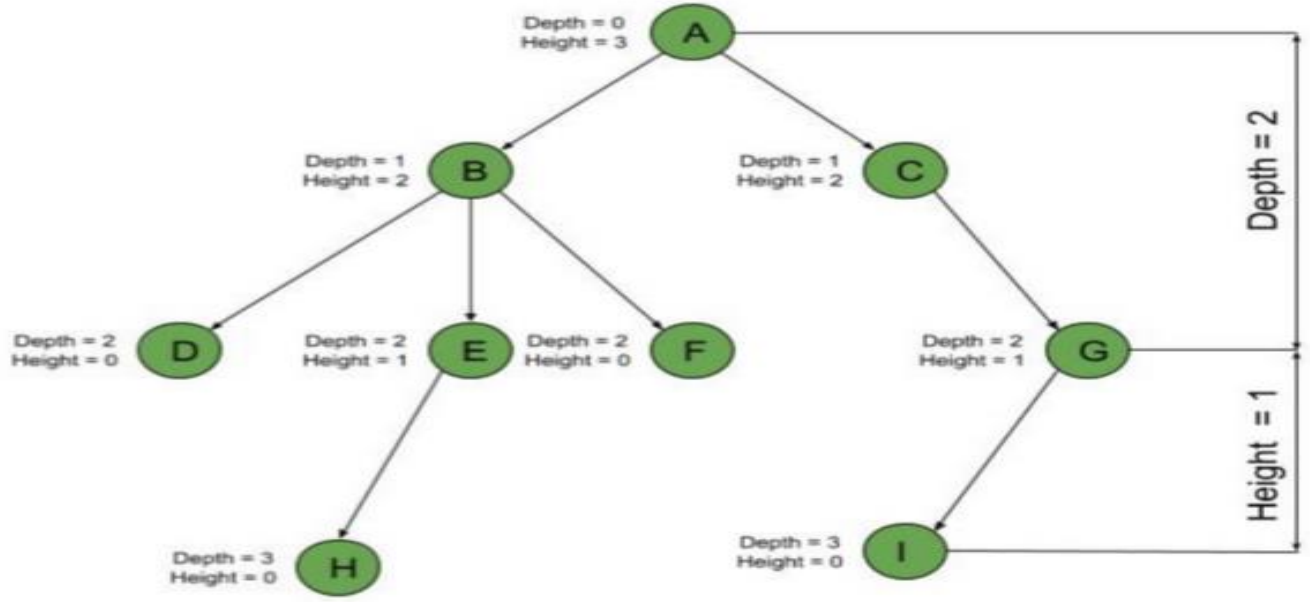
For the given search tree which of the following property represents the blind search strategy if J is a goal node and A is initial state?

A. A search strategy that visits A, B, C,D, J
B. A search strategy that visits all nodes of the tree in a particular order until J is found
C. A search strategy that visits A, D, J
D. None of above



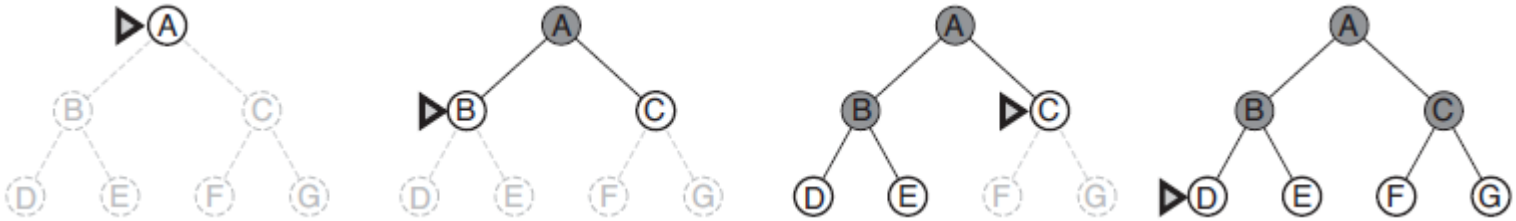
The Height and Depth Of a Tree →The Difference

- The **depth of a node is the number of edges from the node to the tree's root node.**
  A root node will have a depth of 0.
- The **height of a node is the number of edges on the *longest path* from the node to a leaf.**
  A leaf node will have a height of 0.
- The **height** of a tree would be the height of its root node,
  or equivalently, the depth of its deepest node.



Review: Breadth-first search

- **Idea:**
  - Expand *shallowest* unexpanded node
- **Implementation:**
  - *frontier* is FIFO (First-In-First-Out) Queue:
    - Put successors at the *end* of *frontier* successor list.



Properties of breadth-first search

Note: check for goal only when node is expanded.

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;
needed also to reconstruct soln. path)

Optimal soln. found?

Yes (if all step costs are identical)

Space is the bigger problem (more than time)

b: maximum branching factor of the search tree
d: depth of the least-cost solution

Breadth-first search (simplified)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node <- a node with STATE = problem.INITIAL-STATE, PATH-COST=0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier <- a FIFO queue with node as the only element
  loop do
    if EMPTY?(frontier) then return failure
    node <- POP(frontier) // chooses the shallowest node in frontier
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child <- CHILD-NODE(problem, node, action)
      if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
      frontier <- INSERT(child, frontier)
```

Position within queue of new items determines search strategy

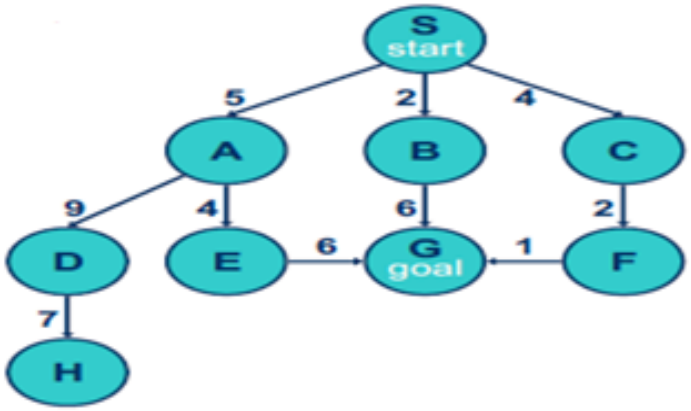
Subtle: *Node inserted into queue only after testing to see if it is a goal state*

Compare with UCS

CW → Now your turn, simulate your brain like a BFS and step wise show how it works including the frontier as well as the explored set for below problem

- Start from “S” and the goal is “G”
- Now find the soln

Breadth-First Search (BFS)

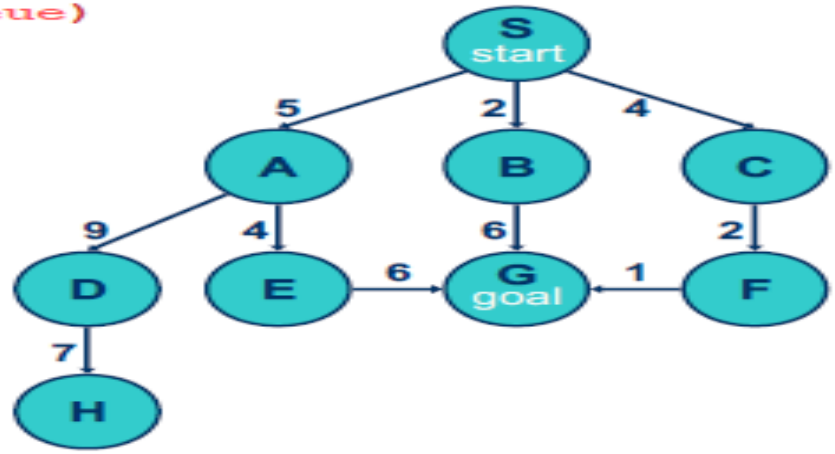


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 0, expanded: 0

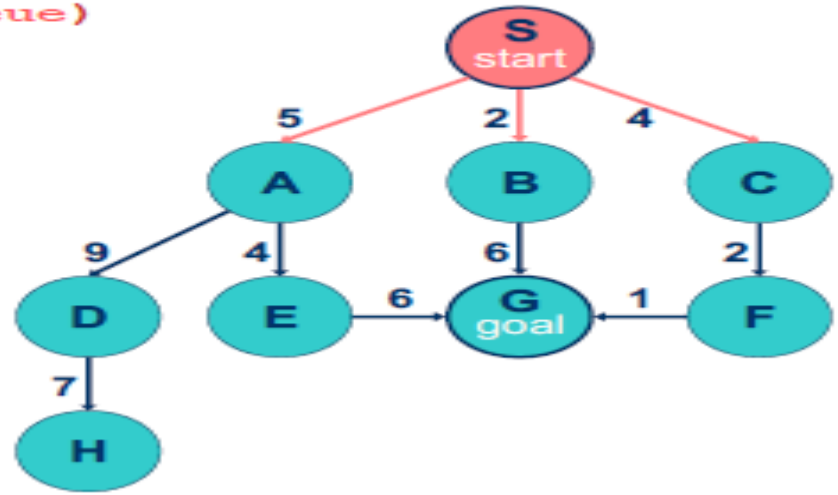
expnd. node	Frontier list
	{S}



`generalSearch(problem, queue)`

# of nodes tested: 1, expanded: 1

expnd. node	Frontier list
	{S}
S not goal	{A,B,C}

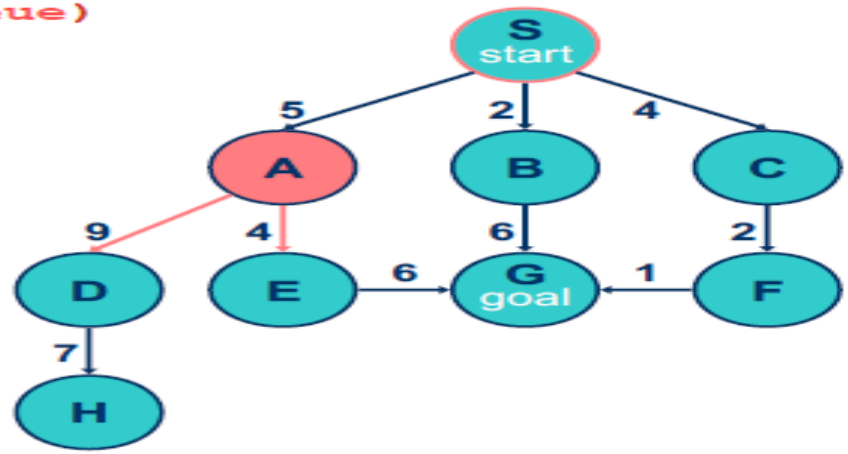


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 2, expanded: 2

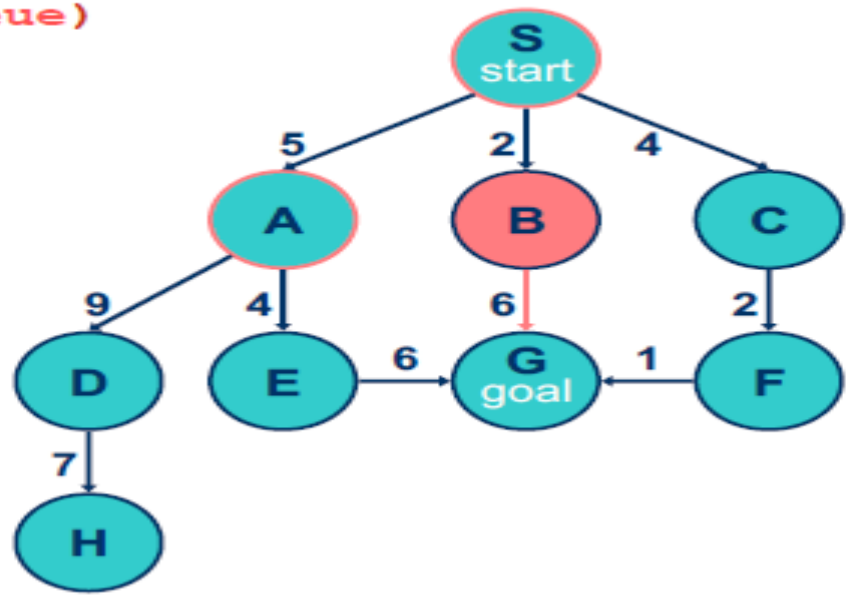
expnd. node	Frontier list
	{S}
S	{A,B,C}
A not goal	{B,C,D,E}



`generalSearch(problem, queue)`

# of nodes tested: 3, expanded: 3

expnd. node	Frontier list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B not goal	{C,D,E,G}

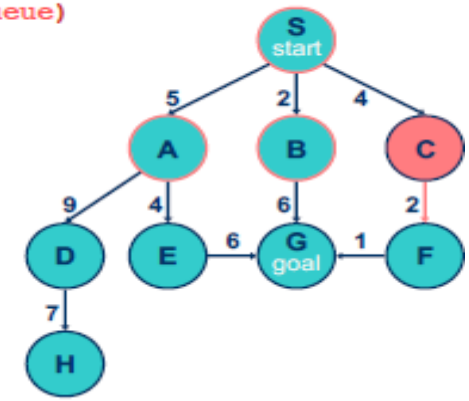


BFS

generalSearch(problem, queue)

# of nodes tested: 4, expanded: 4

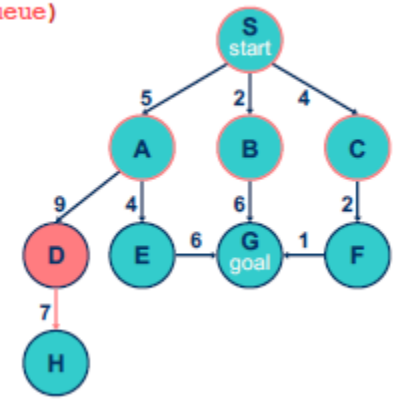
expnd. node	Frontier list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C not goal	{D,E,G,F}



generalSearch(problem, queue)

# of nodes tested: 5, expanded: 5

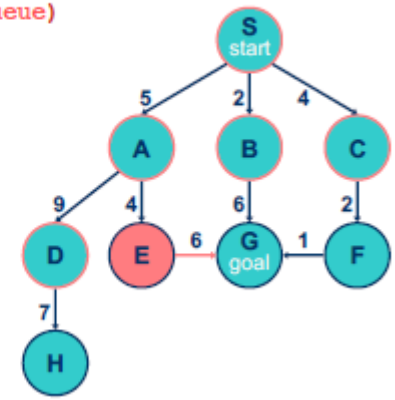
expnd. node	Frontier list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D not goal	{E,G,F,H}



generalSearch(problem, queue)

# of nodes tested: 6, expanded: 6

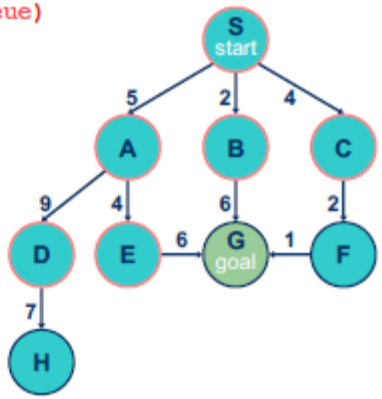
expnd. node	Frontier list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E not goal	{G,F,H,G}



generalSearch(problem, queue)

# of nodes tested: 7, expanded: 6

expnd. node	Frontier list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G goal	{F,H,G} no expand

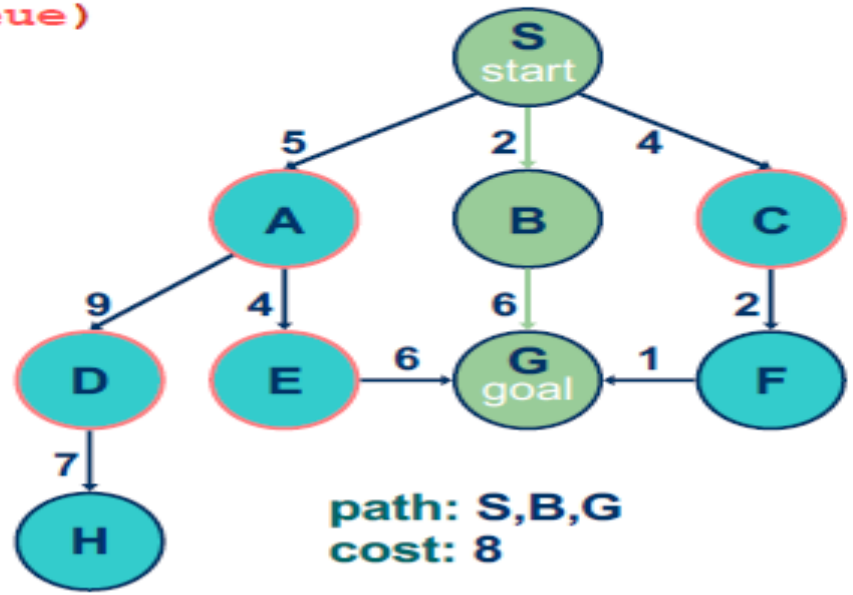


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 7, expanded: 6

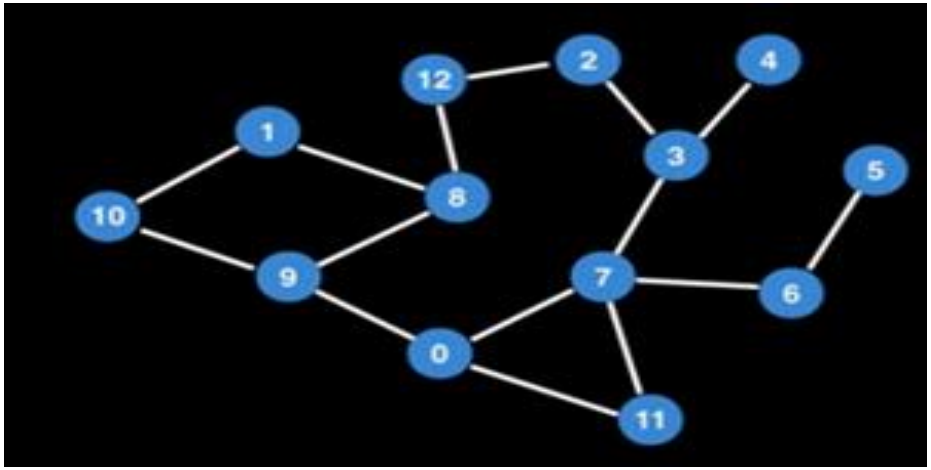
expnd. node	Frontier list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G	{F,H,G}

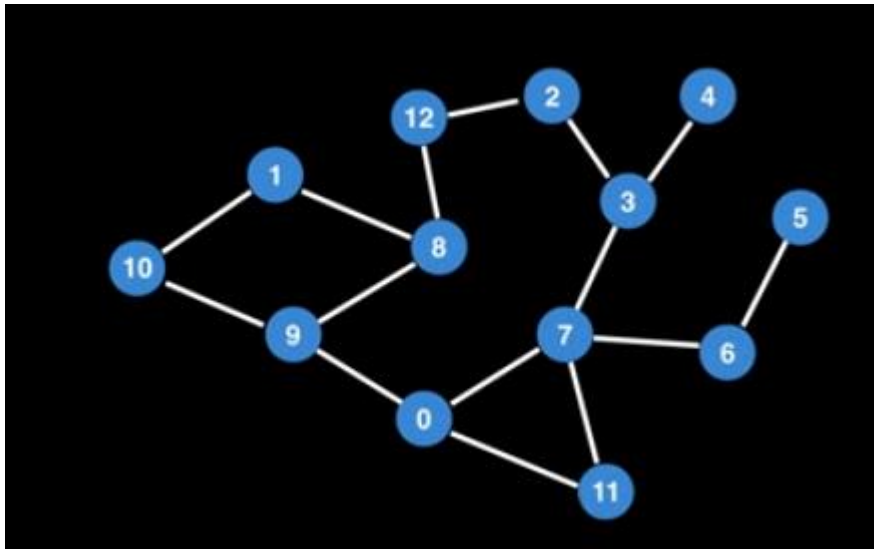


Review: Breadth-first search

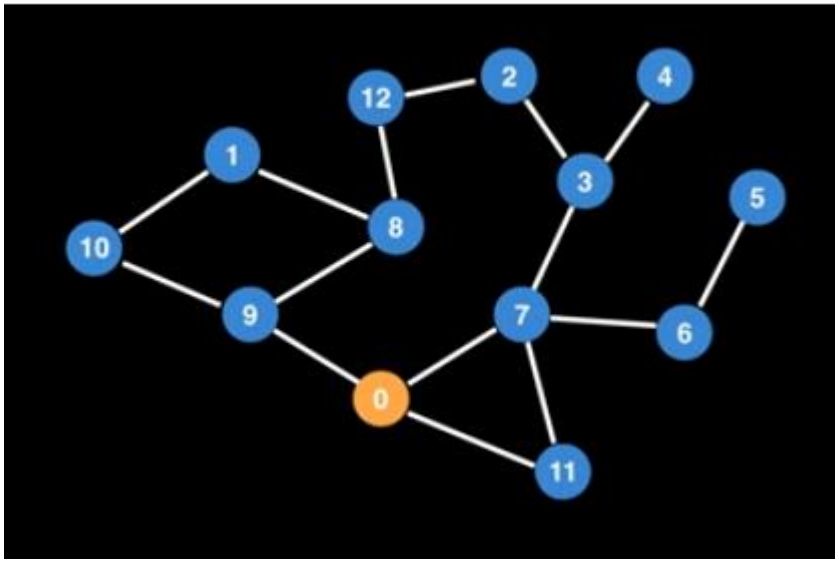
- **Idea:**
  - Expand *shallowest* unexpanded node
- **Implementation:**
  - *frontier* is FIFO (First-In-First-Out) Queue:
    —Put successors at the *end* of *frontier* successor list.

Implement a BFS for below graph traversal. Start with 0

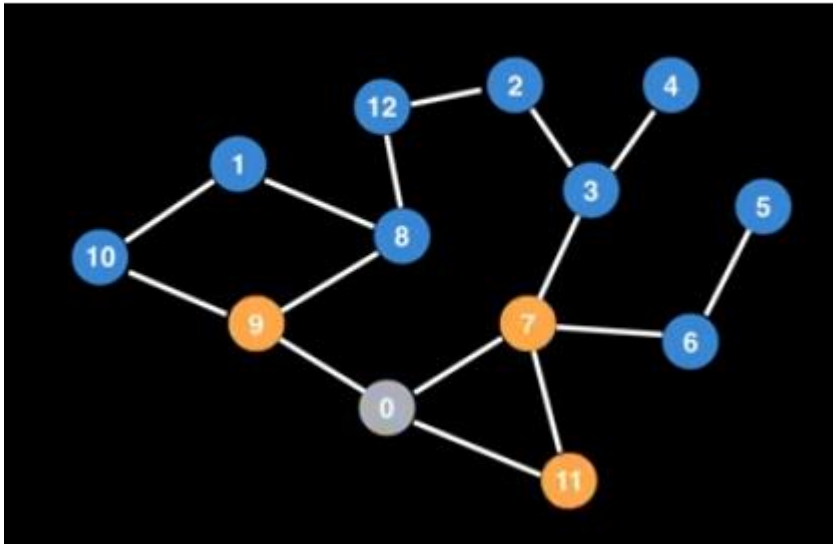



12
2
4
1
3
5
10
8
9
7
6
0
11

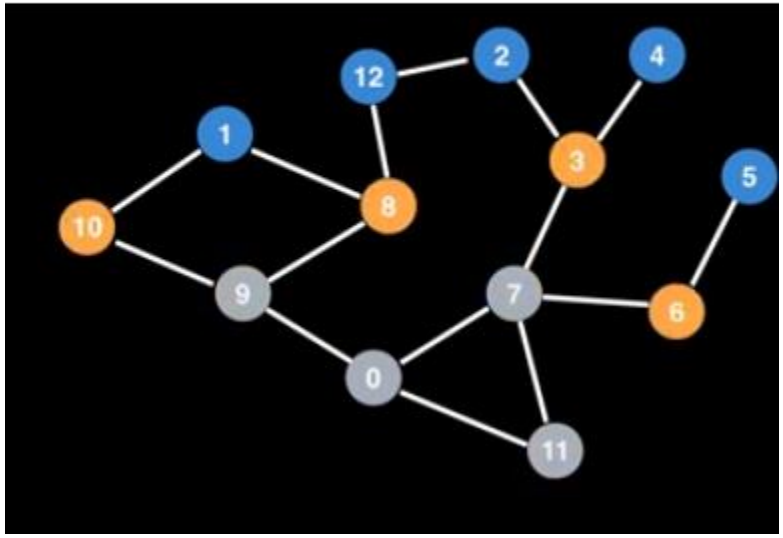
1


12
2
4
1
3
5
10
8
9
7
6
0
11

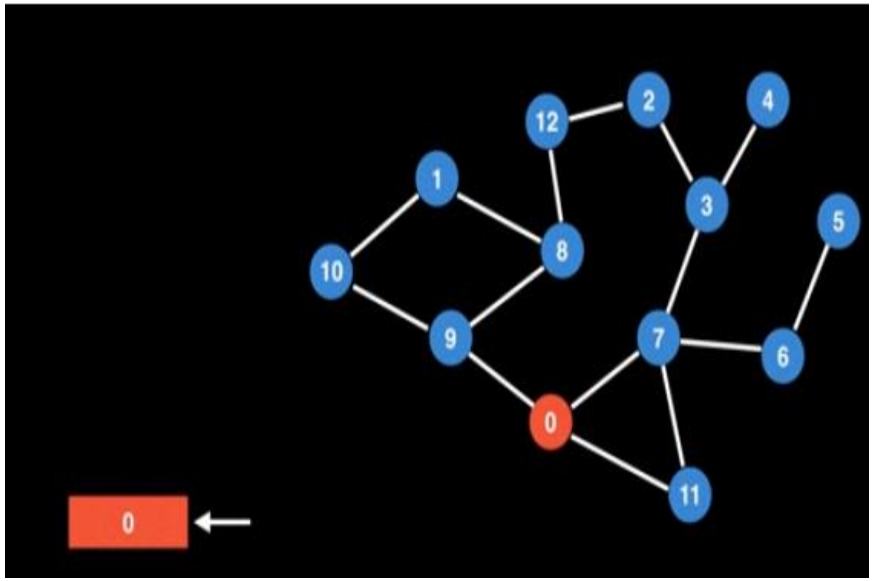
2


12
2
4
1
3
5
10
8
9
7
6
0
11

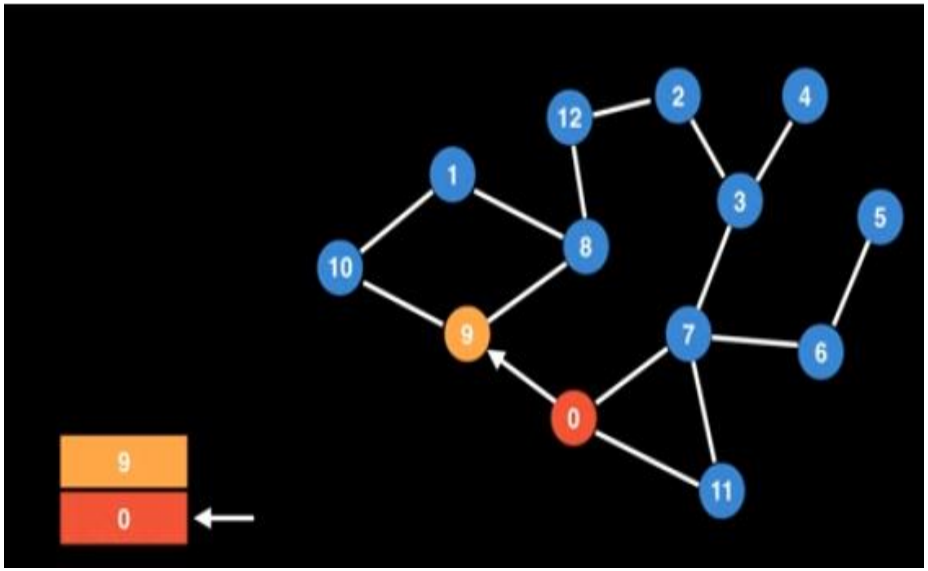
3

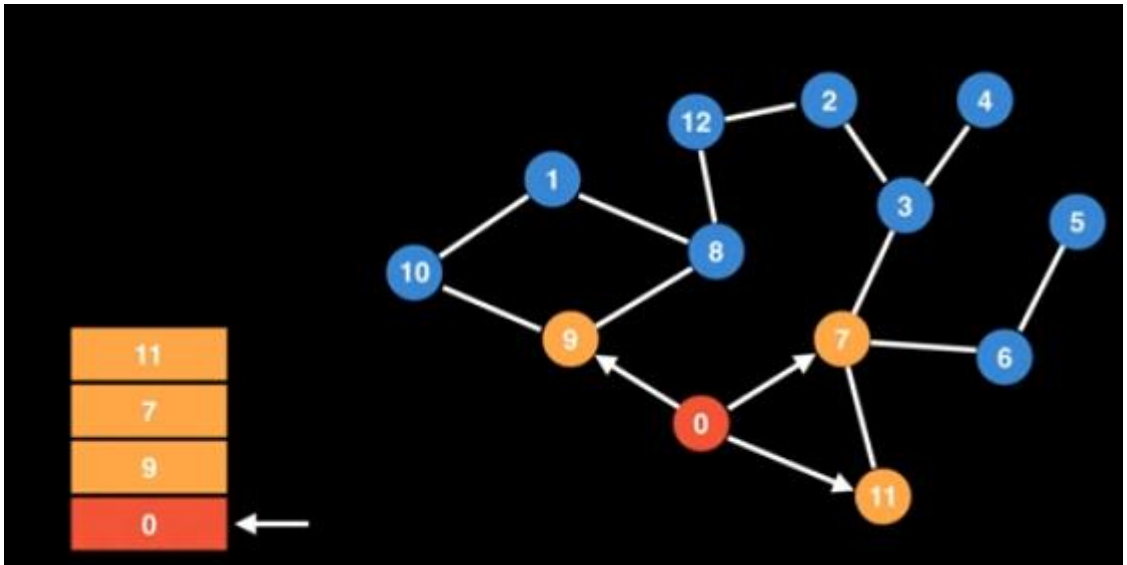

12
2
4
1
3
5
10
8
9
7
6
0
11

1


12
2
4
1
3
5
10
8
9
7
6
0
11
0

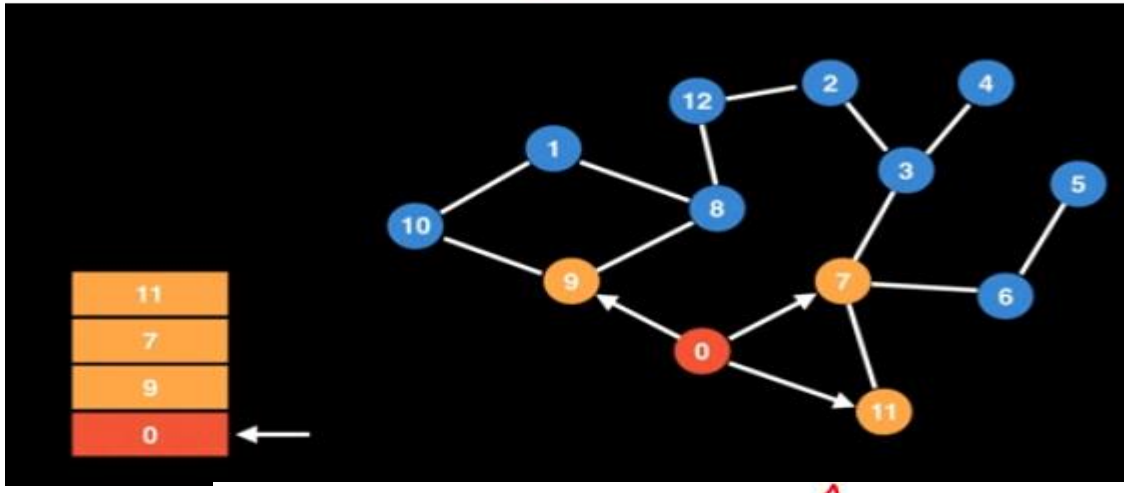
2


12
2
4
1
3
5
10
8
9
7
6
0
11
9
0

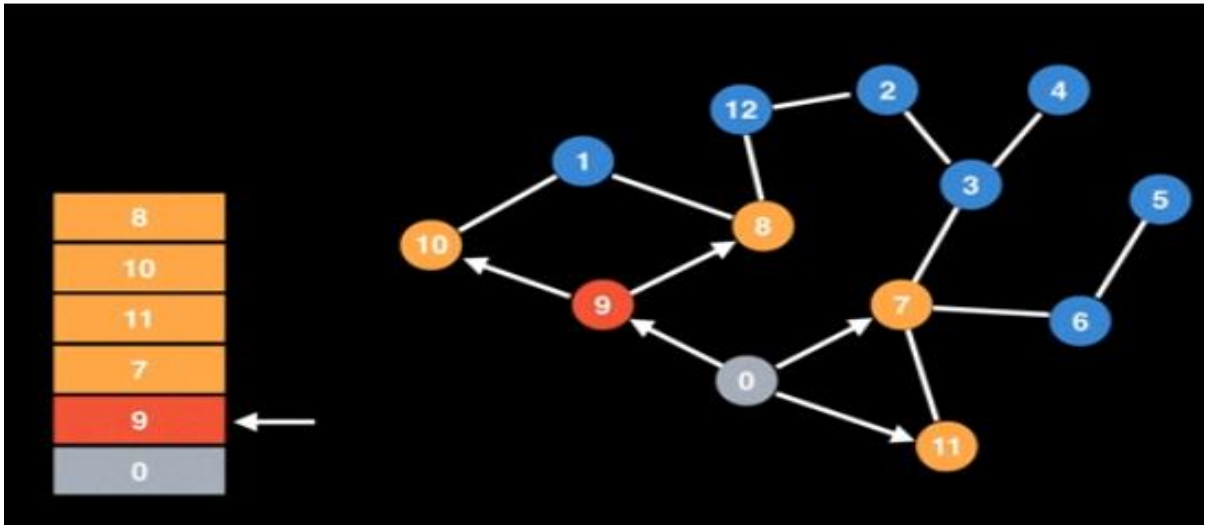

12
2
4
1
3
5
10
8
9
7
6
0
11
11
7
9
0

0 has no more unvisited neighbors
so explore 9 now in the queue

3

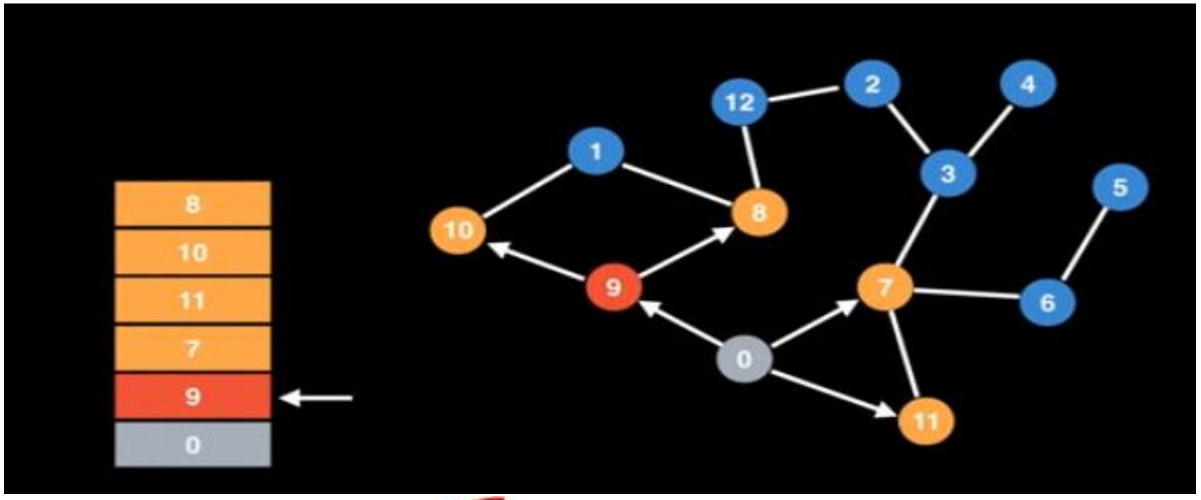


4

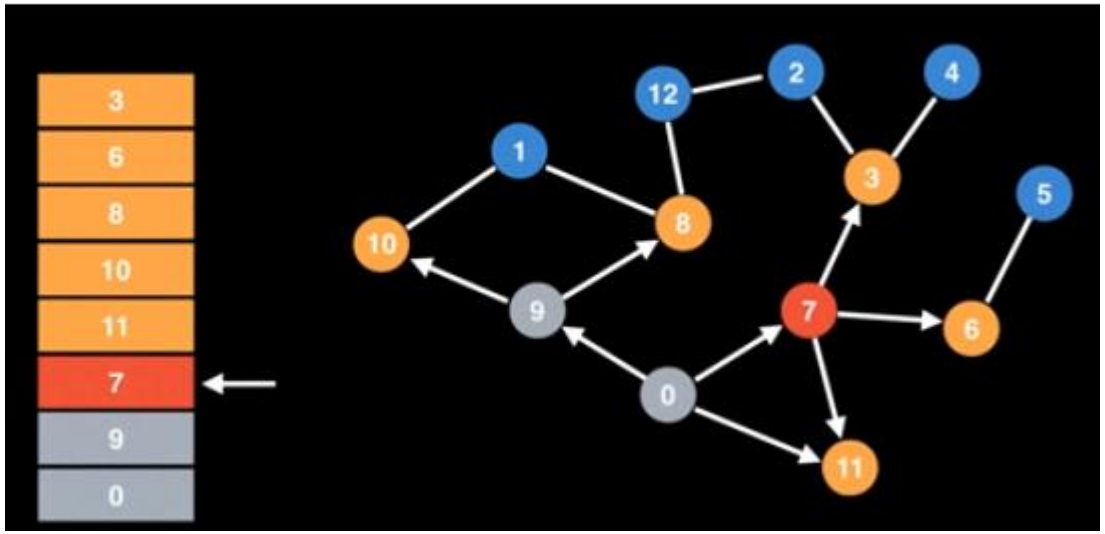


No more neighbors of 9 to visit so expand 7

5

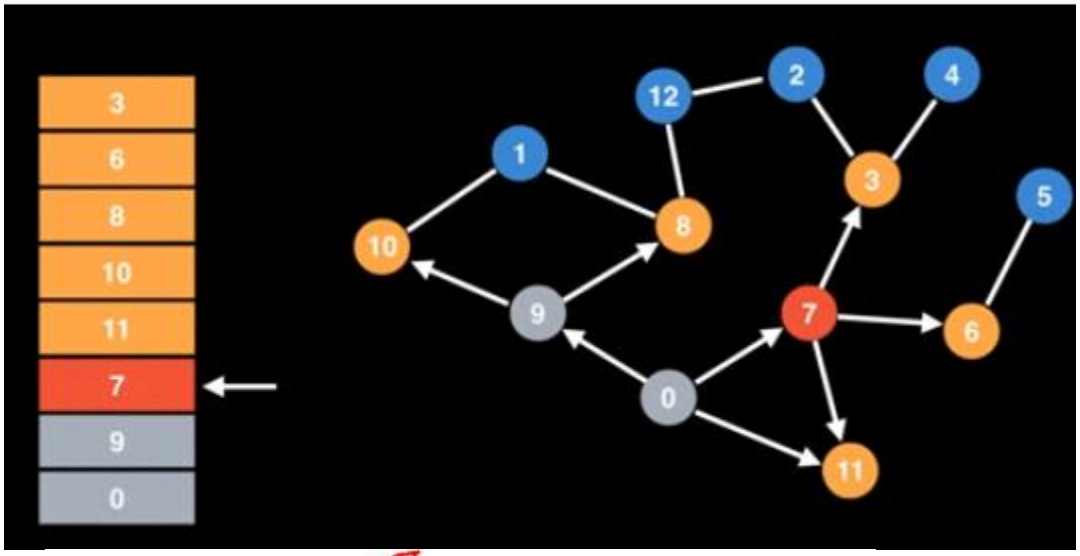


6

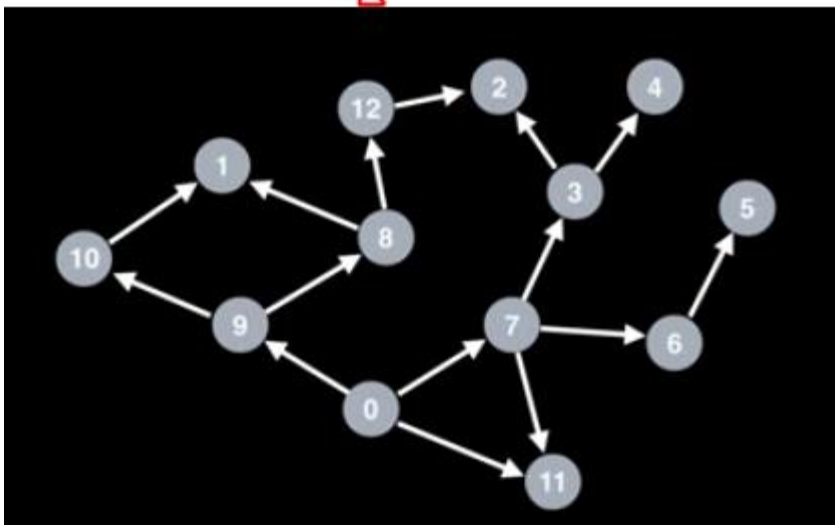


Note 11 is not again inserted to queue since 11 is already in the queue / the frontier set that's where the job of seeing the element in either the explored set or frotnier comes. So just add 6 and 3 in the queue /frontier

7

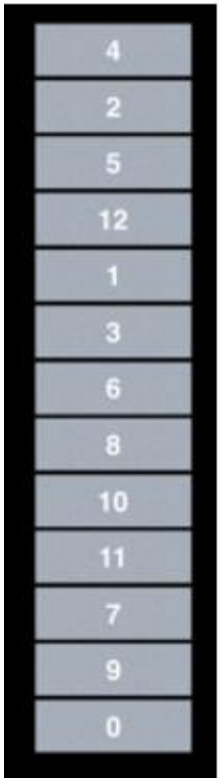


8



The final states once all nodes are visited

8



Optimality Of BFS

- When all step costs are equal, breadth-first search is optimal because it always expands the *shallowest* unexpanded node.

Properties of breadth-first search

- **Complete?** Yes (if b is finite)
- **Time Complexity?** $1+b+b^2+b^3+\ldots+b^d = O(b^d)$
- **Space Complexity?** $O(b^d)$ (keeps every node in memory)
- **Optimal?** Yes, if cost = 1 per step (not optimal in general)

b: maximum branching factor of search tree
d: depth of the least cost solution
m: maximum depth of the state space (∞)

Exponential Space (and time) Not Good...

- Exponential complexity uninformed search problems *cannot* be solved for any but the smallest instances.
- (*Memory* requirements are a bigger problem than *execution* time.)

DEPTH	NODES	TIME	MEMORY
2	110	0.11 milliseconds	107 kilobytes
4	11110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabytes
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabytes
14	10^{14}	3.5 years	99 petabytles

Fig 3.13 Assumes b=10, 1M nodes/sec, 1000 bytes/node

!! note for BFS below

So far, the news about breadth-first search has been good. The news about time and space is not so good. Imagine searching a uniform tree where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of *these* generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is

$$b + b^2 + b^3 + \cdots + b^d = O(b^d) .$$

(If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be $O(b^{d+1})$.)

As for space complexity: for any kind of graph search, which stores every expanded node in the *explored* set, the space complexity is always within a factor of b of the time complexity. For breadth-first graph search in particular, every node generated remains in memory. There will be $O(b^{d-1})$ nodes in the *explored* set and $O(b^d)$ nodes in the frontier,

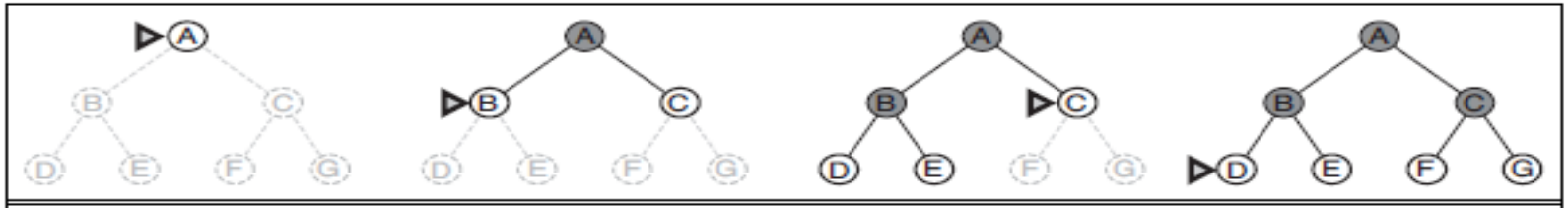
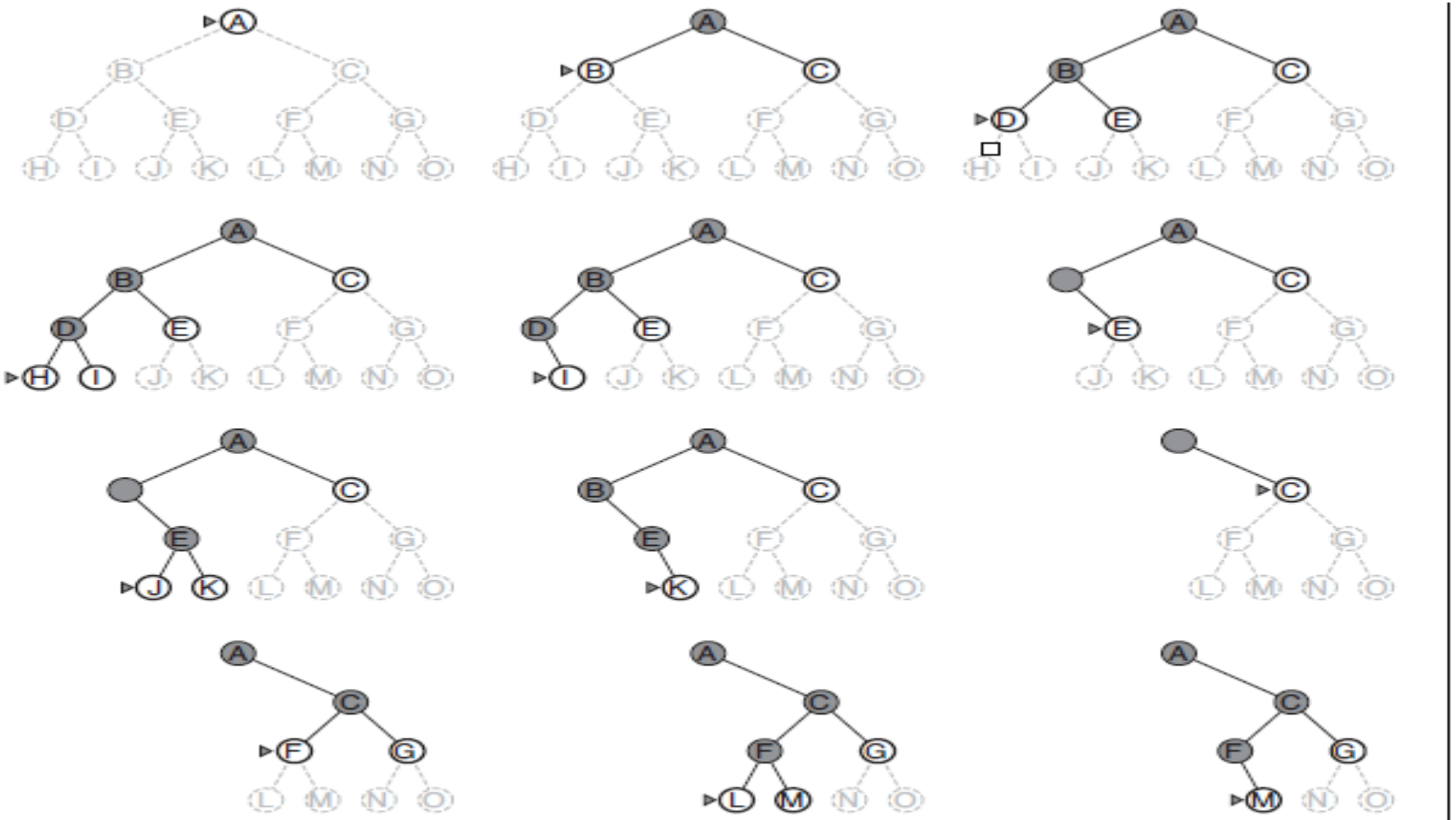


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Depth First Search

- Always expands one of the nodes at the deepest level of the tree
  - Low memory requirements
  - Problem: depth could be infinite
- Uses a stack (LIFO)

The DFS

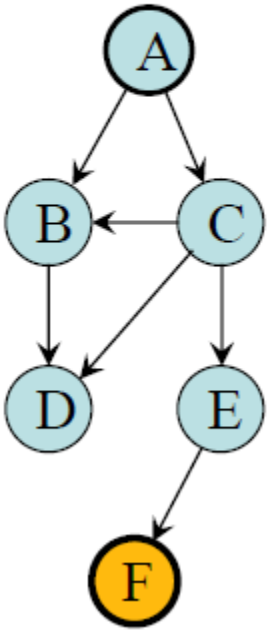


Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

DFS Graph search example

1

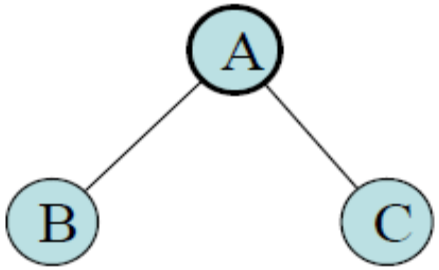
State space graph



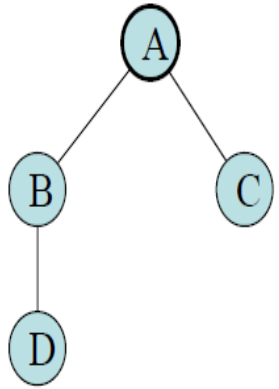
2

Search tree

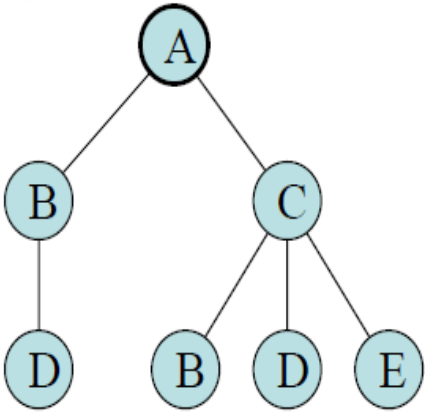
3 Search tree



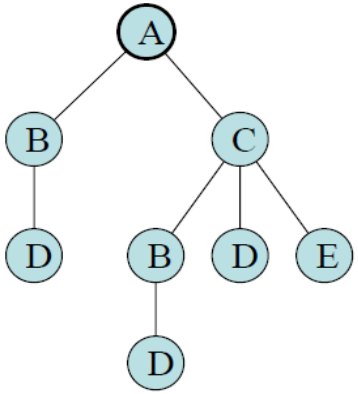
4 Search tree



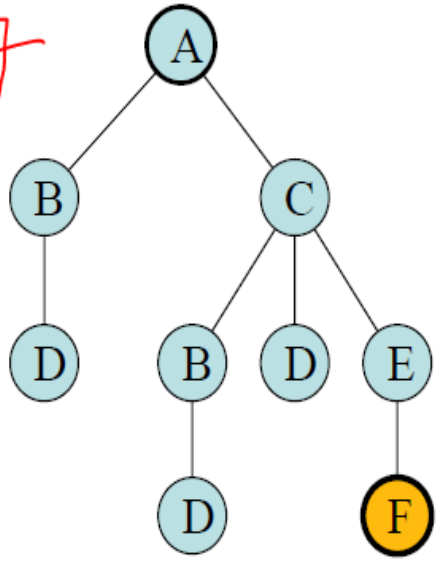
5 Search tree



6 Search tree



7 Search tree



How can we get the best of both?

Search Conundrum

- **Breadth-first**
  - ☑ Complete,
  - ☑ Optimal
  - ☒ *but* uses $O(b^d)$ space
- **Depth-first**
  - ☒ Not complete *unless m is bounded*
  - ☒ Not optimal
  - ☒ Uses $O(b^m)$ time; terrible if m >> d
  - ☑ *but* only uses ***O(b*m) space***

Depth-limited search: A building block

- **Depth-First search *but with depth limit l.***
  - i.e. nodes at depth l *have no successors.*
  - No infinite-path problem!

- **If $l = d$ (by luck!), then optimal**
  - But:
    - **If $l < d$ then incomplete** ☹
    - **If $l > d$ then not optimal** ☹

- **Time complexity:** $O(b^l)$
- **Space complexity:** $O(bl)$ ☺

Iterative deepening search

- A general strategy to find best depth limit ℓ.
  - Key idea: use Depth-limited search as subroutine, with increasing ℓ.

```
For d = 0 to ∞ do
        depth-limited-search to level d
        if it succeeds
                then return solution
```

  - *Complete & optimal*: Goal is always found at depth *d*, the depth of the shallowest goal-node.

Could this possibly be efficient?

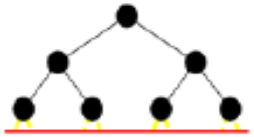
IDS(Iterative Deepening Search)

Nodes constructed at each deepening

- Depth 0: 0 (Given the node, doesn't *construct* it.)

- Depth 1: b^1 nodes

- Depth 2: b nodes + b^2 nodes



- Depth 3: b nodes + b^2 nodes + b^3 nodes
- …

IDS(Iterative Deepening Search)

Total nodes constructed:

- Depth 0: 0 (Given the node, doesn't *construct* it.)
- Depth 1: $b^1 = b$ nodes
- Depth 2: b nodes + b^2 nodes
- Depth 3: b nodes + b^2 nodes + b^3 nodes
- …

Suppose the first solution is the last node at depth 3:

Total nodes constructed:

*3**b nodes + *2**b^2 nodes + *1**b^3 nodes

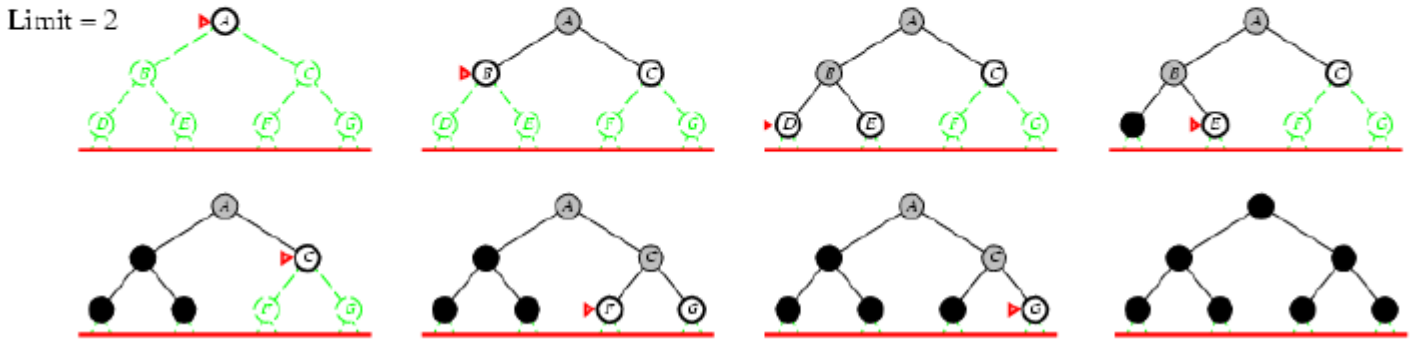
Iterative deepening search l =0

Limit = 0

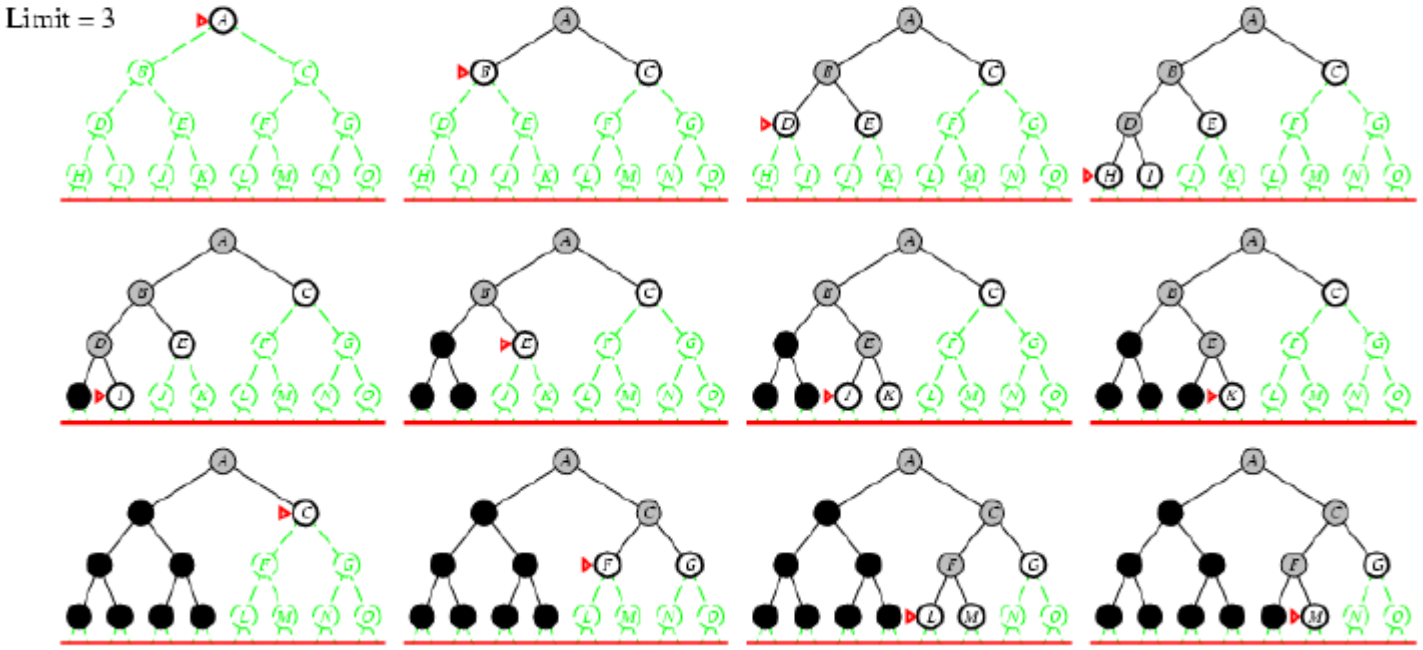
Iterative deepening search l =1

Limit = 1

Iterative deepening search $l = 2$



Iterative deepening search $l = 3$



ID search, Evaluation

- **Complete: YES (no infinite paths)** ☺
- **Time complexity:** $O(b^d)$
- **Space complexity:** $O(bd)$ ☺
- **Optimal: YES if step cost is 1.** ☺

Summary of algorithms

Criterion	Breadth-First	Depth-First	Depth-limited	Iterative deepening
Complete?	**YES**	**NO**	**NO**	**YES**
Time	b^d	b^m	b^l	b^d
Space	b^d	bm	bl	bd
Optimal?	**YES**	**NO**	**NO**	**YES**

Which search is implemented with an empty first-in-first-out queue?
a) Depth-first search
b) Breadth-first search
c) Bidirectional search
d) None of the mentioned

When is breadth-first search is optimal?
a) When there is less number of nodes
b) When all step costs are equal
c) When all step costs are unequal
d) None of the mentioned

Select the most appropriate situation from below options where a blind search can be used.

- Real-life situation
- Complex game
- Small Search Space
- All of the above

- What is the space complexity of Depth-first search? Where m is the max depth of search tree

a) $O(b^m)$
b) O(bl)
c) O(m)
d) O(bm)

How many parts does a problem consists of?
a) 1 b) 2 c) 3 d) 4

Which algorithm is used to solve any kind of problem?
a) Breadth-first algorithm
b) Tree algorithm
c) Bidirectional search algorithm
d) None of the mentioned

Which search implements stack operation for searching the states?
a) Depth-limited search
b) Depth-first search
c) Breadth-first search
d) None of the mentioned

The time and space complexity for IDS is

a) $O(b^d)$ and $O(bd)$

b) $O(bd)$ and $O(b^d)$

c) $O(bm)$ and $O(b^m)$

d) none

Depth first search expands the …… node in the current fringe of the search

a) Child

b) Shallowest

c) Lowest path cost

d) None

DFS is ……efficient and BFS is ……. Efficient

a) Space, Space

b) Time, Time

c) Time, Space

d) Space , Time

uniform-cost search (UCS)

- When all step costs are equal, breadth-first search is optimal because it always expands the *shallowest* unexpanded node.
- By a simple extension, we can find an algorithm that is optimal with any step-cost function.
- Instead of expanding the shallowest node, **UCS** expands the node n with the *lowest path cost* g(n).
- This is done by storing the frontier as a
- priority queue ordered by g.

Uniform-cost search (UCS)

- **Extension of BF-search:**
  - **Expand node with *lowest path cost***
- **Implementation:**
  ***frontier* = priority queue ordered by g(n)**
- **Subtle but significant difference from BFS:**
  - Tests if a node is a goal state when it is selected for expansion, not when it is added to the frontier.
  - Updates a node on the frontier if a better path to the same state is found.
  - So always enqueues a node *before checking whether it is a goal.*

Uniform Cost Search(UCS) algorithm → below algorithm handles repeated states using explored set and the figure represents Part of the Romania state space, selected to illustrate uniform-cost search

```
function Uniform-Cost-Search (problem)
loop do
  if Empty?(frontier) then return failure
  node = Pop(frontier)
  if Goal?(node) then return Solution(node)
  Insert node in explored
  foreach child of node do
    if child not in frontier or explored then
      Insert child in frontier
    else if child in frontier with higher cost then
      Remove that old node from frontier
      Insert child in frontier
```

This is the algorithm in Figure 3.14 in the textbook; note that if *child* is **not** in *frontier* but **is** in *explored*, this algorithm will throw away *child*

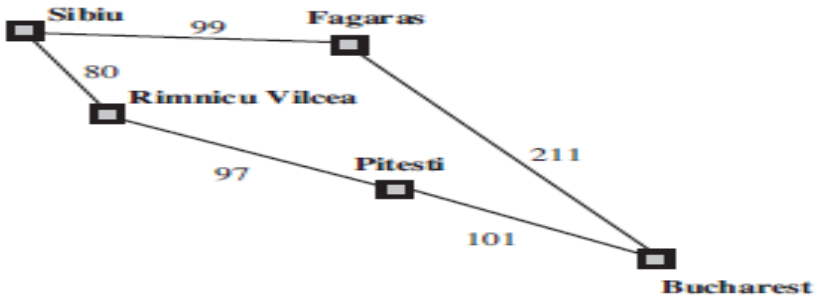


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

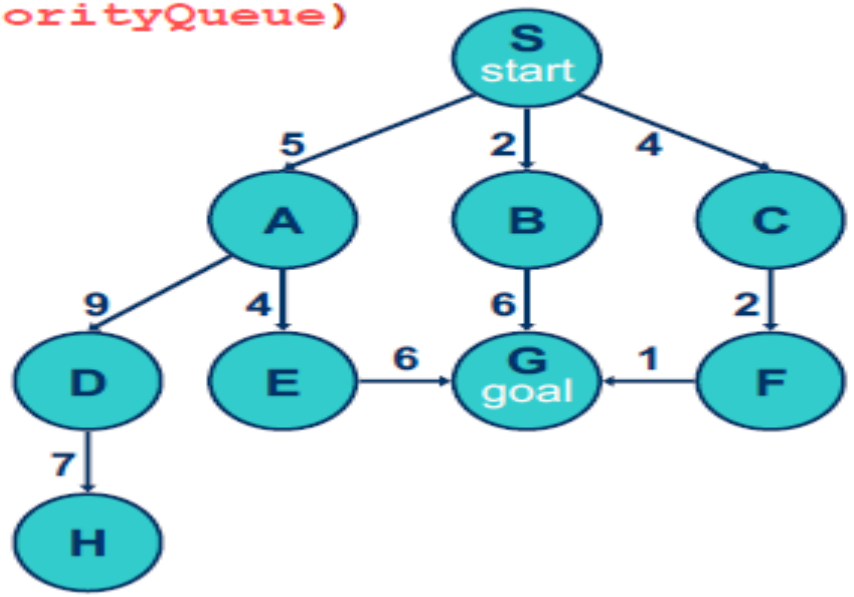
Uniform-Cost Search (UCS)

- Use a **Priority Queue** to order nodes in *Frontier*, sorted by path cost
- Let $g(n)$ = cost of path from start node s to current node n
- Sort nodes by increasing value of g

`generalSearch(problem, priorityQueue)`

# of nodes tested: 0, expanded: 0

expnd. node	Frontier list
	{S}

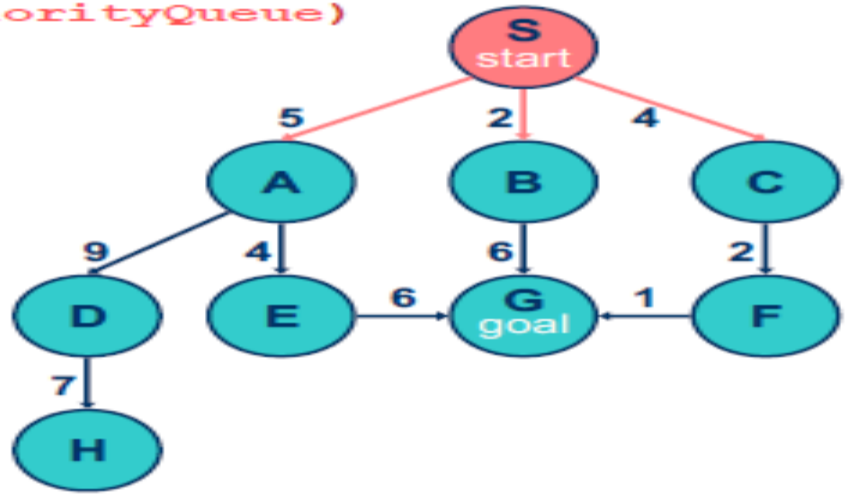


Uniform-Cost Search (UCS)

`generalSearch(problem, priorityQueue)`

# of nodes tested: 1, expanded: 1

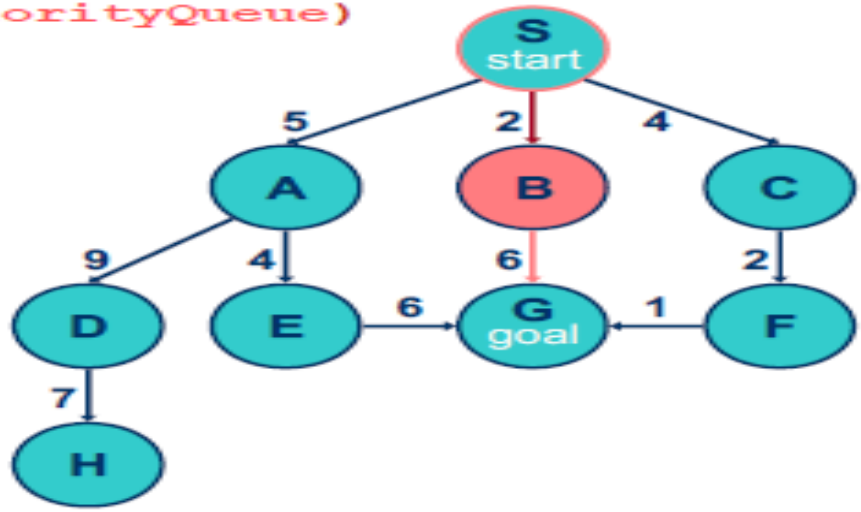
expnd. node	Frontier list
	{S:0}
S not goal	{B:2,C:4,A:5}



`generalSearch(problem, priorityQueue)`

# of nodes tested: 2, expanded: 2

expnd. node	Frontier list
	{S}
S	{B:2,C:4,A:5}
B not goal	{C:4,A:5,G:2+6}

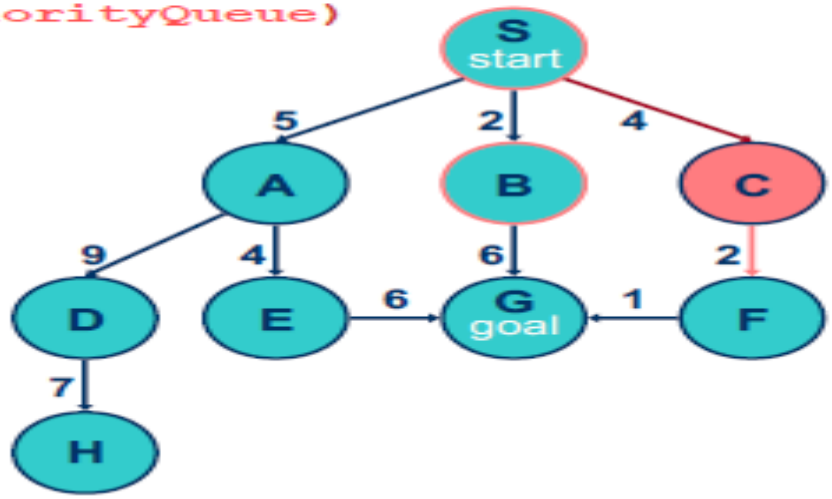


Uniform-Cost Search (UCS)

`generalSearch(problem, priorityQueue)`

# of nodes tested: 3, expanded: 3

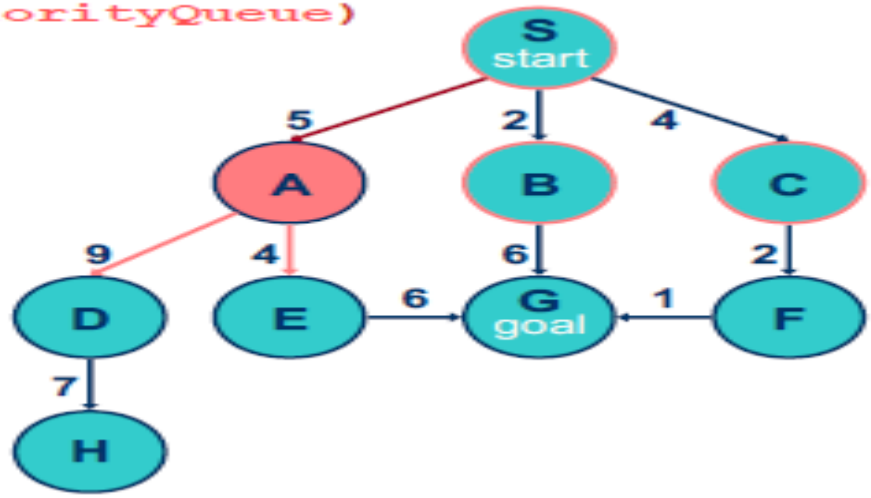
expnd. node	Frontier list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C not goal	{A:5,F:4+2,G:8}



`generalSearch(problem, priorityQueue)`

# of nodes tested: 4, expanded: 4

expnd. node	Frontier list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A not goal	{F:6,G:8,E:5+4, D:5+9}

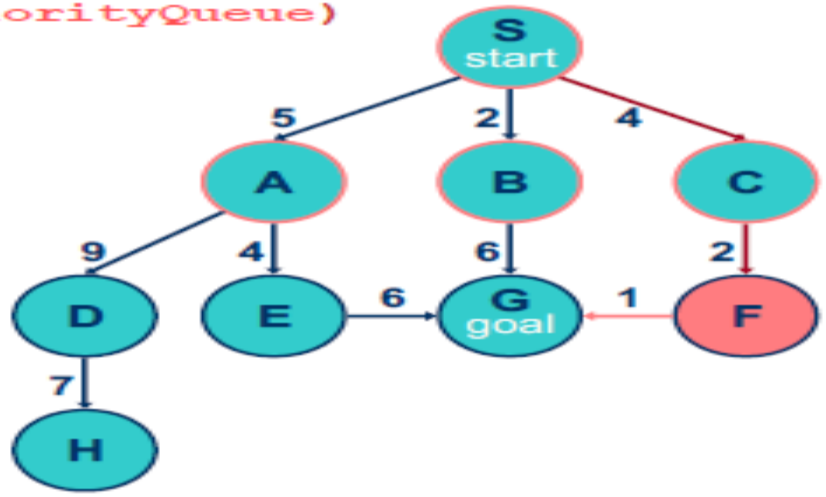


Uniform-Cost Search (UCS)

`generalSearch(problem, priorityQueue)`

# of nodes tested: 5, expanded: 5

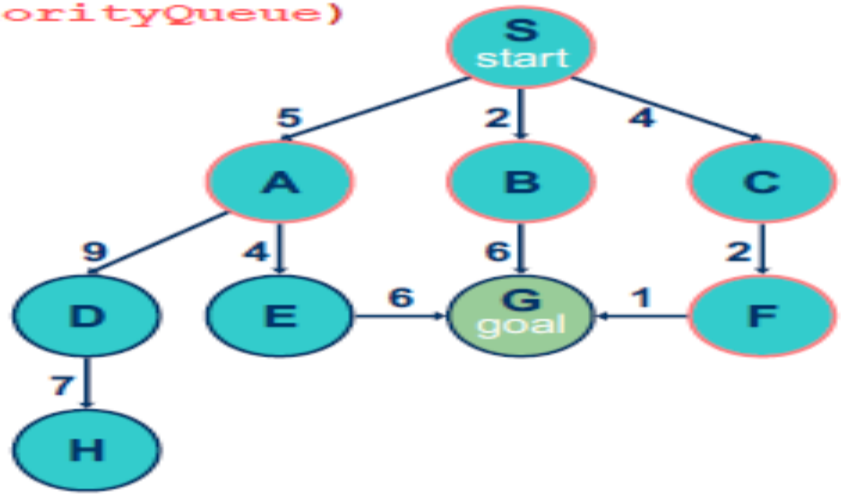
expnd. node	Frontier list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F not goal	{G:4+2+1,G:8,E:9, D:14}



`generalSearch(problem, priorityQueue)`

# of nodes tested: 6, expanded: 5

expnd. node	Frontier list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G goal	{G:8,E:9,D:14} no expand

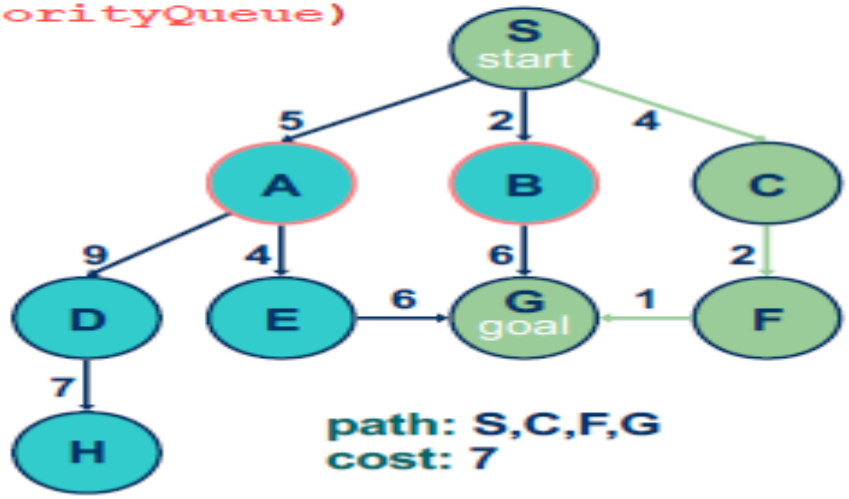


Uniform-Cost Search (UCS)

`generalSearch(problem, priorityQueue)`

# of nodes tested: 6, expanded: 5

expnd. node	Frontier list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G	{G:8,E:9,D:14}



- **Time and space complexity:** $O(b^d)$ **(i.e., exponential)**
  - d is the depth of the solution
  - b is the branching factor at each non-leaf node

- **More precisely, time and space complexity is** $O(b^{C^*/\varepsilon})$ **where all edge costs are** ε, $\varepsilon > 0$, **and** C^* **is the best goal path cost**

!! note for UCS below

- *uniform-cost search expands nodes in order of their optimal path cost.* Hence, **the first goal node selected for expansion must be the optimal solution.**

Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d. Instead, let C^* be the cost of the optimal solution,[7] and assume that every action costs at least ϵ. Then the algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, which can be much greater than b^d. This is because uniform-cost search can explore large trees of small steps before exploring paths involving large and perhaps useful steps. When all step costs are equal, $b^{1+\lfloor C^*/\epsilon \rfloor}$ is just b^{d+1}. When all step costs are the same, uniform-cost search is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost; thus uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily.

Complexity of UCS

- **Complete!**
- **Optimal!**
  - if the cost of each step exceeds some positive bound ε.
- **Time complexity:** $O(b^{1 + C^*/\varepsilon})$
- **Space complexity:** $O(b^{1 + C^*/\varepsilon})$

  where C is the cost of the optimal solution*

  (if all step costs are equal, this becomes $O(b^{d+1})$

NOTE: Dijkstra's algorithm just UCS without goal

Uniform-cost search

Expand least-cost (of path to) unexpanded node

(e.g. useful for finding shortest path on map)

Implementation:

– ***fringe*** **= queue** ***ordered by path cost***

g – cost of reaching a node

Complete? Yes, if step cost $\geq \varepsilon$ **(>0)**

Time? # of nodes with $g \leq$ cost of optimal solution (C^*), $O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$

Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$

Optimal? Yes – nodes expanded in increasing order of $g(n)$

Note: Some subtleties (e.g. checking for goal state).

See p 84 R&N. Also, next slide.

Uniform-cost search

Two subtleties: (bottom p. 83 Norvig)

1) Do goal state test, only when a node is selected for expansion.
(Reason: Bucharest may occur on frontier with a longer than optimal path. It won't be selected for expansion yet. Other nodes will be expanded first, leading us to uncover a shorter path to Bucharest. See also point 2).

2) Graph-search alg. says "don't add child node to frontier if already on explored list or already on frontier." BUT, child may give a shorter path to a state already on frontier. Then, we need to modify the existing node on frontier with the shorter path. See fig. 3.14 (else-if part).

Summary of algorithms (for notes)

Criterion	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Complete ?	YES	YES	NO	NO	YES	YES
Time	b^d	$b^{1+C^*/e}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	$b^{1+C^*/e}$	bm	bl	bd	$b^{d/2}$
Optimal?	YES	YES	NO	NO	YES	YES

Assumes b is finite

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

Figure 3.21 Evaluation of tree-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.

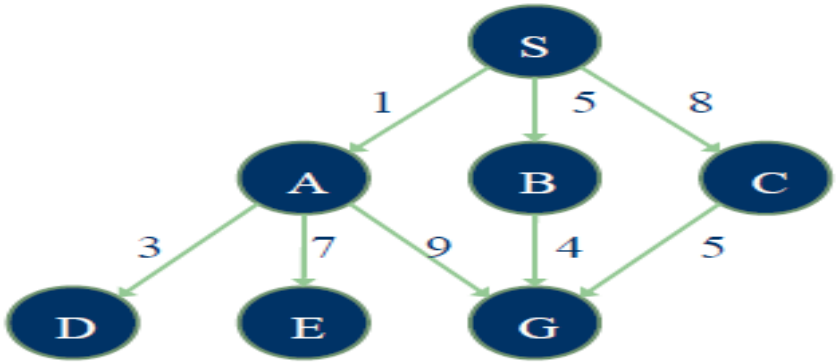
How many types are available in uninformed search method?
a) 3
b) 4
c) 5
d) 6

Which search strategy is also called as blind search?
a) Uninformed search
b) Informed search
c) Simple reflex search
d) All of the mentioned

Which search is implemented with an empty first-in-first-out queue?
a) Depth-first search
b) Breadth-first search
c) Bidirectional search
d) None of the mentioned

Which search is implemented with stack(Last In First Out)?
a) Depth-first search
b) Breadth-first search
c) Bidirectional search
d) None of the mentioned

- Which search is implemented with priority queue?
  a) Depth-first search
  b) Breadth-first search
  c) Bidirectional search
  d) Uniform Cost Search
- Uniform-cost search expands the node n with the _____
  a) Lowest path cost
  b) Heuristic cost
  c) Highest path cost
  d) Average path cost
- Which of the following is false?
- UCS is complete and optimal
- UCS is complete but not optimal
- IDS is complete and optimal
- DFS is not compete and not optimal



How are nodes expanded by

- Depth First Search
- Breadth First Search
- Uniform Cost Search
- Iterative Deepening

Are the solutions the same?

The DFS traversal from S to G is

a) **S A D E G** b) **S A B C D E G** c) **S A G** d) **S B G**

The BFS traversal from S to G is

a) S A D E G **b) S A B C D E G** **c) S A D B C E G** **d) S A B C S A D E G**

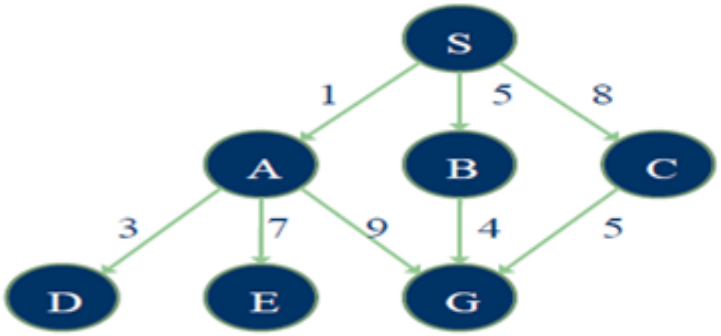
The IDS traversal from S to G is

a) S A D B C E G **b) S A B C S A D E G** **c) S A B C D E G** **d) S A D E G**

The Uniform Cost Search traversal is

a) S A D E G **b) S A B C D E G** **c) S A D B C E G** **d) S A B C S A D E G**

Example



How are nodes expanded by

- Depth First Search
- Breadth First Search
- Uniform Cost Search
- Iterative Deepening

Are the solutions the same?

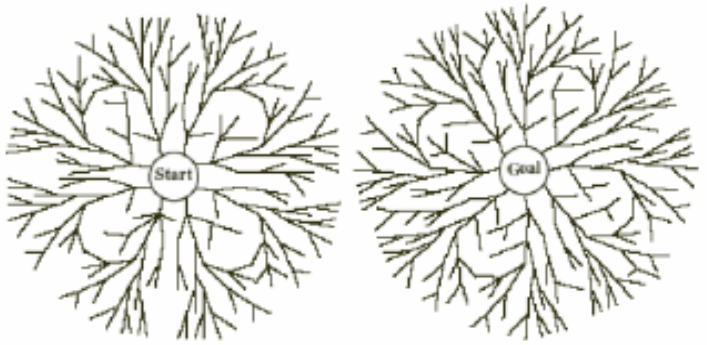
Nodes Expanded by:

- **Depth-First Search: S A D E G**
  Solution found: S A G
- **Breadth-First Search: S A B C D E G**
  Solution found: S A G
- **Uniform-Cost Search: S A D B C E G**
  Solution found: S B G
- **Iterative-Deepening Search: S A B C S A D E G**
  Solution found: S A G

Bidirectional Search

- Simultaneously:
  - Search forward from start
  - Search backward from the goal

  Stop when the two searches meet.



- If branching factor = b in each direction, with solution at depth d

  ➔ only $O(2\ b^{d/2}) = O(2\ b^{d/2})$

- Checking a node for membership in the other search tree can be done in constant time (hash table)

- Key limitations:

  Space $O(b^{d/2})$

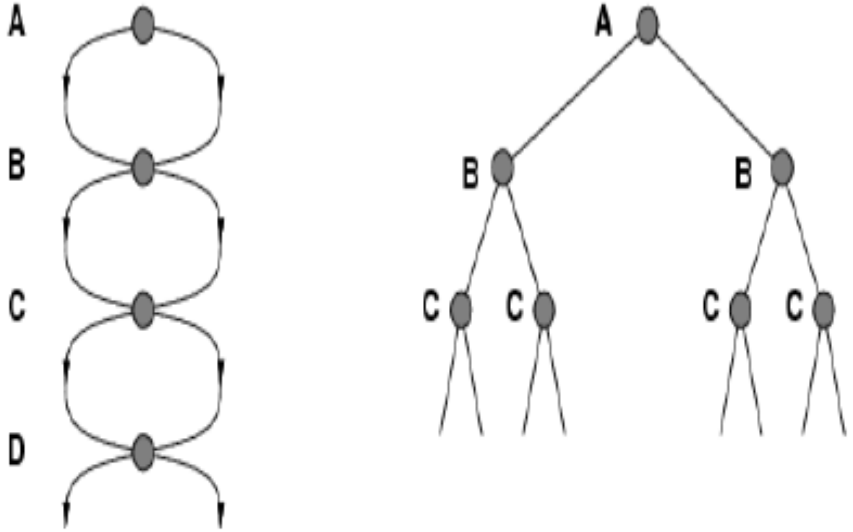
  Also, how to search backwards can be an issue (e.g., in Chess)? What's tricky?

  Problem: lots of states satisfy the goal; don't know which one is relevant.

Aside: The predecessor of a node should be easily computable (i.e., actions are easily reversible).

Repeated states

Failure to detect repeated states can turn linear problem into an exponential one!



Don't return to parent node

Don't generate successor = node's parent

Don't allow cycles

Don't revisit state

Keep every visited state in memory!
$O(b^d)$ (can be expensive)

Problems in which actions are reversible (e.g., routing problems or sliding-blocks puzzle). Also, in eg Chess; uses hash tables to check for repeated states. Huge tables 100M+ size but very useful.

See Tree-Search vs. Graph-Search in Fig. 3.7 R&N. But need to be careful to maintain (path) optimality and completeness.

- In graph search algorithm to get rid of repeated states we need to

  a) create a frontier

  b) create list of explored set where visited vertices are stored

  c) Create a list of vertices yet to be travelled

  d) Create a list of edges yet to be travelled

Summary: General, uninformed search

Original search ideas in AI where inspired by studies of human problem solving in, eg, puzzles, math, and games, but a great many AI tasks now require some form of search (e.g. find optimal agent strategy; active learning; constraint reasoning; NP-complete problems require search).

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.

Variety of uninformed search strategies

Iterative deepening search uses only linear space and not much more time than other uninformed algorithms.

Avoid repeating states / cycles.

Search

Search strategies determined by choice of node (in queue) to expand

Uninformed search:

- Distance to goal not taken into account

Informed search :

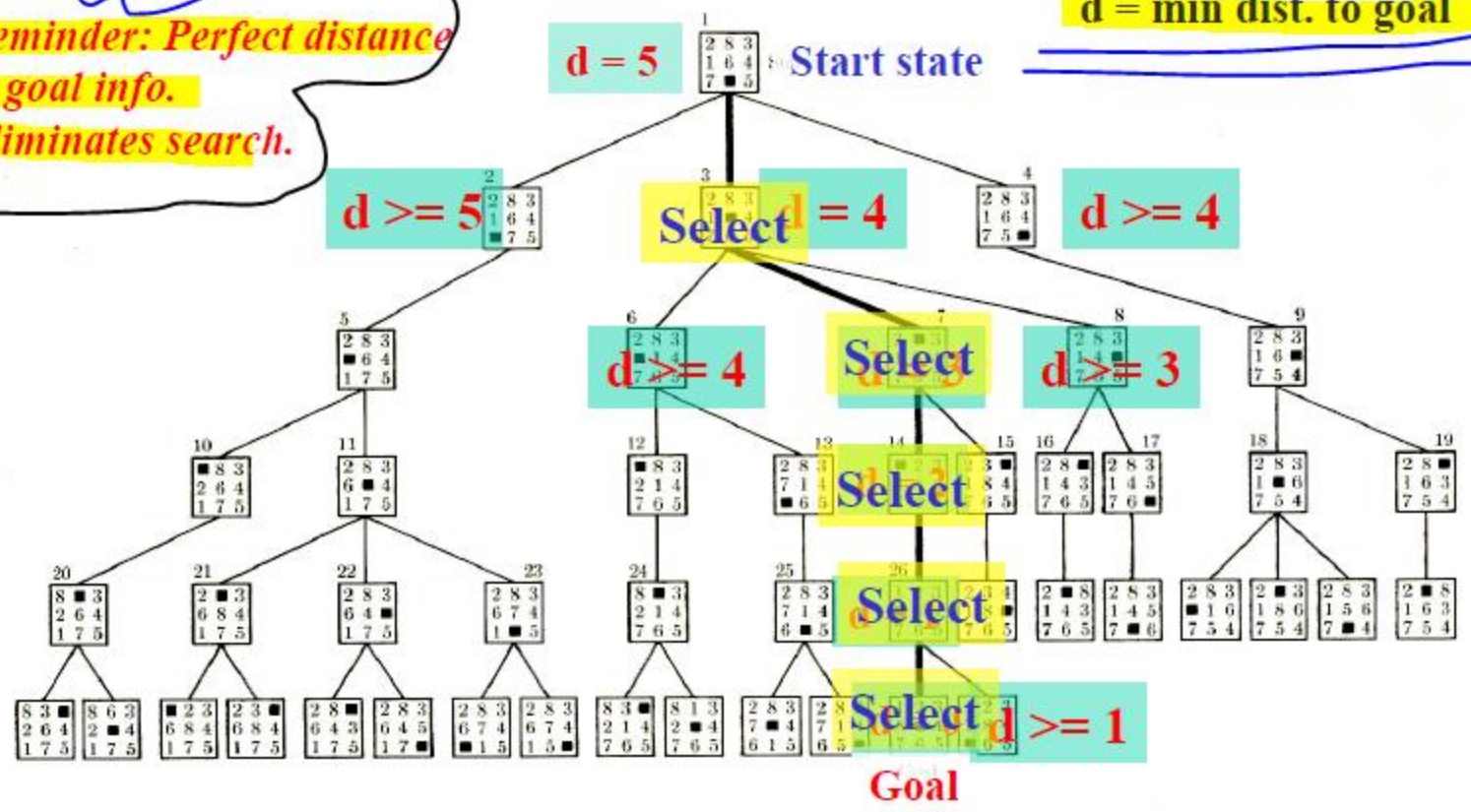
- Information about cost to goal taken into account

Aside: "Cleverness" about what option to explore next, almost seems a hallmark of intelligence. E.g., a sense of what might be a good move in chess or what step to try next in a mathematical proof. We don't do blind search…

INFORMED SEARCH

Reminder: Perfect distance to goal info. Eliminates search.

d = min dist. to goal



A breadth-first search tree.

Practice:

Only have estimate of distance to goal ("heuristic information").

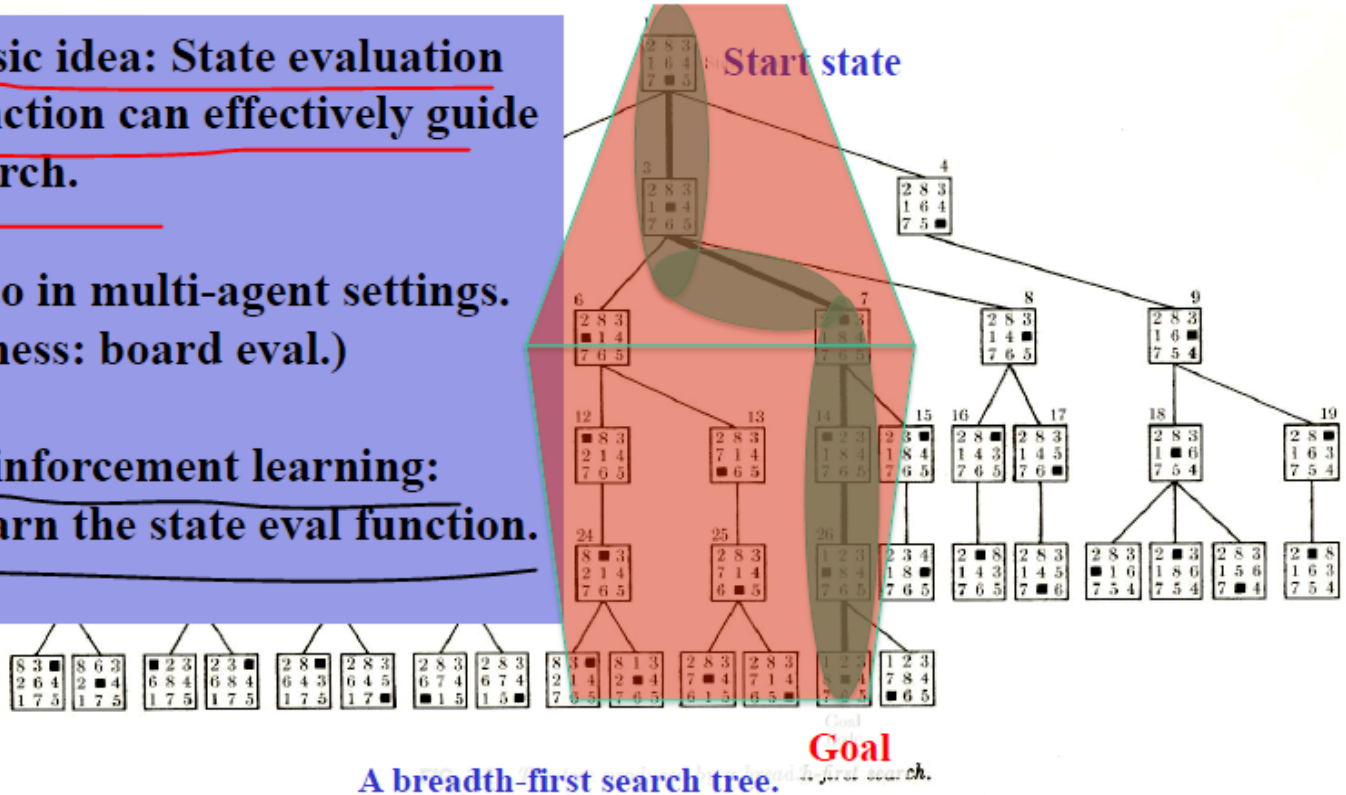
Informed Search Concepts

Informed Search

Basic idea: State evaluation function can effectively guide search.

Also in multi-agent settings. (Chess: board eval.)

Reinforcement learning: Learn the state eval function.



A breadth-first search tree.

Perfect "heuristics," eliminates search.

Approximate heuristics, significantly reduces search.

Best (provably) use of search heuristic info: Best-first / A* search.

Beware while reading RL chapter in Machine Learning

The evaluation function →f(n)

The **heuristic function** →h(n) = estimated cost of the cheapest path from the state at node *n* to a goal state.

Depending on the algorithm f (n) = h(n) or f(n) = h(n) + g(n)

- Best-first search → a general TREE-SEARCH or GRAPH-SEARCH algorithm in which **a node is selected for expansion based on an evaluation function, f(n).**
- The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first.
- The implementation of best-first graph search is identical to that for uniform-cost search (Figure
- 3.14), **except for the use of f instead of g to order the priority queue.**

- **The choice of f determines the search strategy.** (For example, as Exercise 3.21 shows,
- best-first tree search includes depth-first search as a special case.)
- **Most best-first algorithms include as a component of f a heuristic function**, h(n)

Where **h(n) = estimated cost of the cheapest path from the state at node *n* to a goal state.**

- (Notice that h(n) takes a *node* as input, but, unlike g(n), it depends only on the *state* at that
- node.)
- For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.
- **Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm.**

Informed Search

- Informed searches use domain knowledge to guide selection of the best path to continue searching
- Heuristics are used, which are informed guesses
- Heuristic means "serving to aid discovery"

Define a heuristic function, $h(n)$

- uses domain-specific information in some way
- is (easily) computable from the current state description
- estimates
  - the "goodness" of node n
  - how close node n is to a goal
  - the cost of *minimum* cost path from node n to a goal state

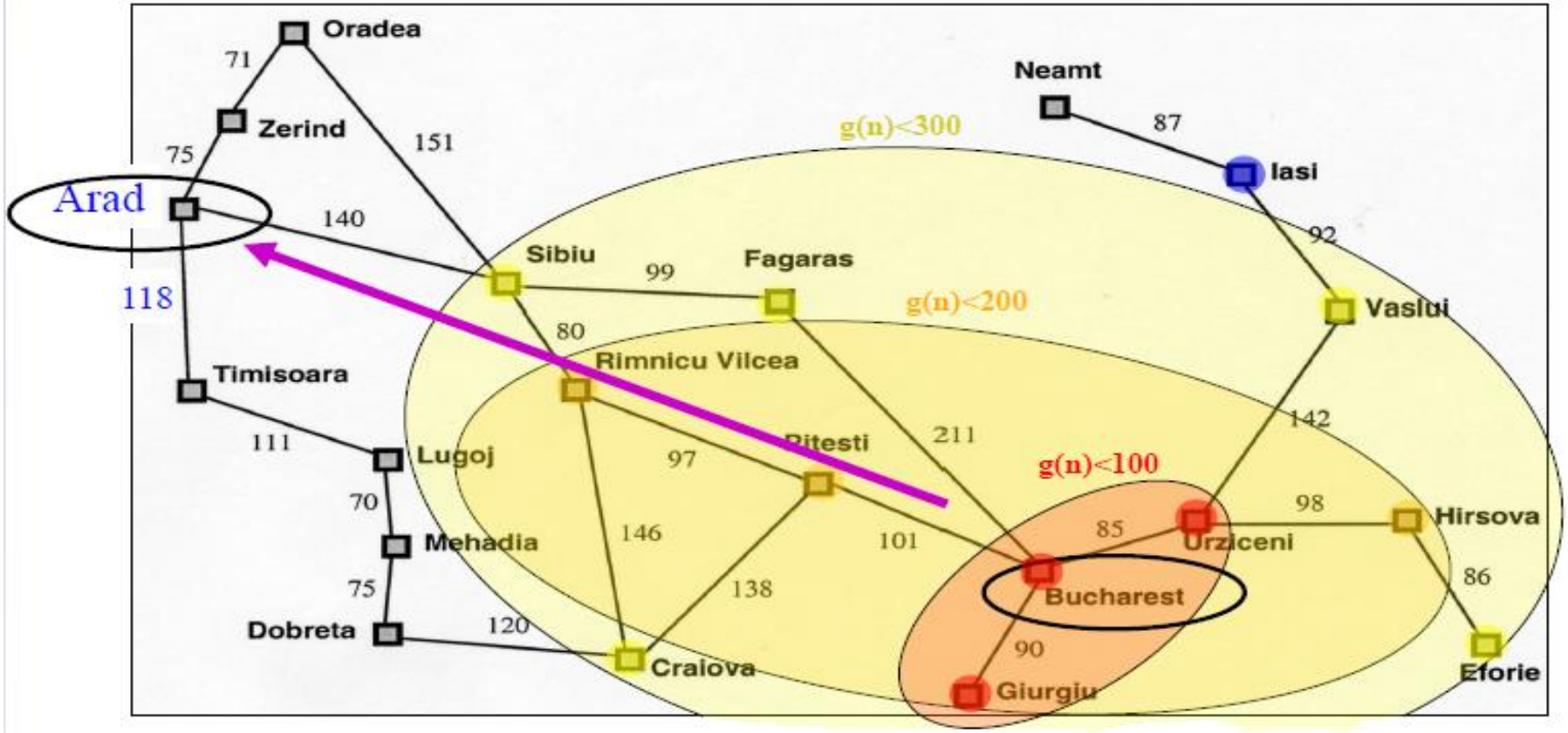
Informed Search

- $h(n) \geq 0$ for all nodes n
- $h(n)$ close to 0 means we think n is close to a goal state
- $h(n)$ very big means we think n is far from a goal state

- All domain knowledge used in the search is encoded in the heuristic function, h

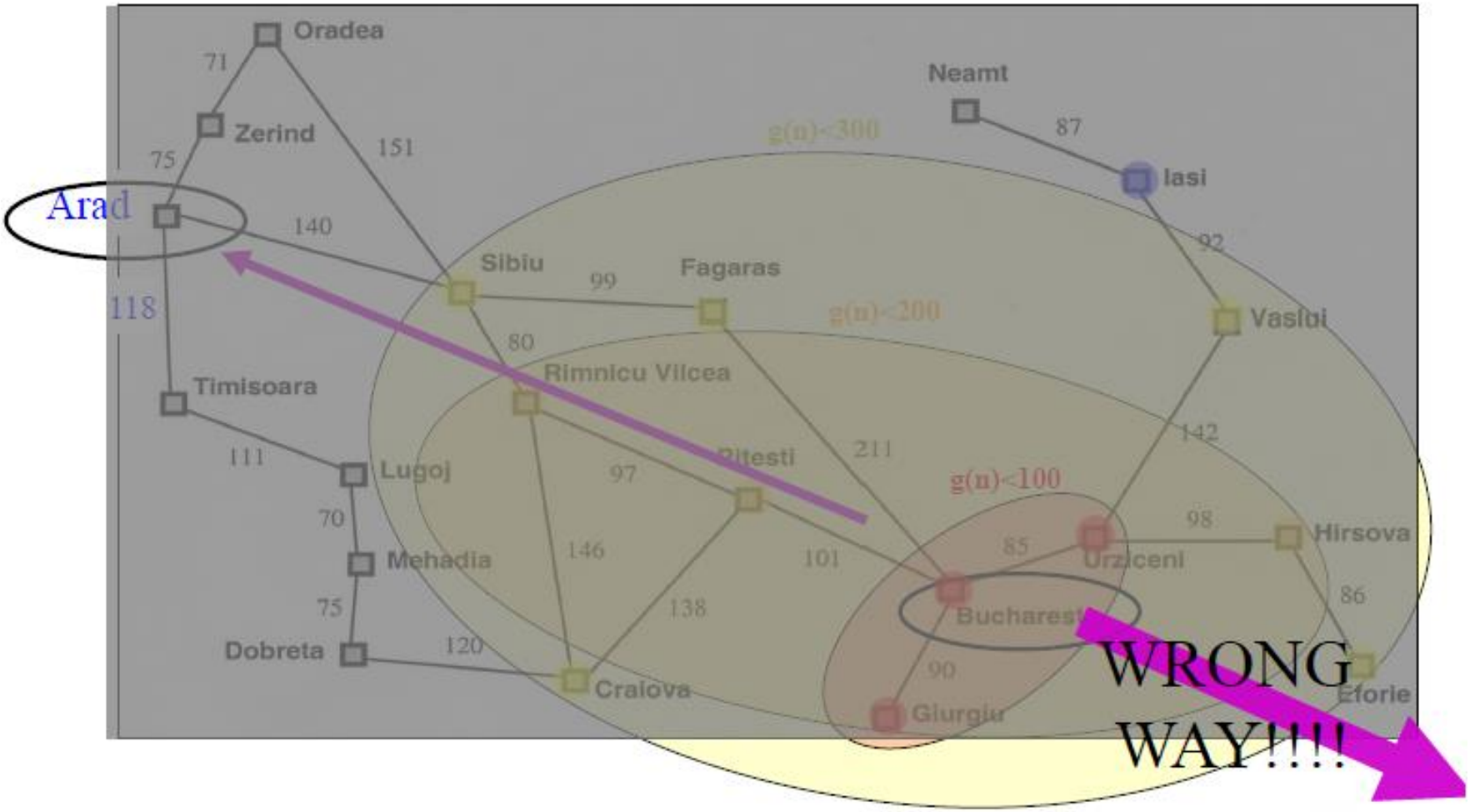
Is Uniform Cost Search the best we can do?

Consider finding a route from Bucharest to Arad..



Is Uniform Cost Search the best we can do?

Consider finding a route from Bucharest to Arad..



A Better Idea…

- **Node expansion based on *an estimate* which *includes distance to the goal***

- **General approach of informed search:**
  - *Best-first search*: node selected for expansion based on an *evaluation function* ***f(n)***
    - —***f(n)*** includes *estimate* of distance to goal ***(new idea!)***

- **Implementation: Sort frontier queue by this new *f(n)*.**
  - Special cases: greedy search, ***A* search***

Informed search → Applying heuristic in search algorithm i.e. we encode additional knowledge about the problem into account using F(n)

→ heuristic Knowledge

How to take information into account? Best-first search.

Idea : use an evaluation function for each node

- Estimate of "desirability" of node
- Expand most desirable unexpanded node first ("best-first search")
- Heuristic Functions :
  - *f*: States → Numbers
  - *f(n)*: expresses the quality of the state *n*
    - Allows us to express problem-specific knowledge.
    - Can be imported in a generic way in the algorithms.
- Use uniform-cost search. See Figure 3.14 but use f(n) instead of path cost g(n).
- Queuing based on f(n): → frontier

  Order the nodes in fringe in decreasing order of desirability

Use

Special cases:

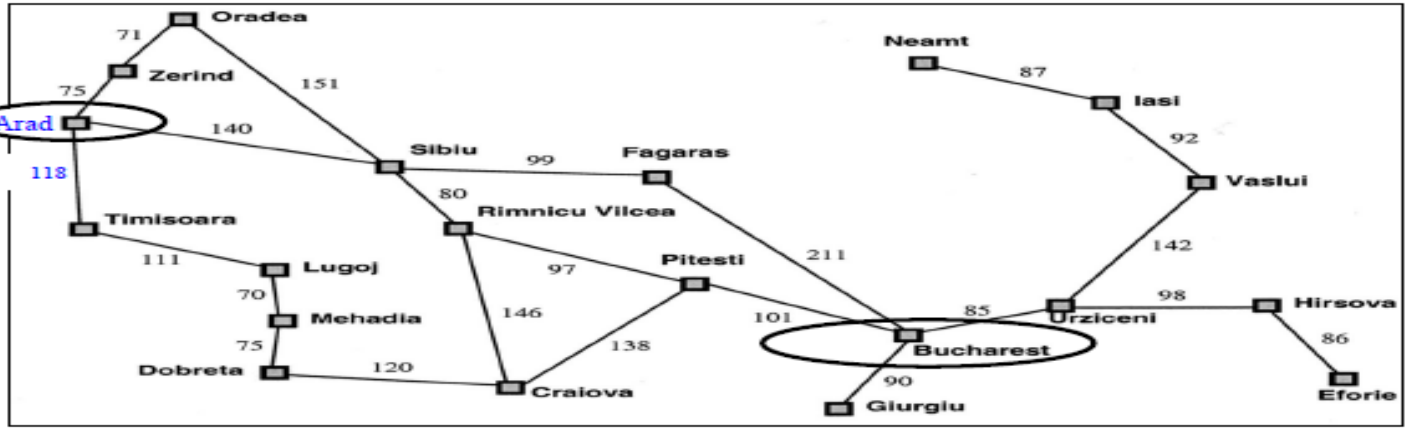
- **greedy best-first search**
- **A* search**

i.e. the most desirable node be on the front of queue

The Heuristics approach → To make search algorithms perform better, make an estimate of goal

i.e. calculate the straight line distance from Bucharest to every other city and encode this information in f(n)→

Simple, useful estimate *heuristic*: *straight-line distances*



Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Heuristic (estimate) functions

Heureka! ---Archimedes

[dictionary] *"A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood."*

Heuristic knowledge is useful, but not necessarily correct.

Heuristic algorithms use heuristic knowledge to solve a problem.

A ***heuristic function*** $h(n)$ takes a state n and returns an *estimate* of the distance from n to the goal.

(graphic: http://hyperbolegames.com/2014/10/20/eureka-moments/)

Compare how does some driving app calls you a vehicle when given your ride location

Best-First Search

- Sort nodes in the Frontier list by increasing values of an evaluation function, $f(n)$, that incorporates domain-specific information

- This is a generic way of referring to the class of informed search methods

Basic idea:

- ***select node for expansion*** **with minimal** ***evaluation function*** $f(n)$
  - where $f(n)$ is some function that includes ***estimate heuristic*** $h(n)$ of the remaining distance to goal

- **Implement using priority queue**
- **Exactly UCS with** $f(n)$ **replacing** $g(n)$

Greedy best-first search

Evaluation function at node n, $f(n) = h(n)$ (heuristic)

= *estimate* of cost from n to *goal*

e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest

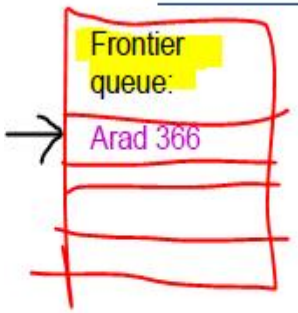
Expands the node that *is estimated* to be closest to goal

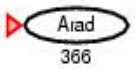
Completely ignores *g(n):* the cost to get to *n*

Idea: those nodes may lead to solution quickly.

Similar to depth-first search: It prefers to follow a single path to goal (guided by the heuristic), backing up when it hits a dead-end.

Greedy best-first search example



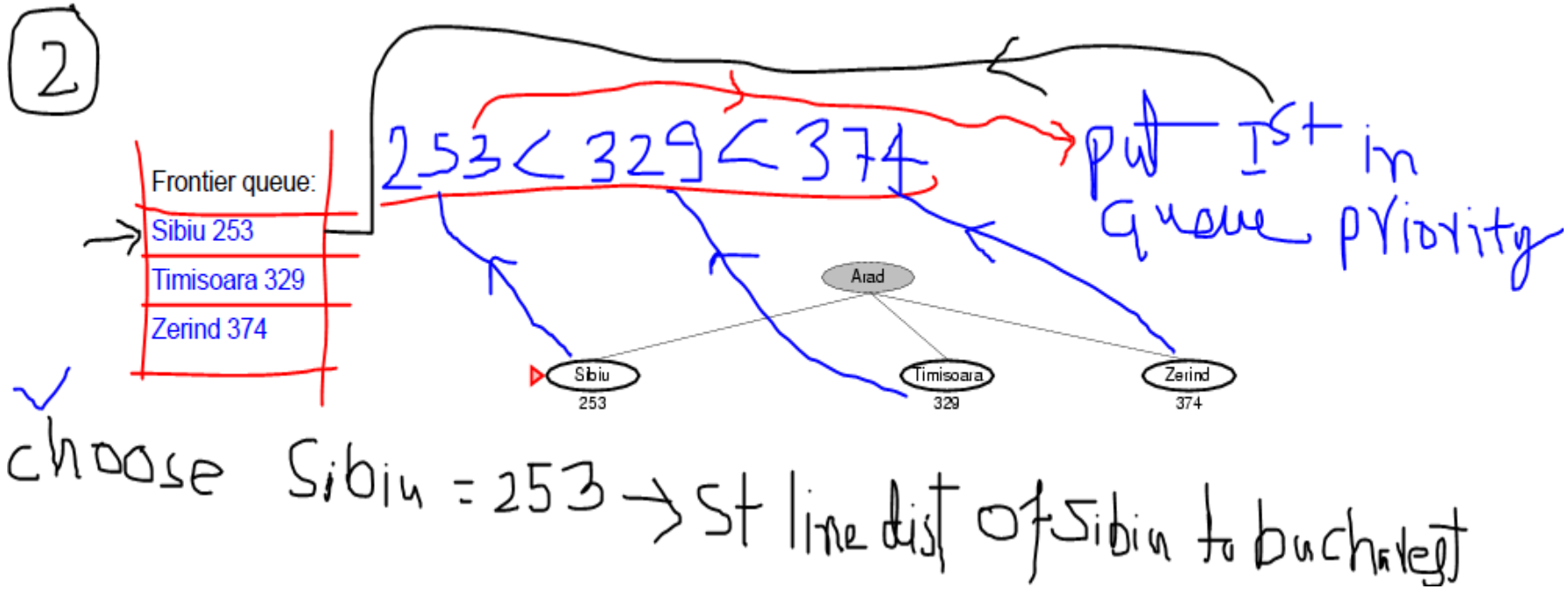


- Initial State = Arad
- Goal State = Bucharest

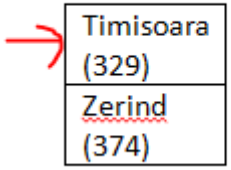
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

↳ St line distance from node (n) to goal node (Bucharest)

Greedy Best First Search(BFS) not to be confused with uninformed BFS(Breadth First Search)



- i.e. choose Sibiu from queue using dequeue operation and after Dqueuing operation on most prioritized element the queue's state will be as below:



3

Frontier queue:

Fagaras 176

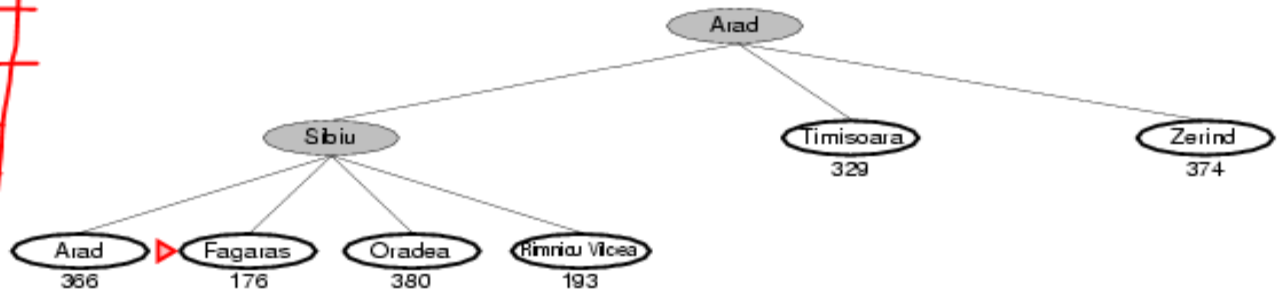
Rimnicu Vilcea 193

Timisoara 329

Arad 366

Zerind 374

Oradea 380



4

Frontier queue:

Bucharest 0

Rimnicu Vilcea 193

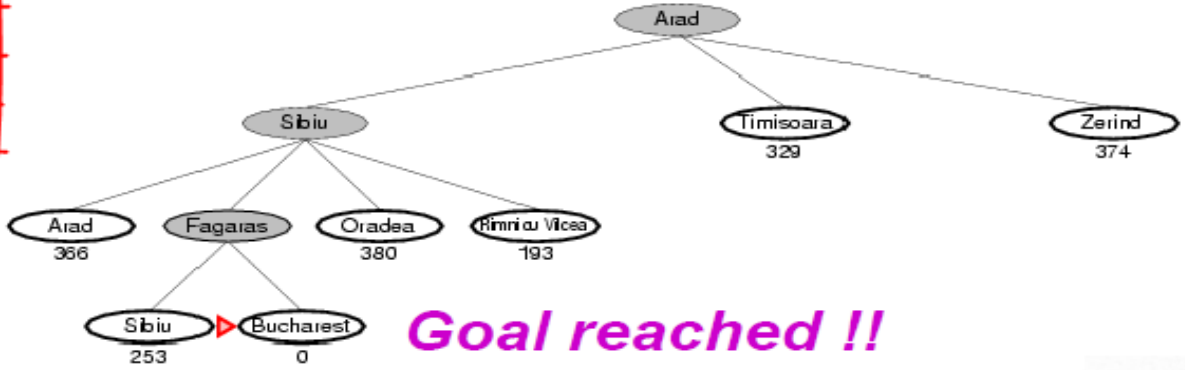
Sibiu 253

Timisoara 329

Arad 366

Zerind 374

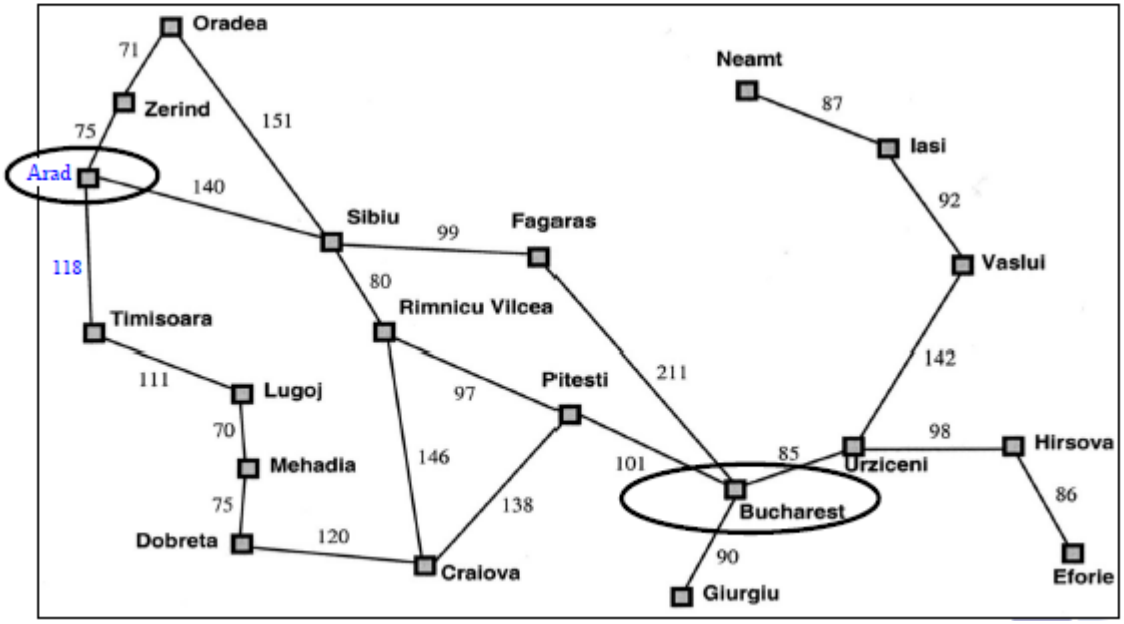
Oradea 380



Goal reached !!

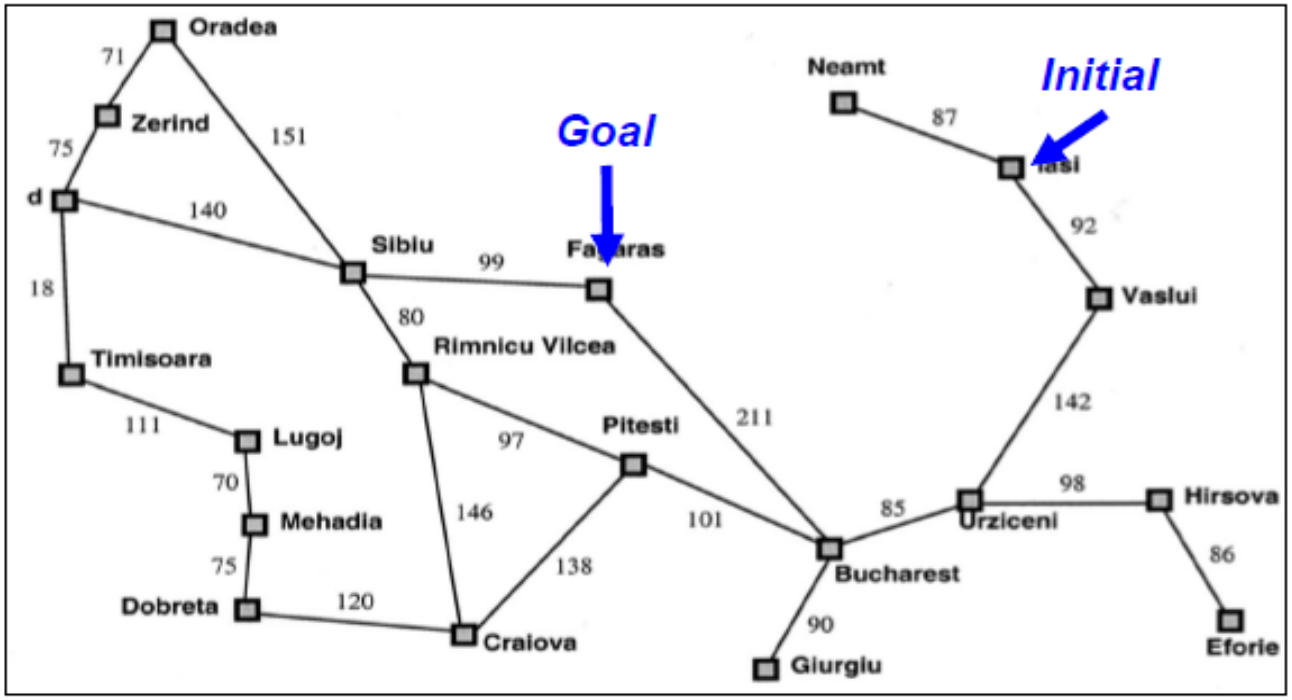
Properties of greedy best-first search

- ***Optimal?***
  - No!
    - —Found: *Arad → Sibiu → Fagaras → Bucharest (450km)*
    - —Shorter: *Arad → Sibiu → Rimnicu Vilcea → Pitesti → Bucharest (418km)*



Properties of greedy best-first search

- ***Complete?***
  - No – can get stuck in loops,
  - e.g., Iasi → Neamt → Iasi → Neamt →…



Another Example→ Note the color change of each node when taken for expansion

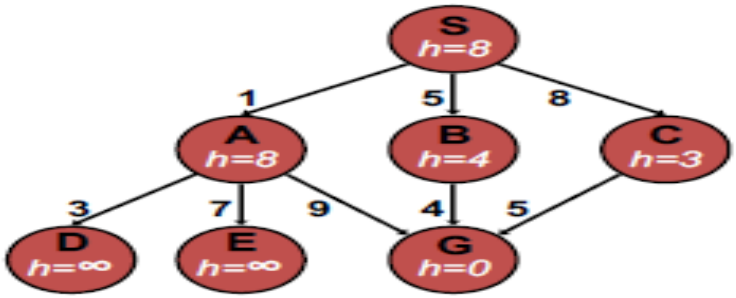
Greedy Best-First Search

1

$f(n) = h(n)$

# of nodes tested: 0, expanded: 0

expnd. node	Frontier
	{S:8}



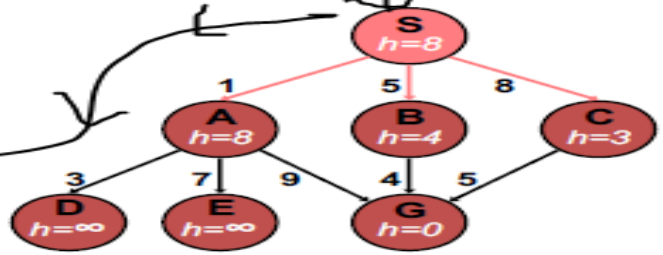
Node expanded

2

$f(n) = h(n)$

# of nodes tested: 1, expanded: 1

expnd. node	Frontier
	{S:8}
S not goal	{C:3,B:4,A:8}



S' child arranged in ascending order

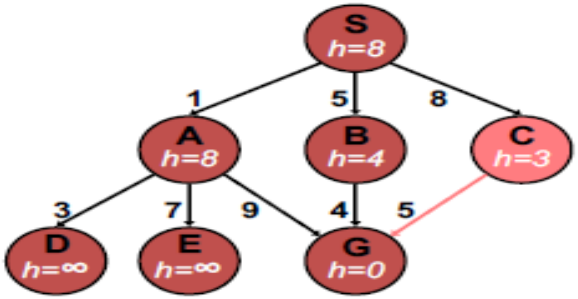
3

Greedy Best-First Search

$f(n) = h(n)$

# of nodes tested: 2, expanded: 2

expnd. node	Frontier
	{S:8}
S	{C:3,B:4,A:8}
C not goal	{G:0,B:4,A:8}

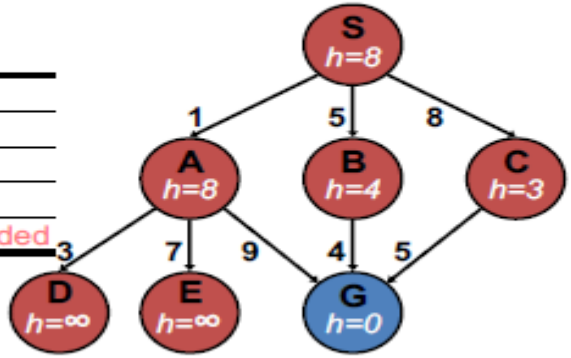


4

$f(n) = h(n)$

# of nodes tested: 3, expanded: 2

expnd. node	Frontier
	{S:8}
S	{C:3,B:4,A:8}
C	{G:0,B:4, A:8}
G goal	{B:4, A:8} not expanded

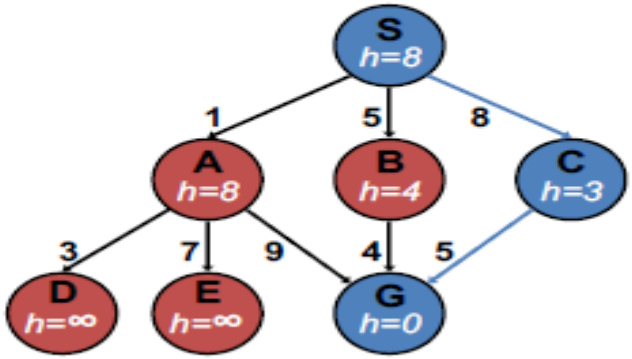


Greedy Best-First Search

$f(n) = h(n)$

# of nodes tested: 3, expanded: 2

expnd. node	Frontier
	{S:8}
S	{C:3,B:4,A:8}
C	{G:0,B:4, A:8}
G	{B:4, A:8}



- *Fast but not optimal*

path: S,C,G
cost: 13

Properties of greedy best-first search

- ***Complete?*** **No – can get stuck in loops,**
  - e.g., Iasi → Neamt → Iasi → Neamt → …

But, complete in finite space with repeated state elimination.

- ***Time?*** ***O(b^m) – worst case*** **(like Depth First Search)**
  - But a good heuristic can give dramatic improvement of *average cost*

- ***Space?*** ***O(b^m)*** **– priority queue, so worst case: keeps all (unexpanded) nodes in memory**

- ***Optimal?*** **No**

How can we fix this?

b: maximum branching factor of the search tree

d: depth of the least-cost solution

m: maximum depth of the state space (may be ∞)

A* Search

Note: Greedy best-first search expands the node that *appears* to have shortest path to goal. But what about cost of getting to that node? Take it into account!

Best-known form of best-first search.

Idea: avoid expanding paths that are already expensive

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

$g(n)$ → actual cost to get from start node to *a node (n)*

$h(n)$ → estimated cost from a node (n) to goal → Straight line distance from node n to goal

– $f(n)$ = estimated total cost of path through n to goal

Implementation: Frontier queue as priority queue by increasing $f(n)$ *(as expected…)*

Aside: do we still have "looping problem"?

Iasi to Fagaras:

Iasi → Neamt → Iasi → Neamt…

No! We'll eventually get out of it. g(n) keeps going up.

Admissible heuristics

- **A heuristic $h(n)$ is *admissible* if it *never overestimates* the cost to reach the goal; i.e. it is *optimistic***
  - Formally: $\forall n$, n a node:
    1. $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n
    2. $h(n) \geq 0$ so $h(G)=0$ for any goal G.

- ***Example:* $h_{SLD}(n)$ never overestimates the actual road distance**

Theorem: If $h(n)$ is *admissible*, A* using Tree Search is *optimal*

Since h(n) is st line distance, so $h(n) <= h^*(n)$

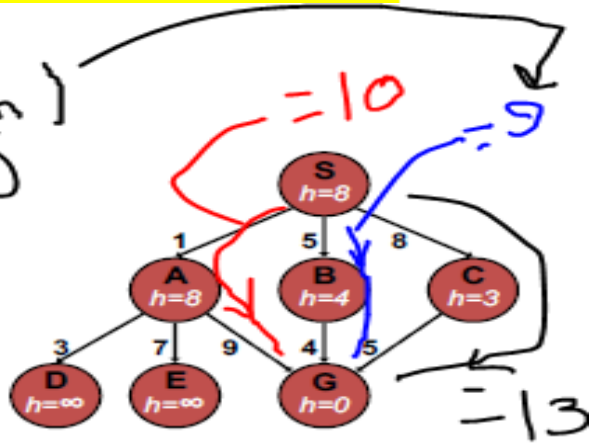
Note→ $h^*(n)$ is the minimum cost path among many actual cost paths from n to goal

- Use the same evaluation function used by Algorithm A, except add the constraint that for *all* nodes n in the search space, $h(n) \leq h^*(n)$, where $h^*(n)$ is the *actual* cost of the minimum-cost path from n to a goal
- The cost to the nearest goal is **never *over-estimated***
- When $h(n) \leq h^*(n)$ holds true for *all* n, *h* is called an **admissible heuristic function**
- **An admissible heuristic guarantees that a node on the optimal path cannot look so bad that it is never considere**

From S→G $h^*(n)$

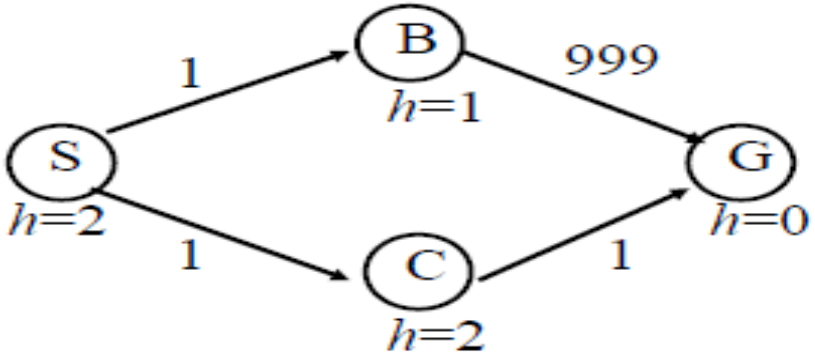
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	∞
E	8	∞	∞	∞
G	10/9/13	0	10/9/13	0

Since $h(n) \leq h^*(n)$ for all n, h is admissible



When should A* Stop?

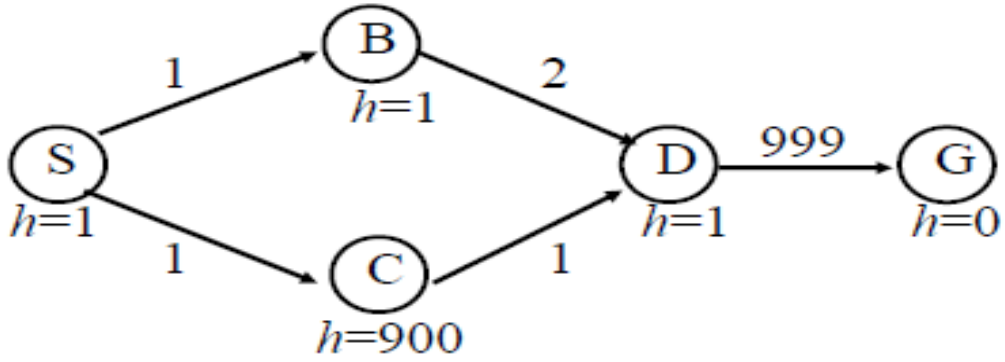
- A* should terminate only when a goal is ***removed*** from the priority queue



- Same rule as for Uniform-Cost Search (UCS)
- A* with $h() = 0$ ***is*** Uniform-Cost Search

A* Revisiting States

- One more complication: A* might revisit a state (in *Frontier* **or** *Explored*), and discover a ***better*** path



- Solution: Put *D* back in the priority queue, using the smaller *g* value (and path)

A and A* Algorithm for General State-Space **Graphs**

Frontier = {*S*} where *S* is the start node
Explored ={}
Loop do
  if *Frontier* is empty **then return** failure
  pick node, *n*, with min *f* value from *Frontier*
  if *n* is a goal node **then return** solution
  foreach each child, *n'*, of *n* **do**
    if *n'* is not in *Explored* or *Frontier*
      then add *n'* to *Frontier*
    else if $g(n') \geq g(m)$ **then** throw *n'* away
      else add *n'* to *Frontier* and remove *m*
  Remove *n* from *Frontier* and add *n* to *Explored*

Variant of algorithm in Fig. 3.14

Note: *m* is the node in *Frontier* or *Explored* that is the same state as *n'*

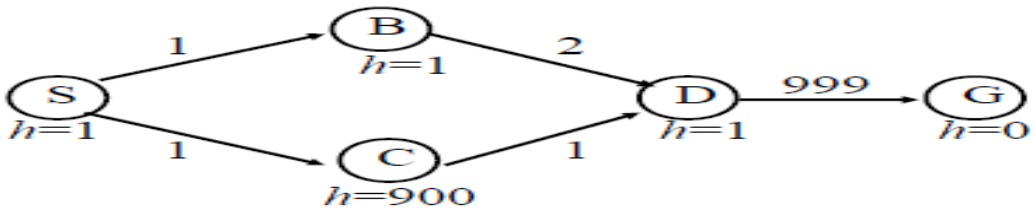
Consistency

- A heuristic, h, is called ***consistent*** (aka ***monotonic***) if, for every node n and every successor n' of n, the estimated cost of reaching the goal from n is *no greater than* the step cost of getting to n' plus the estimated cost of reaching the goal from n':

  $c(n, n') \geq h(n) - h(n')$

  or, equivalently: $h(n) \leq c(n, n') + h(n')$
- **Triangle inequality for heuristics**
- Implies values of f along any path are *nondecreasing*
- *When a node is expanded by A*, the optimal path to that node has been found*
- Consistency is a ***stronger*** condition than admissibility

Is this h consistent?



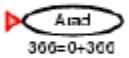
$h(C)=900$, $h(D)=1$, $c(C, D) = 1$
but $h(C) \not\leq c(C,D) + h(D)$ since
$900 \not\leq 1 + 1$, so h is *NOT* consistent
(but h is admissible)

(1)

A* search example

Frontier queue:

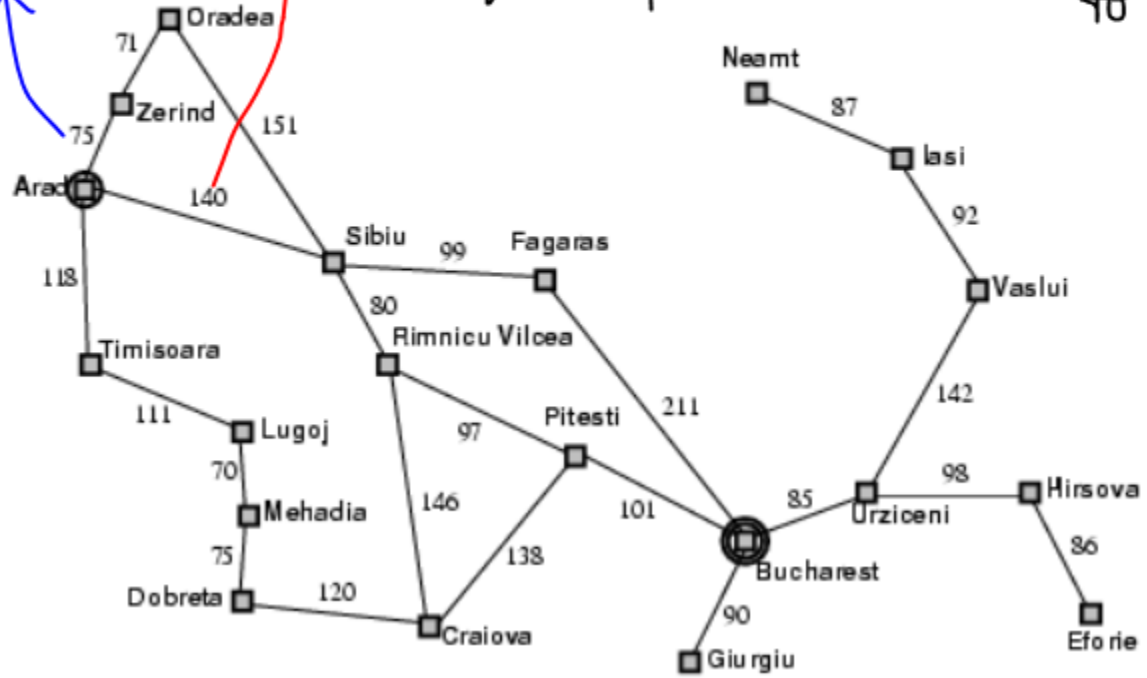
Arad 366



g(n) for A→Z

g(n) for Arad to Sibiu

but g(n) for " " " to Arad = 140+140 = 280



A* Search

Frontier queue:

Sibiu 393

Timisoara 447

Zerind 449

We add the three nodes we found to the Frontier queue.

We sort them according to the g()+h() calculation.

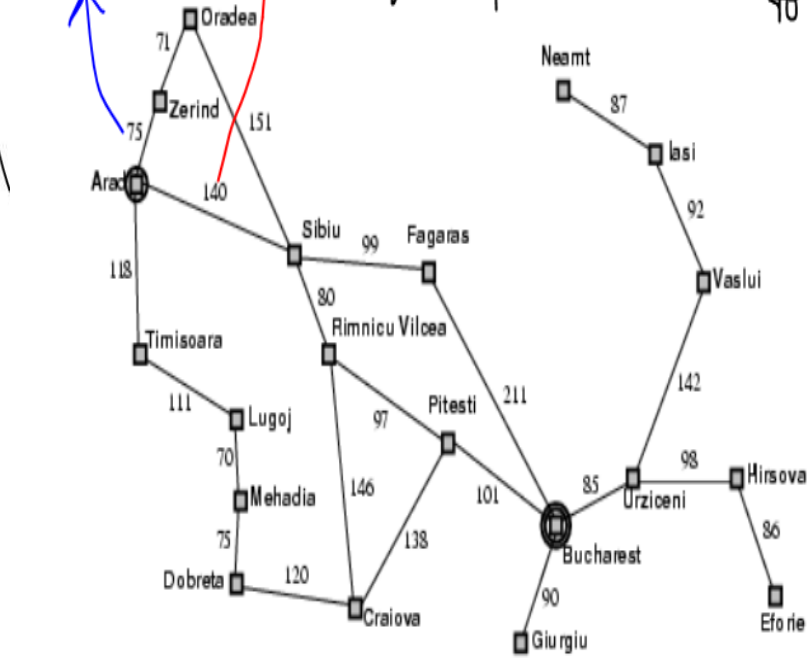
h(n) → st line dist from node(n) to goal

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

g(n) for A→Z

g(n) for Arad to Sibiu

but g(n) for " " " to Arad = 140+140 = 280



A* Search

Frontier queue:

Rimricu Vicea 413

Fagaras 415

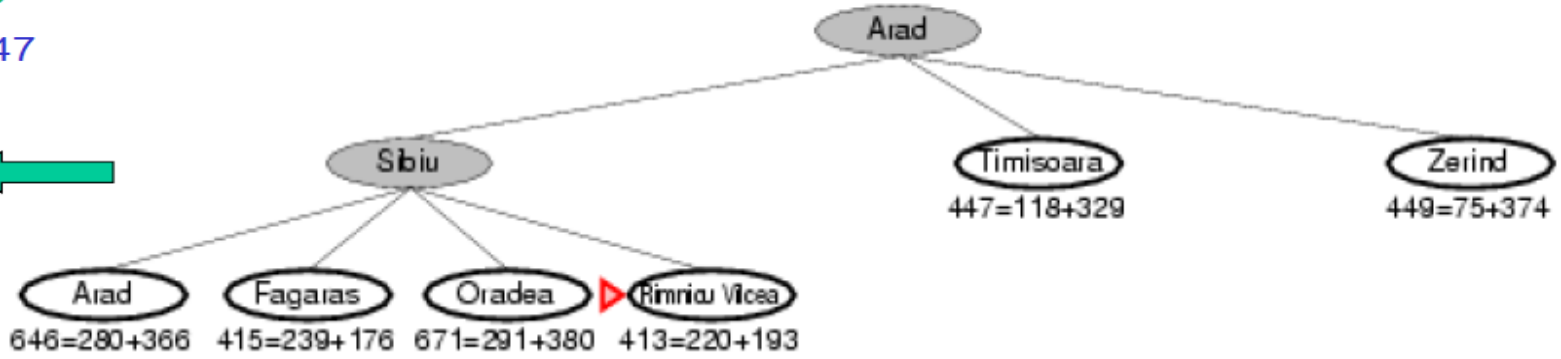
Timisoara 447

Zerind 449

Arad 646

Oradea 671

3



When we expand Sibiu, we run into Arad again. Note that we've already expanded this node once; but we still add it to the Frontier queue again.

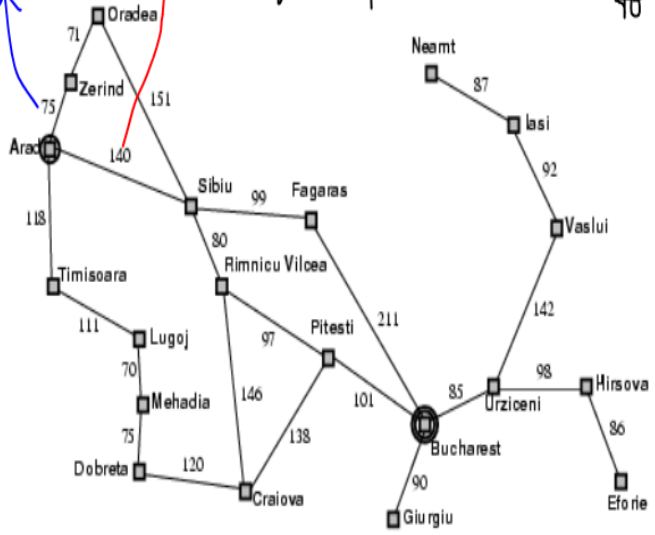
h(n) → st line dist from node(n) to goal

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

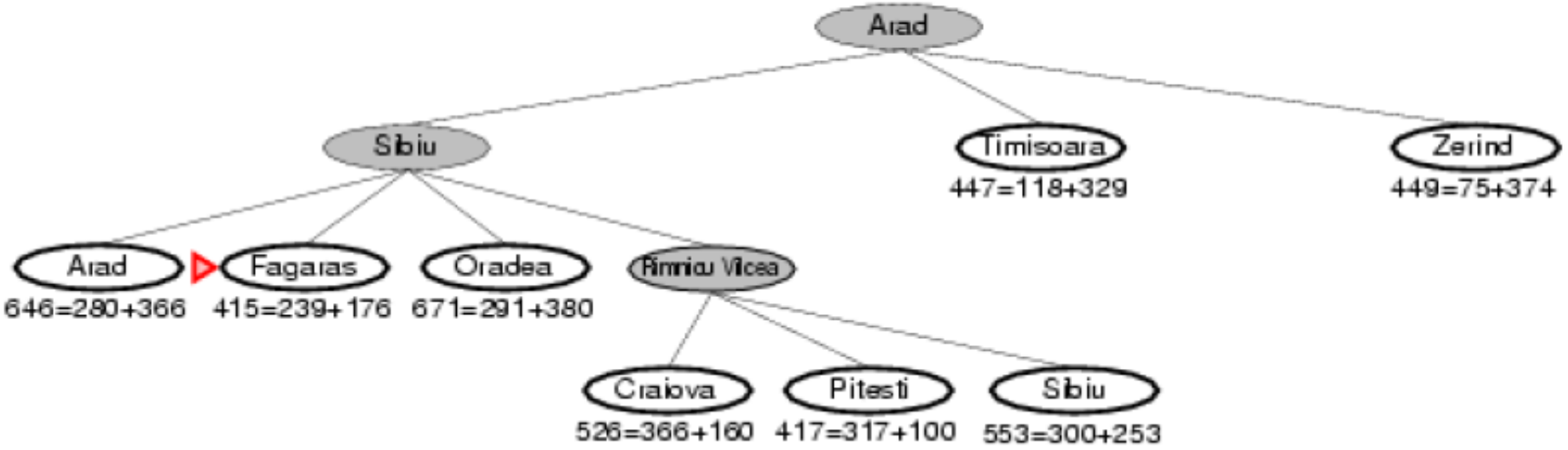
g(n) for A→Z

g(n) for Arad to Sibiu

but g(n) for " " " to Arad = 140+140 = 280



A* Search



Frontier queue:

Fagaras 415

Pitesti 417

Timisoara 447

Zerind 449

Craiova 526

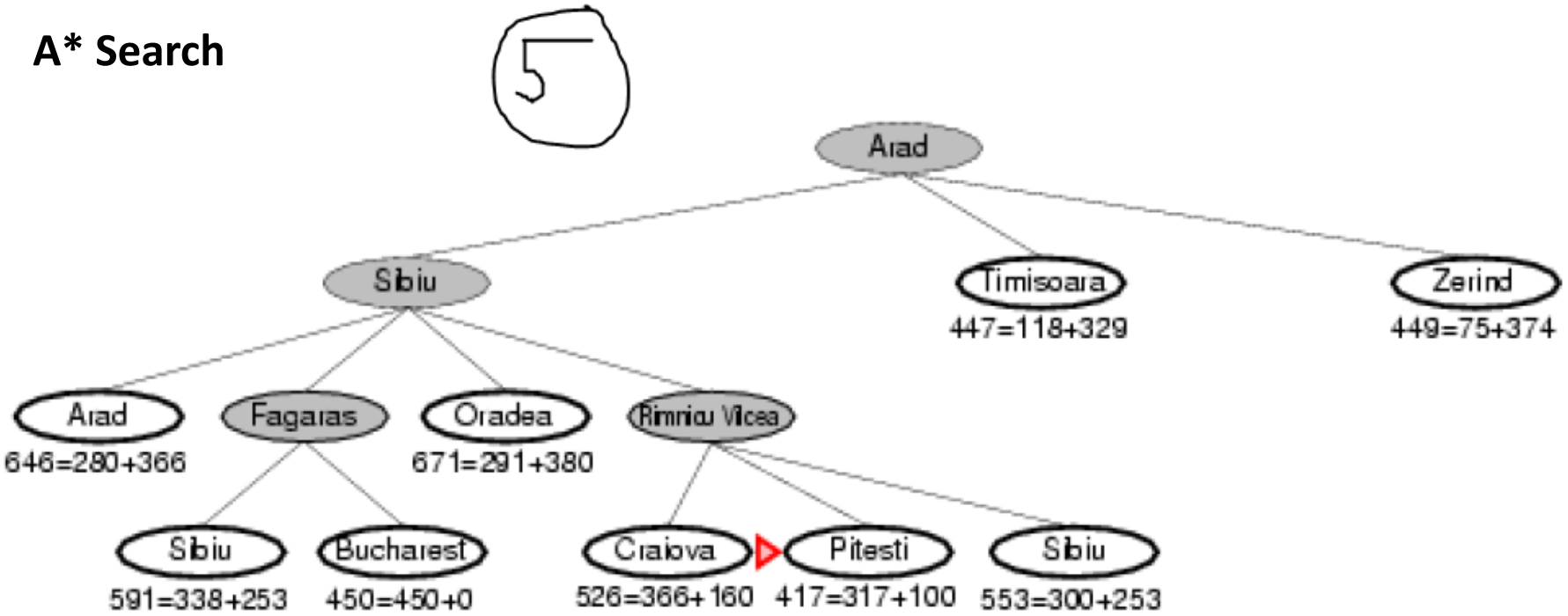
Sibiu 553

Arad 646

Oradea 671

We expand Rimricu Vicea.

A* Search



Frontier queue:

Pitesti 417

Timisoara 447

Zerind 449

Bucharest 450

Craiova 526

Sibiu 553

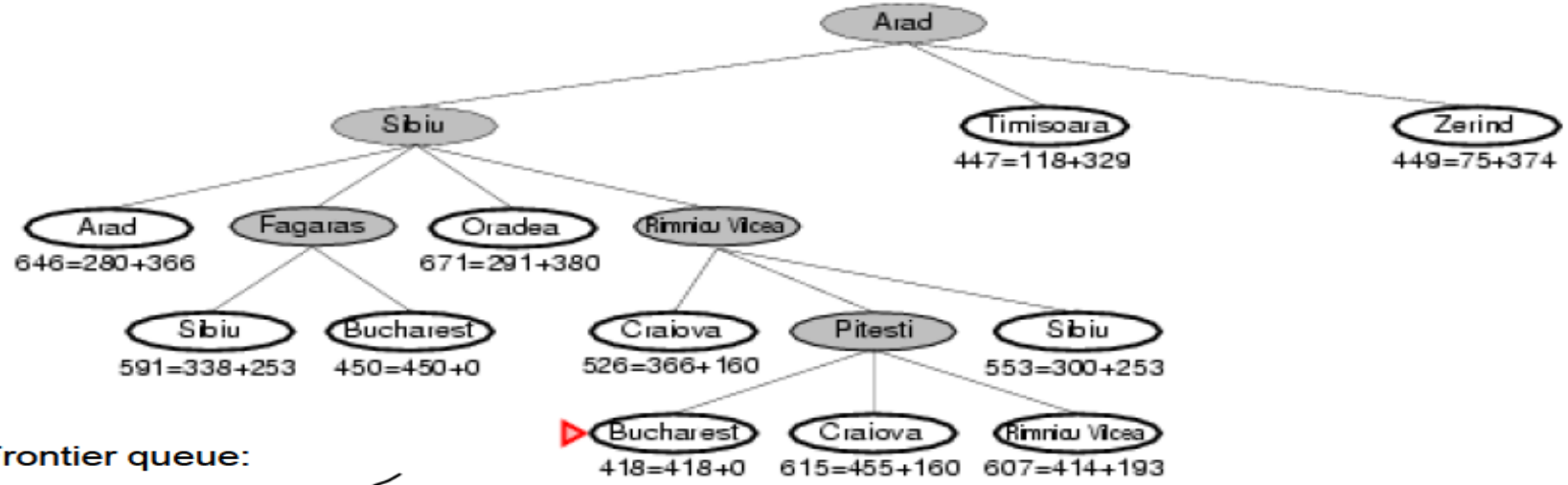
Sibiu 591

Arad 646

Oradea 671

When we expand Fagaras, we find Bucharest, but we're not done. The algorithm doesn't end until we "expand" the goal node – it has to be at the top of the Frontier queue.

A* Search



Frontier queue:

Bucharest 418

Timisoara 447

Zerind 449

Bucharest 450

Craiova 526

Sibiu 553

Sibiu 591

Rimricu Vicea 607

Craiova 615

Arad 646

Oradea 671

Note that we just found a better value for Bucharest!

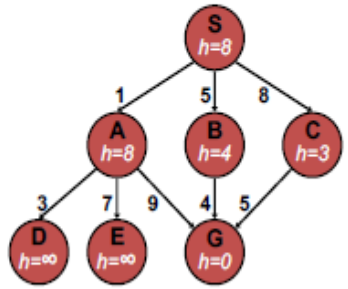
Now we expand this better value for Bucharest since it's at the top of the queue.

We're done and we know the value found is optimal!

g(n),f(n), h(n) and h*(n) calculation in A*

Example

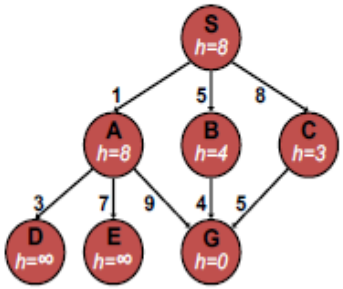
n	g(n)	h(n)	f(n)	h*(n)
S				
A				
B				
C				
D				
E				
G				



$g(n)$ = actual cost to get to node n from start

Example

n	g(n)	h(n)	f(n)	h*(n)
S	0			
A				
B				
C				
D				
E				
G				

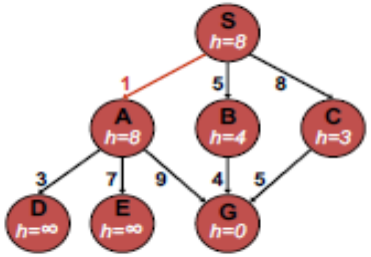


$g(n)$ = actual cost to get to node n from start

g(n),f(n), h(n) and h*(n) calculation in A*

Example

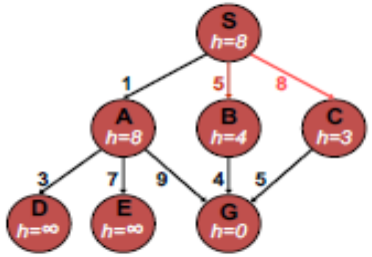
n	g(n)	h(n)	f(n)	h*(n)
S	0			
A	1			
B				
C				
D				
E				
G				



$g(n)$ = actual cost to get to node n from start

Example

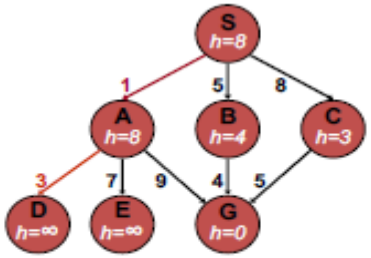
n	g(n)	h(n)	f(n)	h*(n)
S	0			
A	1			
B	5			
C	8			
D				
E				
G				



$g(n)$ = actual cost to get to node n from start

Example

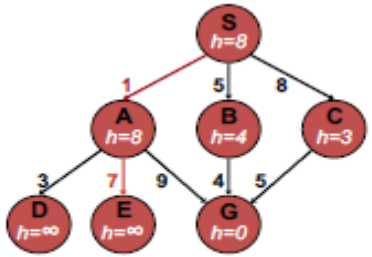
n	g(n)	h(n)	f(n)	h*(n)
S	0			
A	1			
B	5			
C	8			
D	1+3 = 4			
E				
G				



$g(n)$ = actual cost to get to node n from start

Example

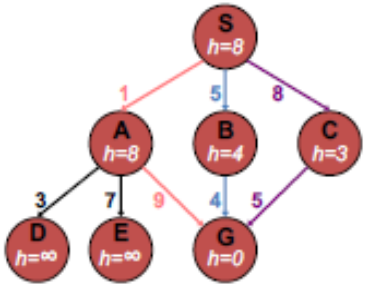
n	g(n)	h(n)	f(n)	h*(n)
S	0			
A	1			
B	5			
C	8			
D	4			
E	1+7 = 8			
G				



$g(n)$ = actual cost to get to node n from start

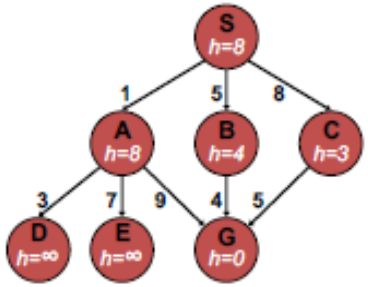
g(n),f(n), h(n) and h*(n) calculation in A*

n	g(n)	h(n)	f(n)	h*(n)
S	0			
A	1			
B	5			
C	8			
D	4			
E	8			
G	10/9/13			



$g(n)$ = actual cost to get to node n from start

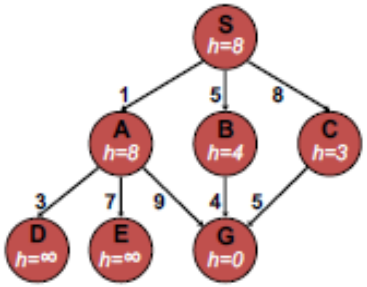
n	g(n)	h(n)	f(n)	h*(n)
S	0			
A	1			
B	5			
C	8			
D	4			
E	8			
G	10/9/13			



$h(n)$ = estimated cost to get to a goal from node n

Example

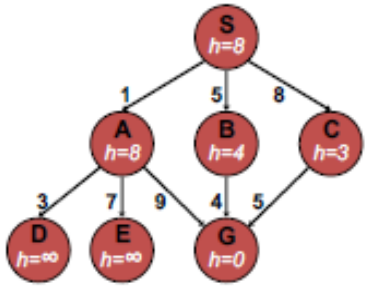
n	g(n)	h(n)	f(n)	h*(n)
S	0	8		
A	1	8		
B	5	4		
C	8	3		
D	4	∞		
E	8	∞		
G	10/9/13	0		



$h(n)$ = estimated cost to get to a goal from node n

Example

n	g(n)	h(n)	f(n)	h*(n)
S	0	8	8	
A	1	8	9	
B	5	4	9	
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



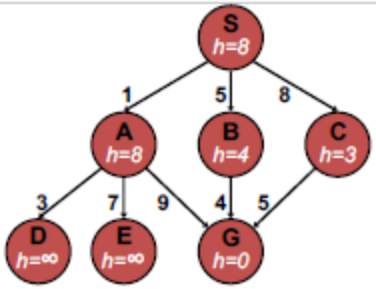
$f(n) = g(n) + h(n)$

actual cost to get from start to n

plus estimated cost from n to goal

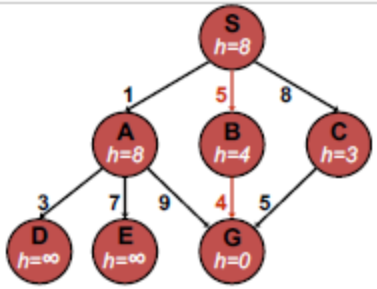
g(n),f(n), h(n) and h*(n) calculation in A*

n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	
A	1	8	9	
B	5	4	9	
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



$h^*(n)$ = true cost of minimum-cost path from n to a goal

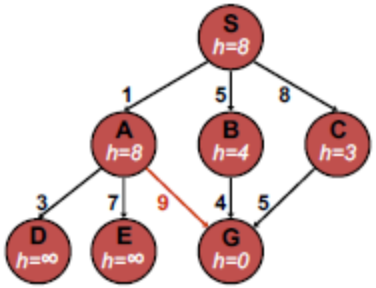
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	
B	5	4	9	
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



$h^*(n)$ = true cost of minimum-cost path from n to a goal

Example

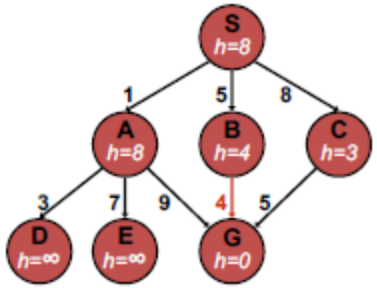
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	9
B	5	4	9	
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



$h^*(n)$ = true cost of minimum-cost path from n to a goal

Example

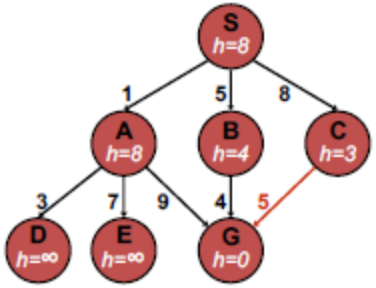
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



$h^*(n)$ = true cost of minimum-cost path from n to a goal

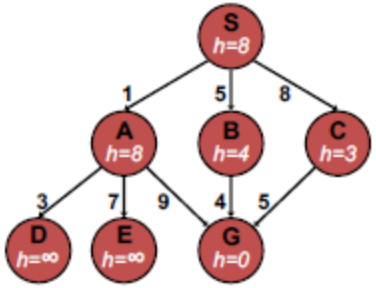
g(n),f(n), h(n) and h*(n) calculation in A*

n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	
E	8	∞	∞	
G	10/9/13	0	10/9/13	



$h^*(n)$ = true cost of minimum-cost path from n to a goal

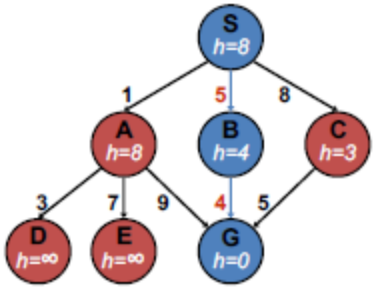
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	∞
E	8	∞	∞	∞
G	10/9/13	0	10/9/13	0



$h^*(n)$ = true cost of minimum-cost path from n to a goal

Example

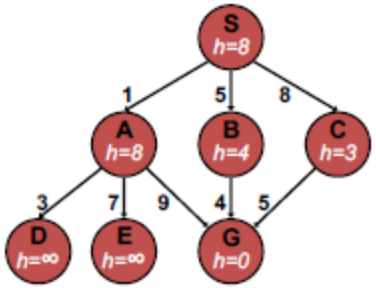
n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	∞
E	8	∞	∞	∞
G	10/9/13	0	10/9/13	0



optimal path = S,B,G
cost = 9

Example

n	$g(n)$	$h(n)$	$f(n)$	$h^*(n)$
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	∞
E	8	∞	∞	∞
G	10/9/13	0	10/9/13	0



Since $h(n) \leq h^*(n)$ for all n, h is admissible

A* Search solution→ Another Example

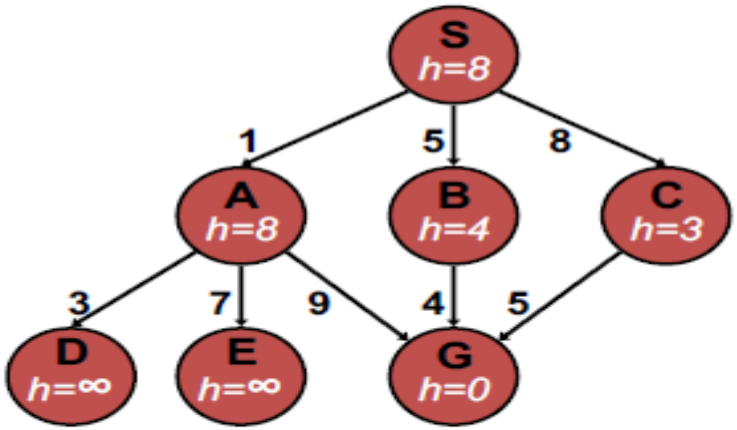
A* Search

Is *h* admissible and/or consistent?

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

# of nodes tested: 0, expanded: 0

expnd. node	Frontier
	{S:0+8}



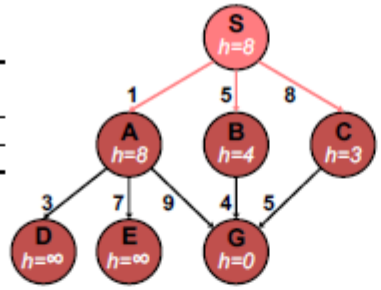
h is consistent since $h(S) - h(A) = 8 - 8 \leq 1$, etc.
and therefore is also admissible

A* Search→ Note the change in color as the node is expanded

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

# of nodes tested: 1, expanded: 1

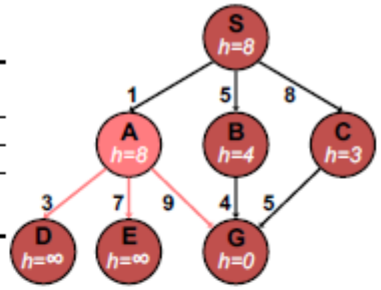
expnd. node	Frontier
	{S:8}
S not goal	{A:1+8,B:5+4,C:8+3}



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

# of nodes tested: 2, expanded: 2

expnd. node	Frontier
	{S:8}
S	{A:9,B:9,C:11}
A not goal	{B:9,G:1+9+0,C:11, D:1+3+∞,E:1+7+∞}

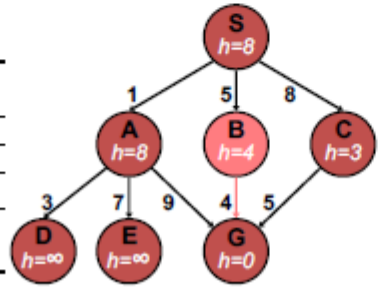


A* Search

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

# of nodes tested: 3, expanded: 3

expnd. node	Frontier
	{S:8}
S	{A:9,B:9,C:11}
A	{B:9,G:10,C:11,D:∞,E:∞}
B not goal	{G:5+4+0,~~G:10~~,C:11, D:∞,E:∞} replace

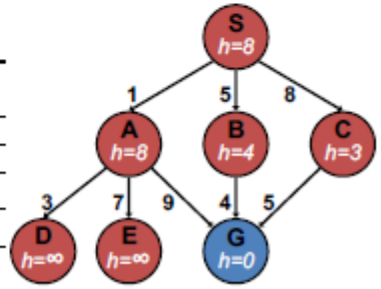


A* Search

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

# of nodes tested: 4, expanded: 3

expnd. node	Frontier
	{S:8}
S	{A:9,B:9,C:11}
A	{B:9,G:10,C:11,D:∞,E:∞}
B	{G:9,C:11,D:∞,E:∞}
G goal	{C:11,D:∞,E:∞} not expanded

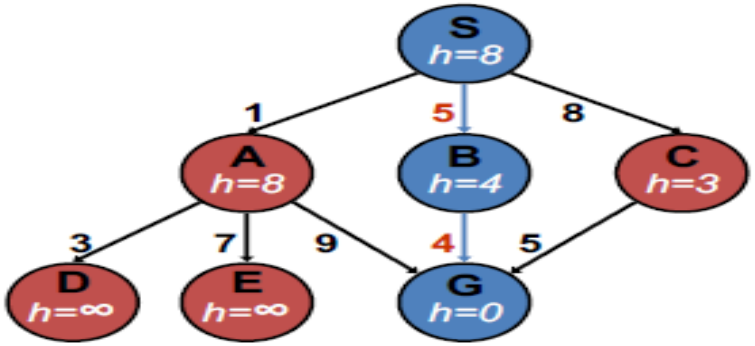


A* Search

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

# of nodes tested: 4, expanded: 3

expnd. node	Frontier
	{S:8}
S	{A:9,B:9,C:11}
A	{B:9,G:10,C:11,D:∞,E:∞}
B	{G:9,C:11,D:∞,E:∞}
G	{C:11,D:∞,E:∞}



- *Pretty fast and optimal*

path: S,B,G
cost: 9

A* search, evaluation

- **Completeness: YES**
- **Time complexity: (exponential with path length)**
- **Space complexity:(all nodes are stored)**
- **Optimality: YES**
  - Cannot expand f_{i+1} until f_i is finished.
  - A* expands all nodes with $f(n) < f(G)$
  - A* expands one node with $f(n) = f(G)$
  - A* expands no nodes with $f(n) > f(G)$

Also *optimally efficient* (not including ties)

A* Search evaluation

- Time complexity of A* depends on the quality of the heuristic function.
- In a worst-case, the algorithm time complexity can be *O(b^d)*, where *b* is the branching factor – the average number of edges from each node, and *d* is the number of nodes on the resulting path.
- **The better the heuristic function, the less of these nodes need to be visited, and so the complexity drops.**
- If we assume heuristic function as the effective branching factor – the average number of edges from each node that we need to visit.
- Similarly **the space complexity of it is exponential** *O(b^d)*, **; keeps all nodes in Memory in the worst case scenario.**

Summary of informed search

- How to use a heuristic function to improve search
  - Greedy Best-first search + Uniform-cost search = A* Search

- When is A* search optimal?
  - h is Admissible (optimistic) for Tree Search
  - h is Consistent for Graph Search

- Choosing heuristic functions
  - A good heuristic function can reduce time/space cost of search by orders of magnitude.
  - Good heuristic function may take longer to evaluate.

Informed Search II

1. When A* fails – Hill climbing, simulated annealing
2. Genetic algorithms

- Local Search: Hill Climbing
- Escaping Local Maxima: Simulated Annealing
- Genetic Algorithms

Local search and optimization

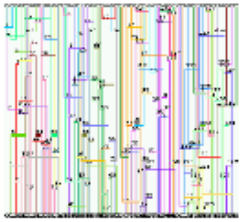
- **Local search:**
  - Use single current state and move to neighboring states.
- **Idea: start with an initial guess at a solution and incrementally improve it until it is one**
- **Advantages:**
  - Use very little memory
  - Find often *reasonable* solutions in large or infinite state spaces.
- **Useful for pure optimization problems.**
  - Find or approximate best state according to some *objective function*
  - *Optimal if the space to be searched is convex*

Solving Optimization Problems using Local Search Methods

Now a different setting:

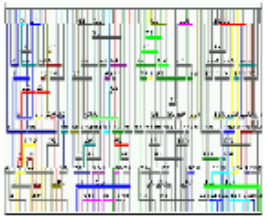
- Each state *s* has a **score** or **cost**, *f*(*s*), that we can compute
- The goal is to find the state with the highest (or lowest) score, or a reasonably high (low) score
- We do ***not*** care about the path
- Use *variable-based models*
  - Solution is not a path but an assignment of values for a set of variables
- **Enumerating all the states is intractable**
- Previous search algorithms are too expensive

Example Problem: Chip Layout



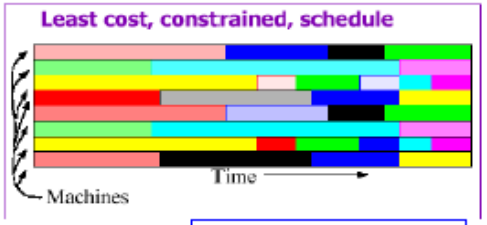
Lots of Chip Real Estate

Channel Routing



Same connectivity, much less space

Example Problem: Scheduling



Also:

parking lot layout, product design, aero-dynamic design, "Million Queens" problem, radiotherapy treatment planning, …

Local Searching

- Hard problems can be solved in polynomial time by using either an:
  - approximate model: find an exact solution to a simpler version of the problem
  - approximate solution: find a non-optimal solution to the original hard problem
- We'll explore ways to search through a solution space by **iteratively improving solutions** until one is found that is optimal or near optimal

Local Searching

- Local searching: **every node is a solution**
  - **Operators/actions go from one solution to another**
  - **can stop at any time and have a valid solution**
  - goal of search is to find a ***better/best*** solution
- No longer searching a state space for a solution path and then executing the steps of the solution path
- A* *isn't* a local search since it considers different partial solutions by looking at the estimated cost of a *solution path*

Local Searching

- An ***operator/action*** is needed to transform one solution to another

Informal Characterization

These are problems in which

- There is some combinatorial structure being optimized
- There is a cost function: Structure → Real number, to be optimized, or at least a reasonable solution is to be found
- Searching all possible structures is intractable
- There's no known algorithm for finding the optimal solution efficiently
- "Similar" solutions have similar costs

Local Searching

- Those solutions that can be reached with *one* application of an operator are in the current solution's ***neighborhood*** (aka "***move set***")
- **Local search** considers next only those solutions in the neighborhood
- The neighborhood should be much smaller than the size of the search space (otherwise the search degenerates)

Local Searching

- An evaluation function, f, is used to map each solution/state to a number corresponding to the *quality/cost* of that solution
- TSP: Use the length of the tour;
  A better solution has a shorter tour length
- Maximize f:
  called **hill-climbing** (gradient ascent if continuous)
- Minimize f:
  called or **valley-finding** (gradient descent if continuous)
- Can be used to maximize/minimize some cost

Hill-Climbing (HC)

- **Question**: What's a neighbor?
  - Problem spaces tend to have structure. A *small change* produces a neighboring state
  - The size of the neighborhood must be small enough for efficiency
  - Designing the neighborhood is critical; This is the real ingenuity – not the decision to use hill-climbing
- **Question**: Pick which neighbor?
  - The best one (greedy)
- **Question**: What if no neighbor is better than the current state? Stop

Hill-climbing search

I. **While (∃ uphill points):**
   - Move in the direction of increasing evaluation function ***f***

II. **Let** $s_{next} = \arg\max_{s} f(s)$ **, s a successor state to the current state n**
   - If $f(n) < f(s)$ then move to s
   - Otherwise halt at n

- **Properties:**
  - Terminates when a peak is reached.
  - Does not look ahead of the immediate neighbors of the current state.
  - Chooses randomly among the set of best successors, if there is more than one.
  - Doesn't *backtrack*, since it doesn't remember where it's been
- **a.k.a. *greedy local search***

"Like climbing Everest in thick fog with amnesia"

Hill-Climbing Algorithm

1. Pick initial state *s*
2. Pick *t* in neighbors(*s*) with the largest *f*(*t*)
3. **if** *f*(*t*) ≤ *f*(*s*) **then** stop and **return** *s*
4. *s* = *t*. **Goto** Step 2.

- Simple
- Greedy
- **Stops at a *local* maximum**

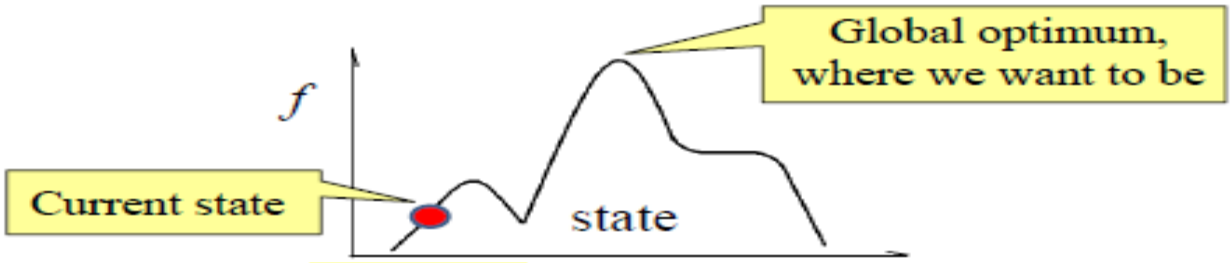
- HC *exploits* the neighborhood
  - like Greedy Best-First search, it chooses what looks best *locally*
  - but doesn't allow backtracking or jumping to an alternative path since there is no *Frontier* list
- HC is *very* space efficient
  - Similar to **Beam Search** with a beam width of 1

- HC is very fast and often effective in practice

Hill Climbing →Iteration may stop at local optima which isn't desired

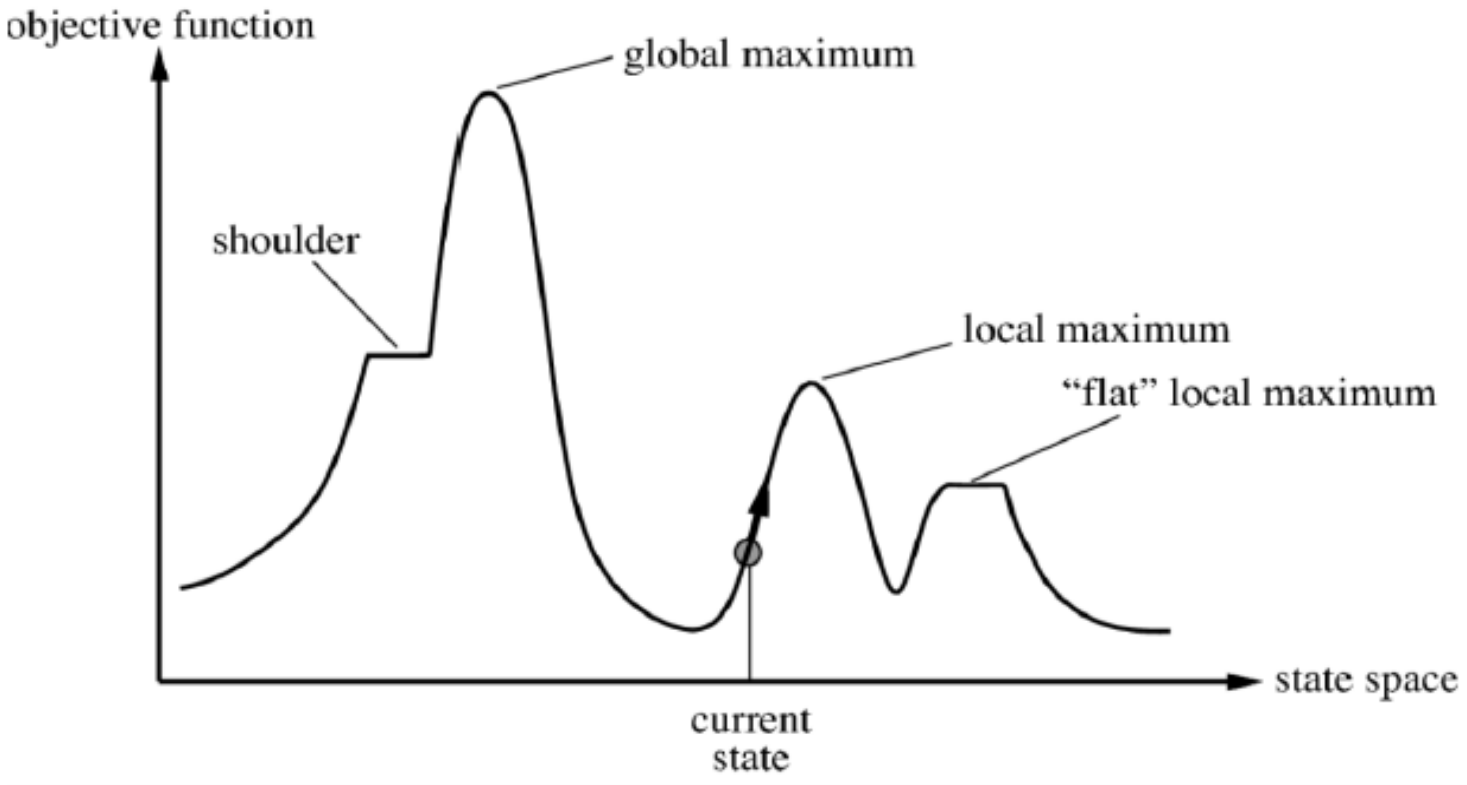
Local Optima in Hill-Climbing

- Useful mental picture: f is a surface ("hills") in state space



- But we can't see the entire landscape all at once. Can only see a neighborhood; like climbing in fog

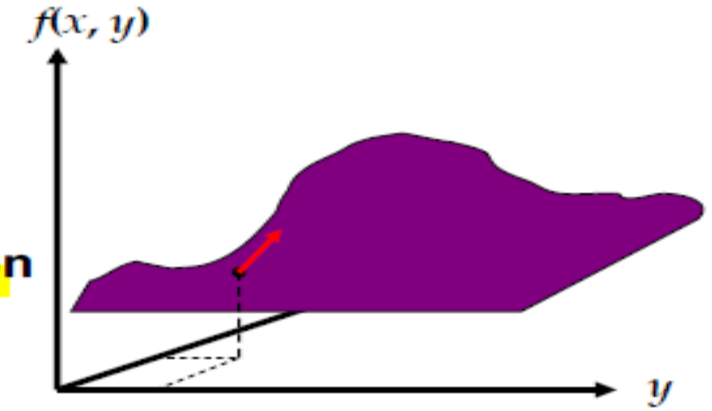
Search Space features



Hill-Climbing

Visualized as a 2D surface

- **Height is quality/cost of solution** $f = f(x, y)$

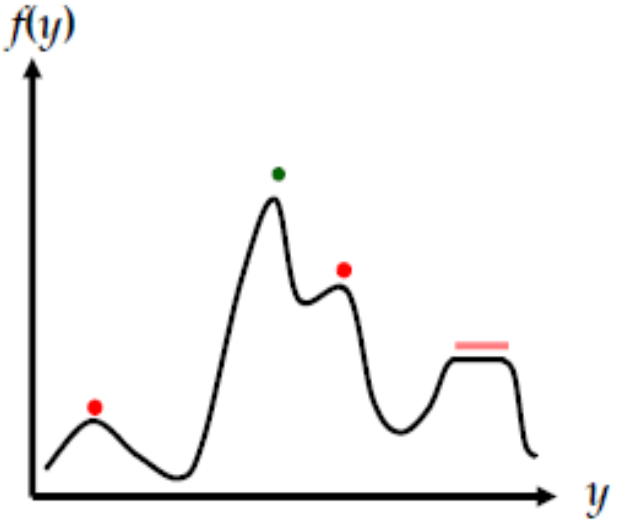


- **Solution space is a 2D surface**
- **Initial solution is a point**
- **Goal is to find highest point on the surface of solution space**
- **Hill-Climbing follows the *direction of the steepest ascent*, i.e., where *f* increases the most**

Hill-Climbing (HC)

Solution found by HC is totally determined by the starting point; its fundamental weakness **is** getting stuck:

- **At a local maximum**
- **At plateaus and ridges**

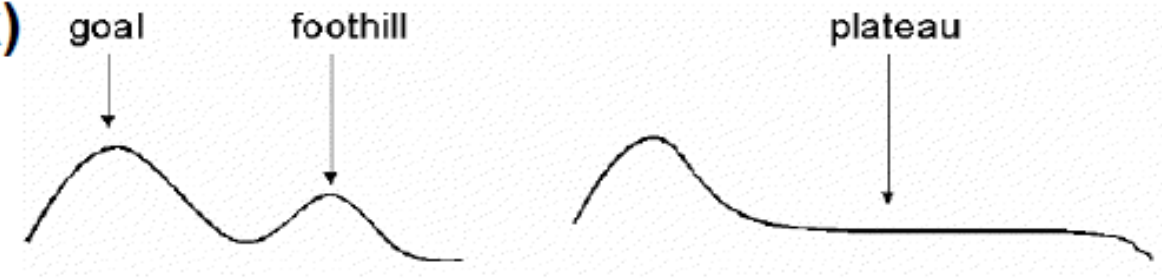


Global maximum may *not* be found

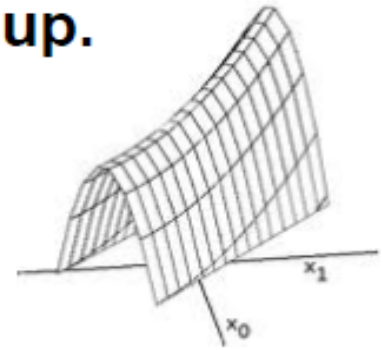
Trade off:
greedily exploiting locality as in HC
vs. exploring state space as in BFS

Drawbacks of hill climbing

- Local Maxima: peaks that aren't the highest point in the space
- Plateaus: the space has a broad flat region that gives the search algorithm no direction (random walk)



- Ridges: dropoffs to the sides; steps to the North, East, South and West may go down, but a step to the NW may go up.



Hill-Climbing with Random Restarts

Very simple modification:

1. When stuck, pick a *random* new starting state and re-run hill-climbing from there
2. Repeat this *k* times
3. Return the best of the *k* local optima found

- Can be very effective
- Should be tried whenever hill-climbing is used
- Fast, easy to implement; works well for many applications where the solution space surface is not too "bumpy" (i.e., not too many local maxima)

One Remedy to Drawbacks of Hill Climbing: *Random Restart*

- In the end: Some problem spaces are great for hill climbing and others are terrible.

One solution for hill climbing being trapped at local maxima

Simulated annealing (SA)

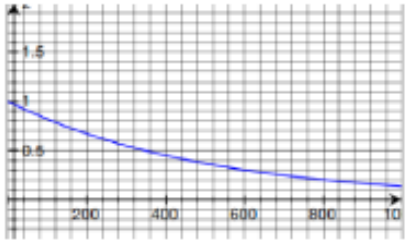
- *Annealing*: the process by which a metal cools and freezes into a minimum-energy crystalline structure (the annealing process)
- Conceptually SA exploits an analogy between annealing and the search for a minimum in a more general system.
  - AIMA: Switch viewpoint from *hill-climbing* to *gradient descent*
  - *(But: AIMA algorithm hill-climbs & larger ΔE is good…)*
- SA hill-climbing can avoid becoming trapped at local maxima.
- SA uses a random search that occasionally accepts changes that decrease objective function *f*.
- SA uses a control parameter T, which by analogy with the original application is known as the system *"temperature."*
- T starts out high and gradually decreases toward 0.

How Simulated annealing strategy is used in hill climbing

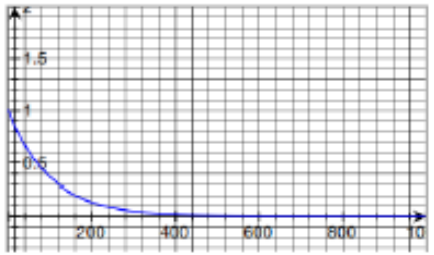
Simulated annealing (cont.)

- A "bad" move from A to B (f(B)<f(A)) is accepted with the probability

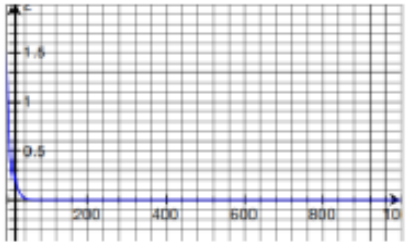
$$P(\text{move}_{A\to B}) = e^{(f(B)-f(A))\ /\ T}$$



$\exp\left(-\frac{x}{500}\right)$



$\exp\left(-\frac{x}{100}\right)$

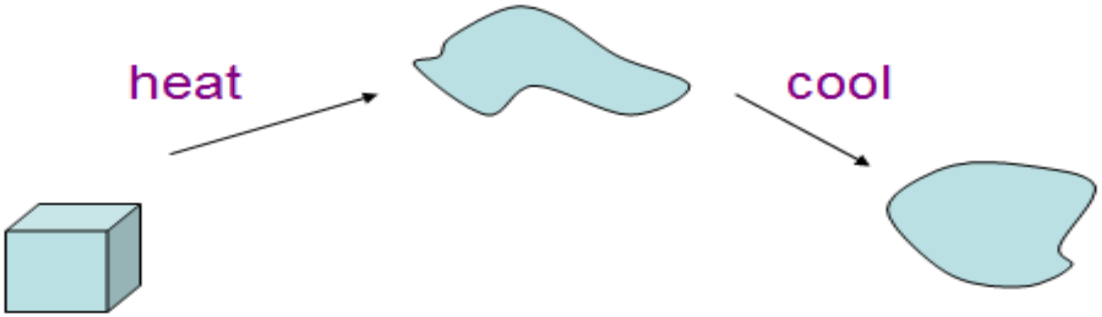


$\exp\left(-\frac{x}{10}\right)$

- The higher T, the more likely a bad move will be made.
- As T tends to zero, this probability tends to zero, and SA becomes more like hill climbing
- *If T is lowered slowly enough, SA is complete and admissible.*

Simulated annealing

- Comes from the physical process of annealing in which substances are raised to high energy levels (melted) and then cooled to solid state.



- The probability of moving to a higher energy state, instead of lower is

  $p = e^{(-\Delta E/kT)}$

  where ΔE is the positive change in energy level, T is the temperature, and k is Bolzmann's constant.

- At the beginning, the temperature is high.
- As the temperature becomes lower
  - kT becomes lower
  - $\Delta E/kT$ gets bigger
  - $(-\Delta E/kT)$ gets smaller
  - $e^{(-\Delta E/kT)}$ gets smaller
- As the process continues, the probability of a downhill move gets smaller and smaller.

For Simulated Annealing

- ΔE represents the change in the value of the objective function.
- Since the physical relationships no longer apply, drop k. So p = e^(-ΔE/T)
- We need an annealing schedule, which is a sequence of values of T: T_0, T_1, T_2, …

SA algorithm

- *current* ← start node;
- for each T on the schedule /* need a schedule */
  - *next* ← randomly selected successor of *current*
  - evaluate next; it it's a goal, return it
  - ΔE ← *next*.Value – *current*.Value /* already negated */
  - if ΔE > 0
    - then *current* ← *next* /* better than current */
    - else *current* ← *next* with probability e^(ΔE/T)

How would you do this probabilistic selection?

Probabilistic Selection

- Select *next* with probability p

- Generate a random number
- If it's <= p, select *next*

Simulated Annealing

Applicability

- **Discrete Problems where state changes are *transforms* of local parts of the configuration**
  - E.G. Travelling Salesman problem, where moves are swaps of the order of two cities visited:
    - —Pick an initial tour randomly
    - —Successors are all neighboring tours, reached by swapping adjacent cities in the original tour
    - —Search using simulated annealing..

Time and space complexity of Hill climbing → It is **exactly the same as DFS** – the only difference is the *order* that nodes are expanded in. **That doesn't change the time or space complexity in the worst case (though in the average case, the whole idea of a heuristic is to ensure that we get to a Goal faster…so, if it's a good heuristic, the average time complexity ought to improve).**
Just like DFS then, it will always find an answer, though not necessarily the one earliest in the search tree.

Game Playing

The Minimax Rule: `Don't play hope chess'

Idea: Make the best move for MAX *assuming that MIN always replies with the best move for MIN*

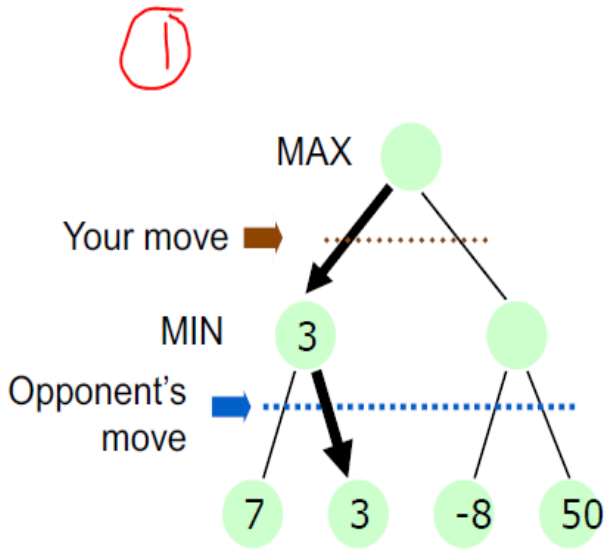
Easily computed by a recursive process

- The ***backed-up value*** of each node in the tree is determined by the values of its children:
  - For a **MAX** node, the backed-up value is the ***maximum*** of the values of its children *(i.e. the best for MAX)*
  - For a **MIN** node, the backed-up value is the ***minimum*** of the values of its children *(i.e. the best for MIN)*

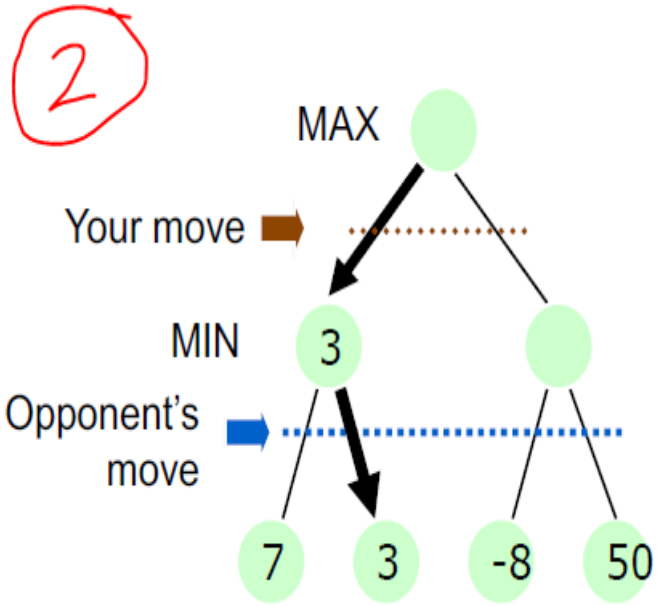
Minimax search

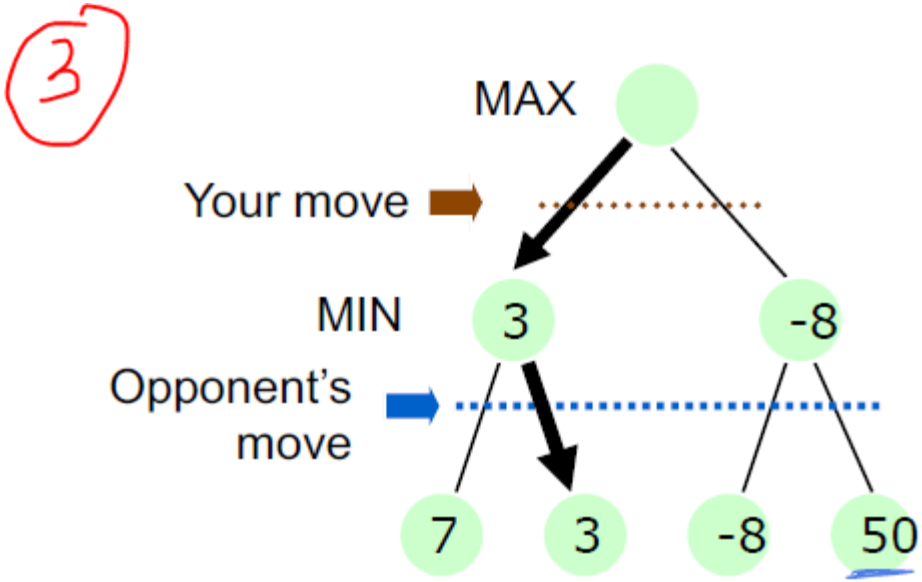
- Assume that both players play perfectly
  - do not assume player will miss good moves or make mistakes
- Score(s): The score that MAX will get towards the end if both player play perfectly from s onwards.
- Consider MIN's strategy
  - MIN's best strategy:
    - choose the move that minimizes the score that will result when MAX chooses the maximizing move
  - MAX does the opposite

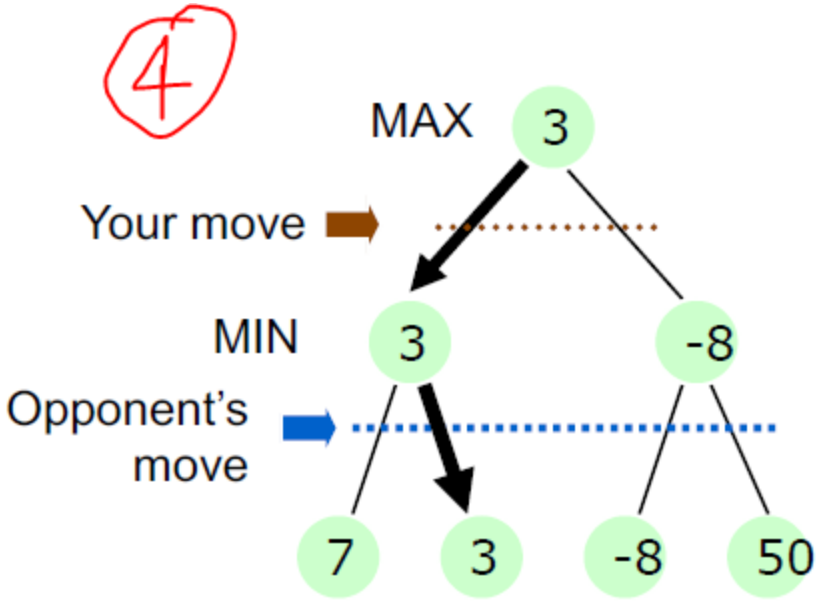
Min maxing



- Your opponent will choose smaller numbers
- If you move left, your opponent will choose 3
- If you move right, your opponent will choose -8
- Thus your choices are only 3 or -8
- You should move left



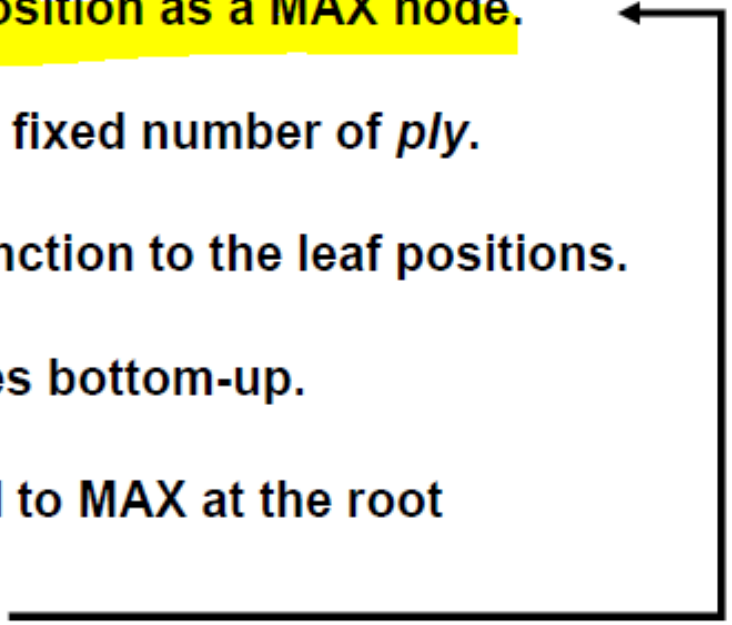




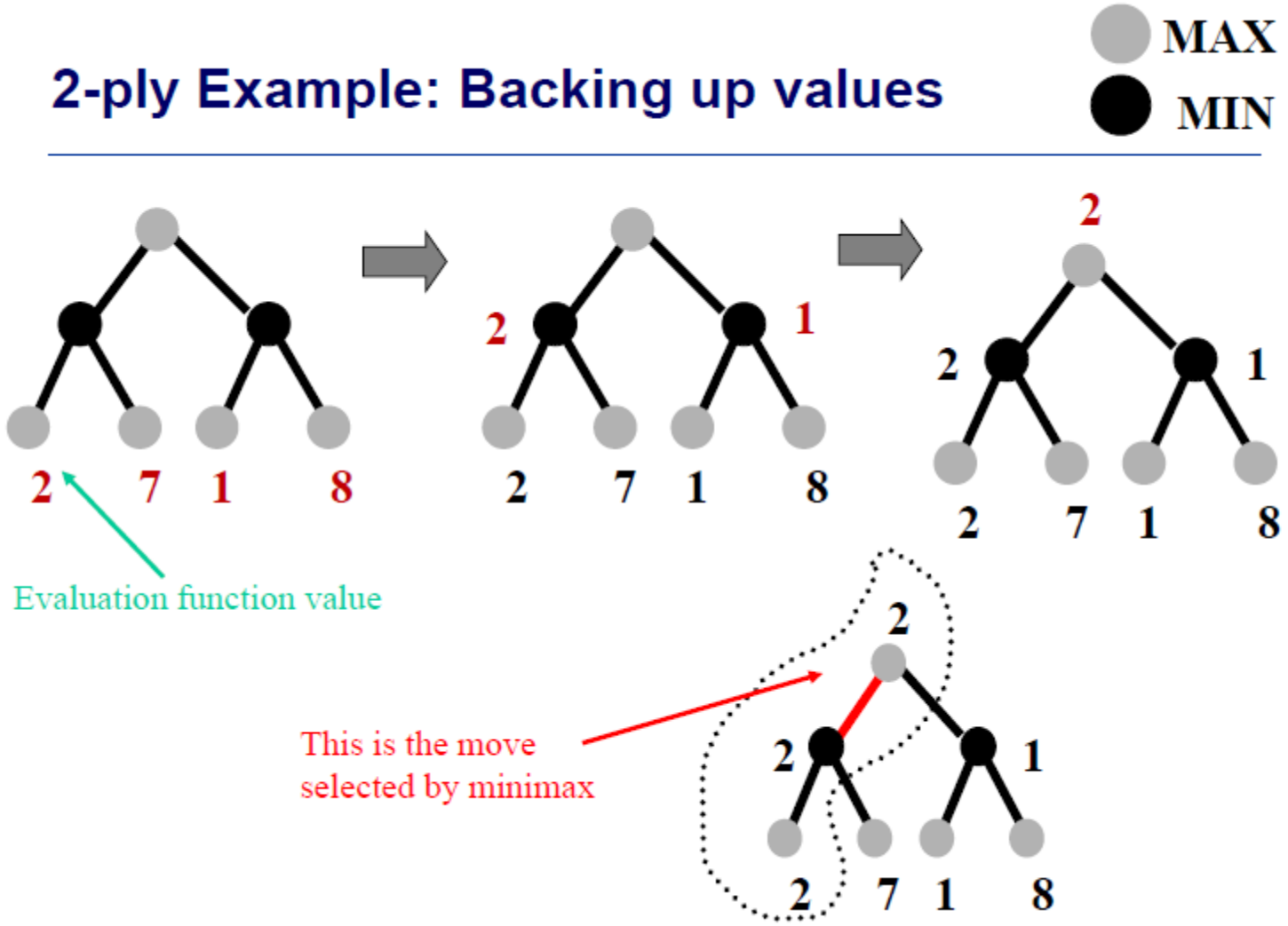
The Minimax Procedure

Until game is over:

1. Start with the current position as a MAX node.
2. Expand the game tree a fixed number of *ply*.
3. Apply the evaluation function to the leaf positions.
4. Calculate back-up values bottom-up.
5. Pick the move assigned to MAX at the root
6. Wait for MIN to respond

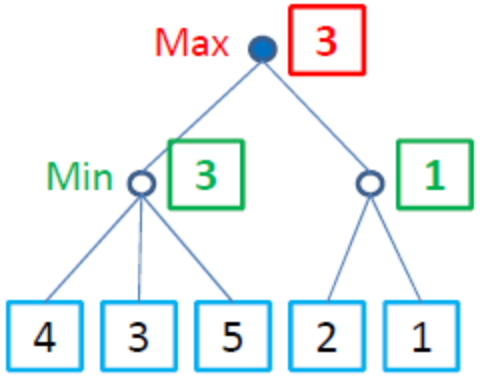


2-ply Example: Backing up values



MiniMax Algorithm

- Restrictions
  - 2 players: Max = Computer & Min = Opponent
  - Deterministic, perfect information
- Depth-bound & Evaluation function
  - Construct tree (depth-bound)
  - Compute evaluation leaves
  - Propagate upwards (min/max)



What if MIN does not play optimally?

- **Definition of optimal play for MAX assumes MIN plays optimally:**
  - *Maximizes worst-case outcome* for MAX.
  - (Classic game theoretic strategy)
- **But if MIN does not play optimally, MAX will do even better. [Theorem-not hard to prove]**

Minimax properties

- Optimal? Yes, against an optimal opponent, **if** the tree is finite
- Complete? Yes, **if** the tree is finite
- Time complexity? Exponential: **O(b^m)**
- Space complexity? Polynomial: **O(bm)**

The minimax algorithm performs a complete depth-first exploration of the game tree.
Where the maximum depth of the tree is m and there are b legal moves at each point.

Comments on Minimax Search

- **Depth-first search with fixed number of ply *m* as the limit.**
  - *O(b^m)* time complexity – *As usual!*
  - *O(bm)* space complexity

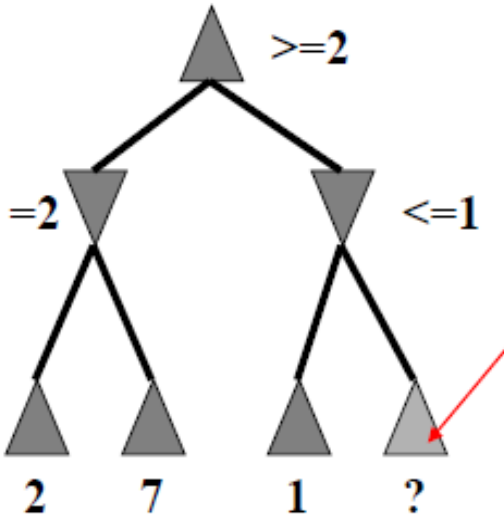
- **Performance will depend on**
  - the quality of the static evaluation function (expert knowledge)
  - depth of search (computing power and search algorithm)

- **Differences from normal state space search**
  - Looking to make *one* move only, despite deeper search
  - No cost on arcs – costs from backed-up static evaluation
  - MAX can't be sure how MIN will respond to his moves

- **Minimax forms the basis for other game tree search algorithms.**

Alpha-Beta Pruning

- A way to improve the performance of the Minimax Procedure
- Basic idea: *"If you have an idea which is surely bad, don't take the time to see how truly awful it is"* ~ Pat Winston



- We don't need to compute the value at this node.
- No matter what it is it can't effect the value of the root node.

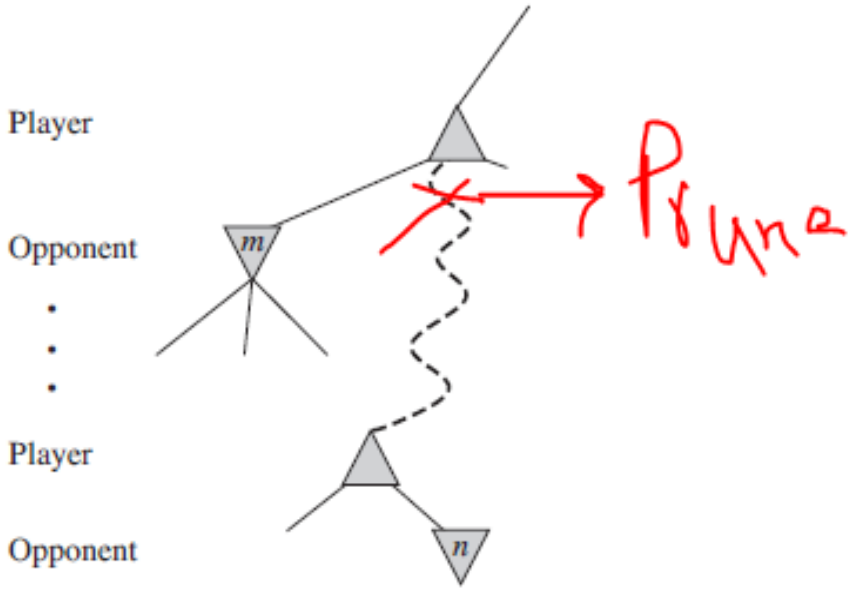


Figure 5.6 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

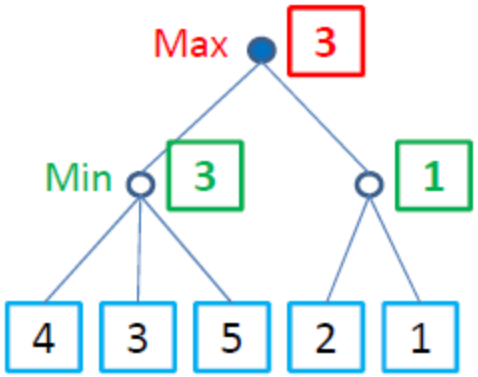
At first max player will start first move generally where $\alpha = -\infty$

$\beta = +\infty$

Alpha-Beta Pruning II

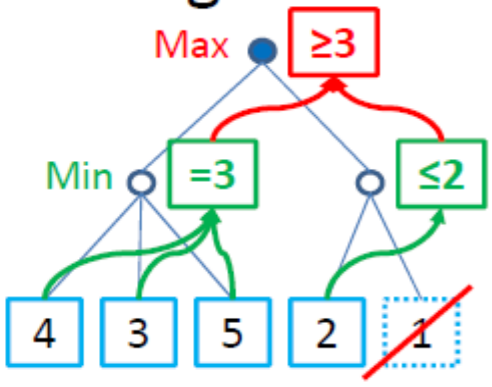
- During Minimax, keep track of two additional values:
  - α: MAX's current *lower* bound on MAX's outcome
  - β: MIN's current *upper* bound on MIN's outcome
- MAX will never allow a move that could lead to a worse score (for MAX) than α
- MIN will never allow a move that could lead to a better score (for MAX) than β
- Therefore, stop evaluating a branch whenever:
  - When evaluating a MAX node: a value $v \geq \beta$ is backed-up
    - —MIN will never select that MAX node
  - When evaluating a MIN node: a value $v \leq \alpha$ is found
    - —MAX will never select that MIN node

Min Max only

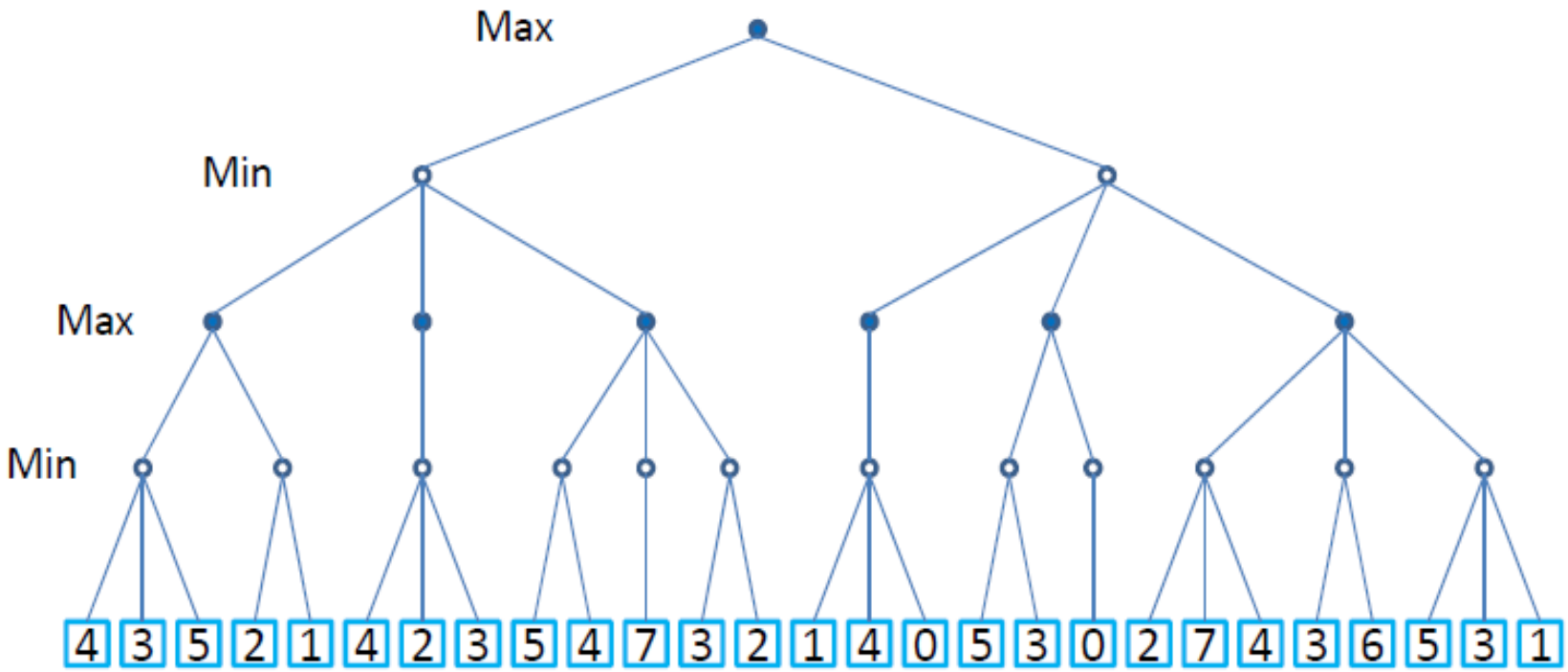


$\alpha\beta$-Pruning

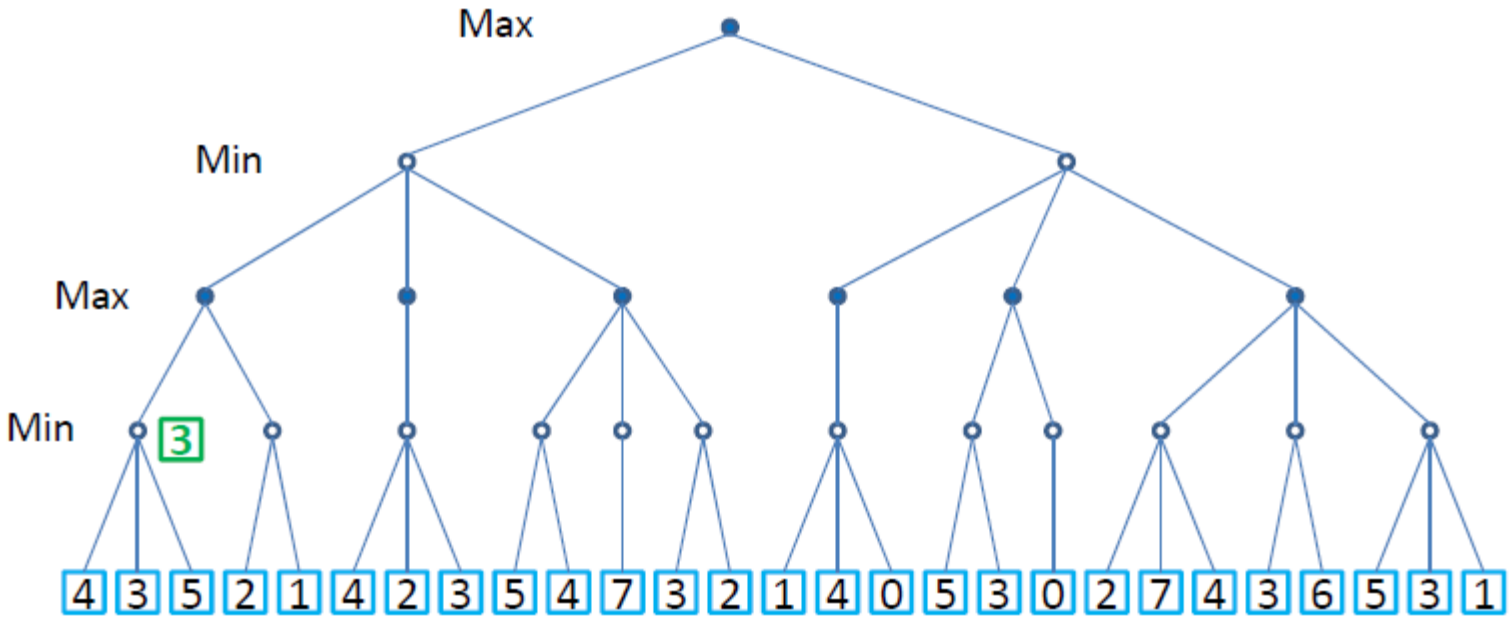
- Generally applied optimization
  - Instead of generating, then propagating
  - Interleave generation and propagation
    - Obtain information on redundant parts
- Generate tree: depth-first & Left-to-right
  - Propagate values of nodes
  - Estimates for parent nodes



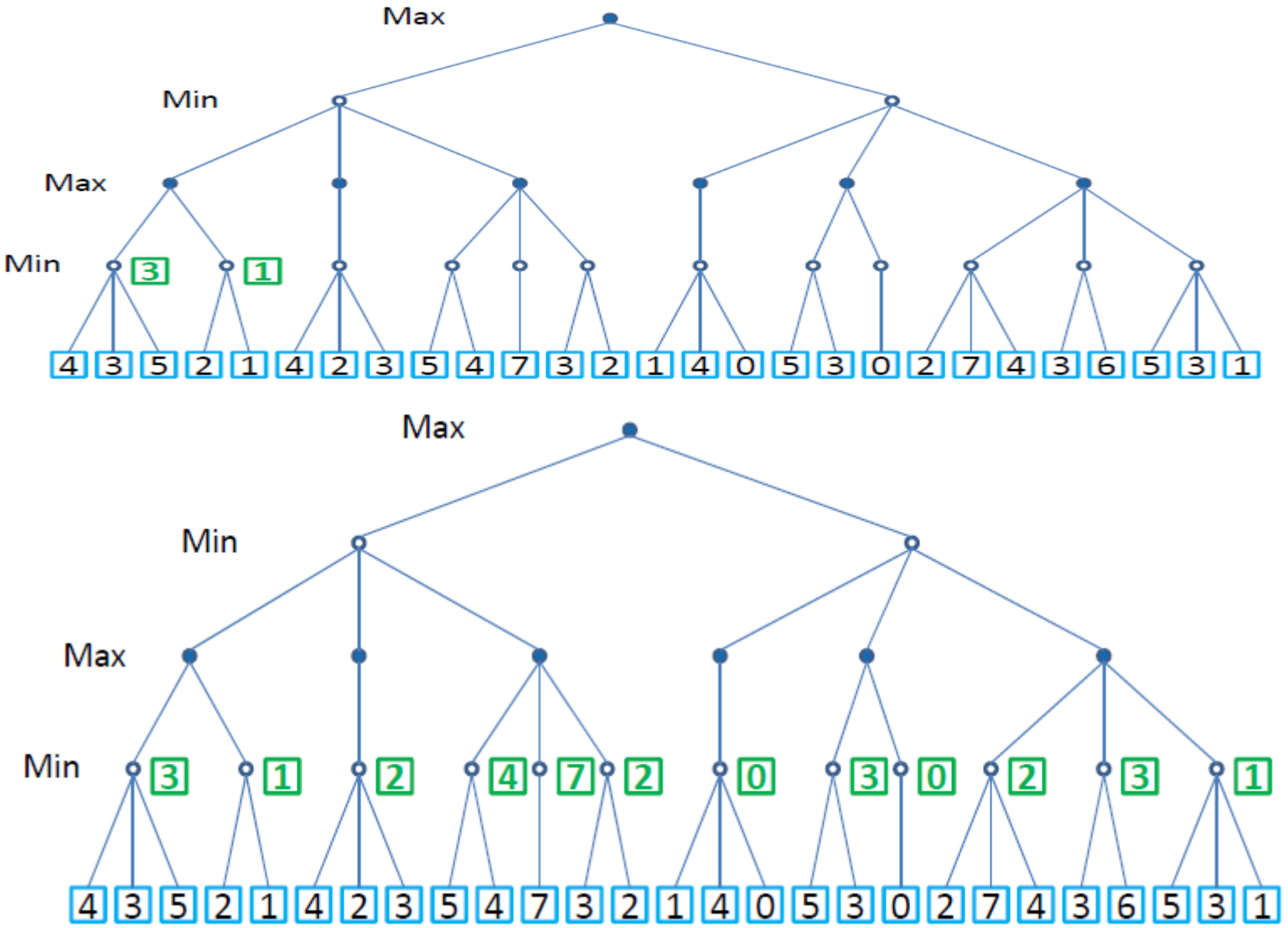
- Perform the minimax algorithm on the figure below. First without, later with $\alpha\beta$-pruning.



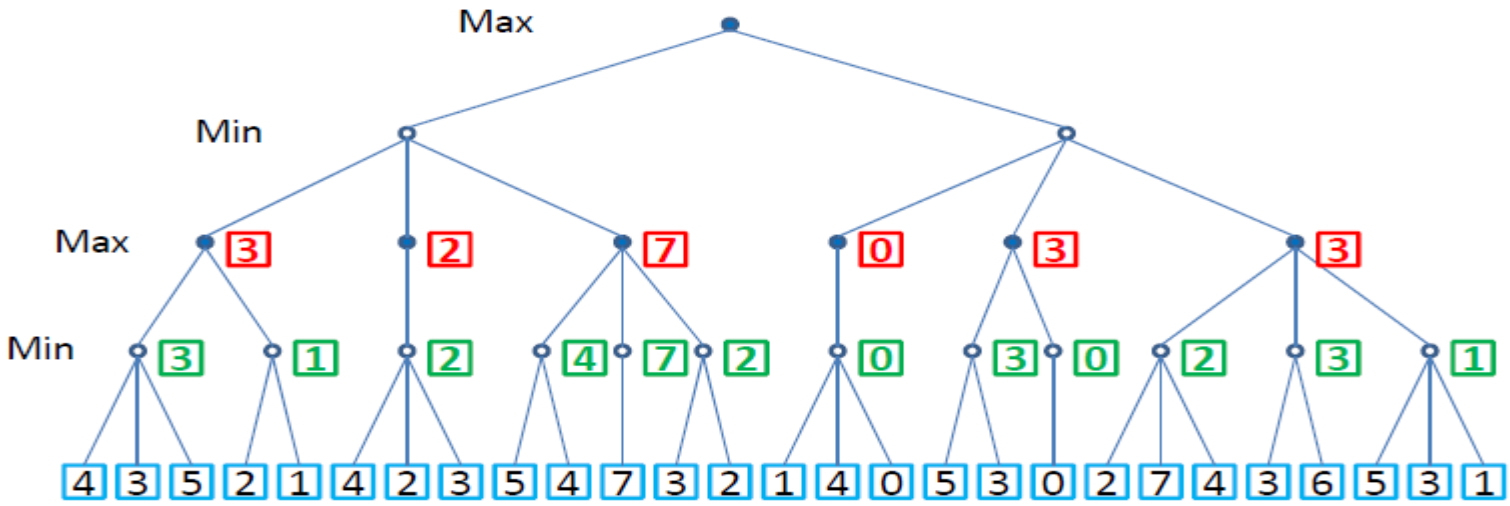
MiniMax without $\alpha\beta$-pruning

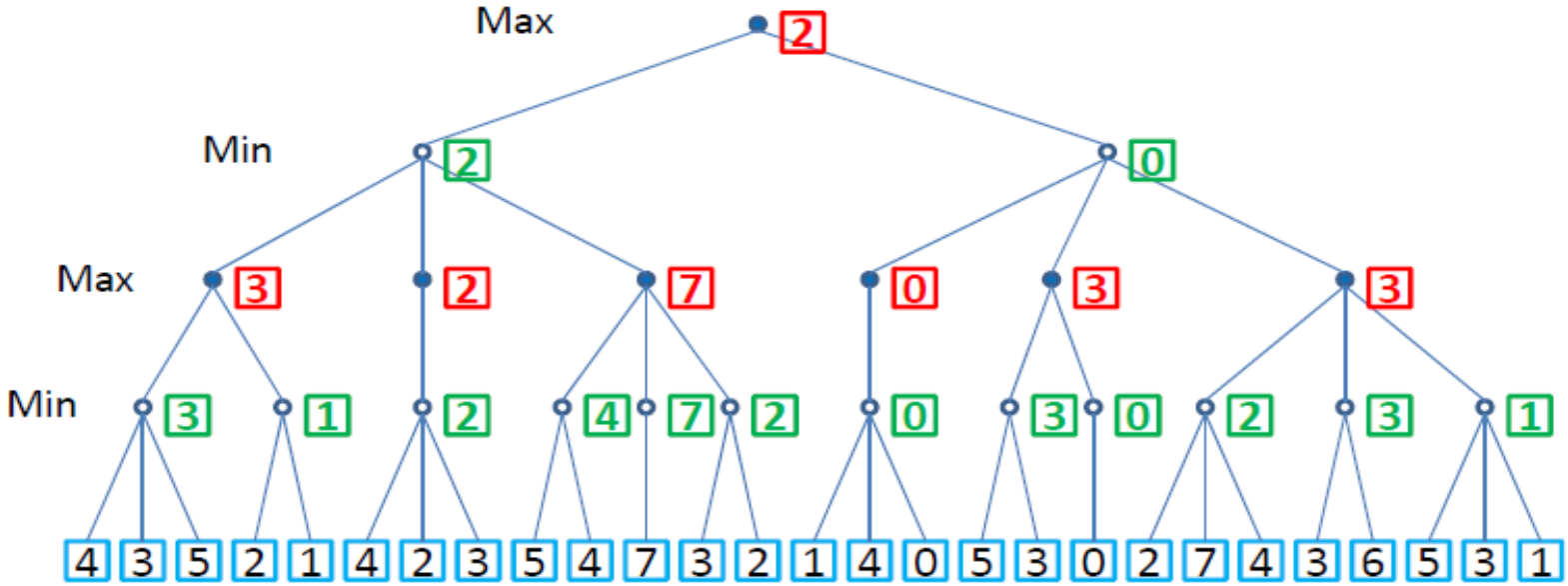


MiniMax without $\alpha\beta$-pruning

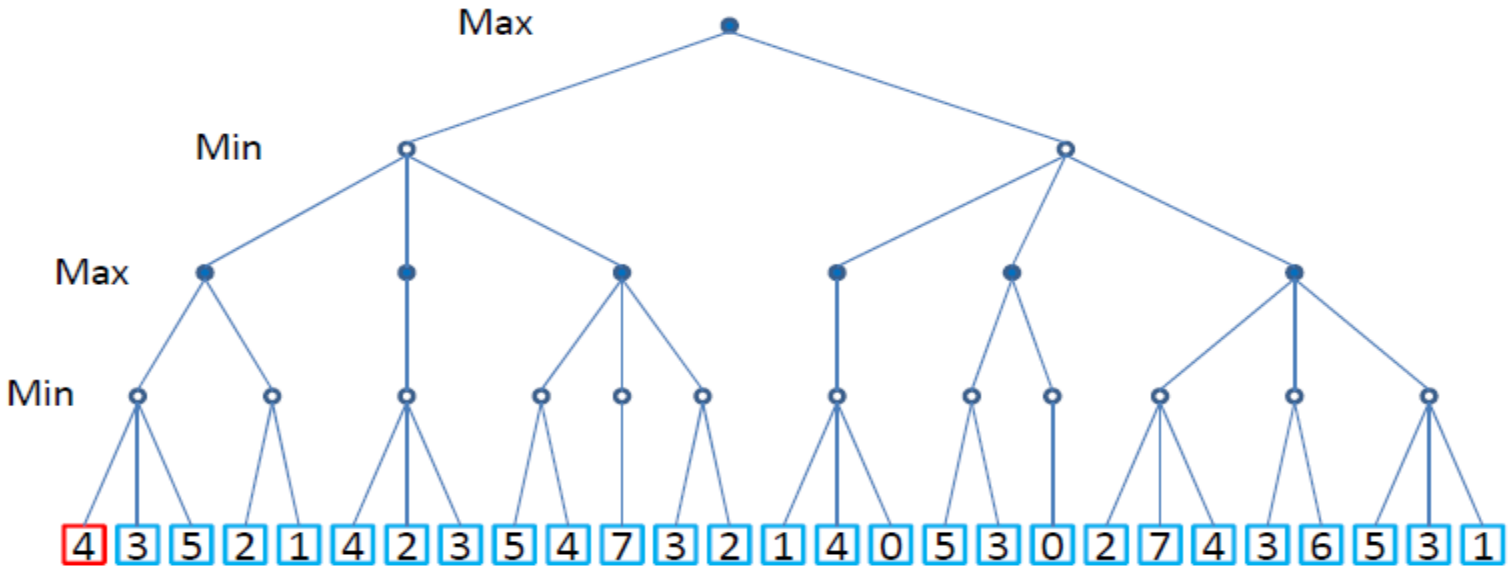


MiniMax without $\alpha\beta$-pruning

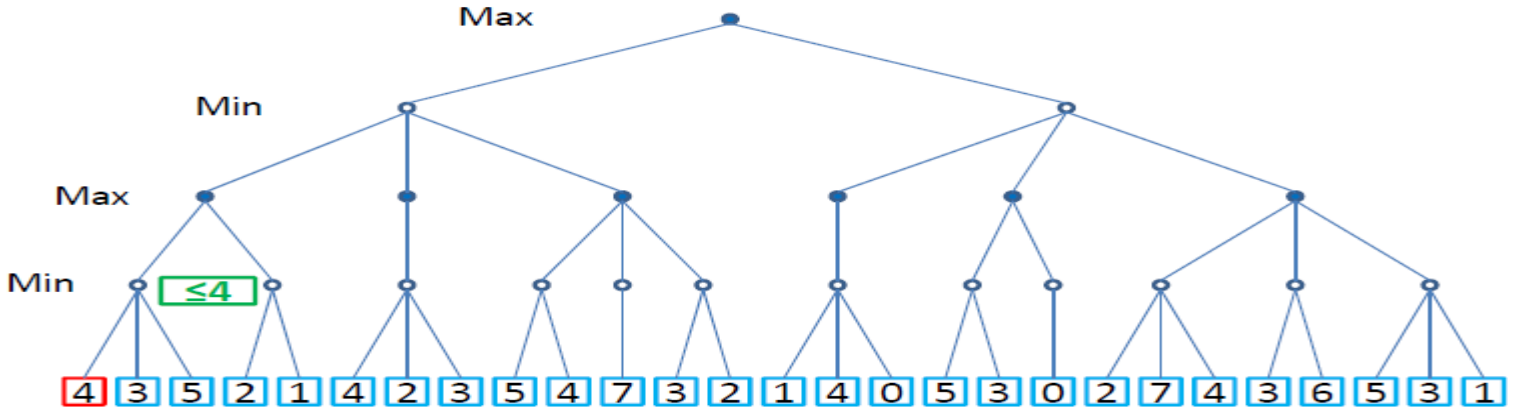




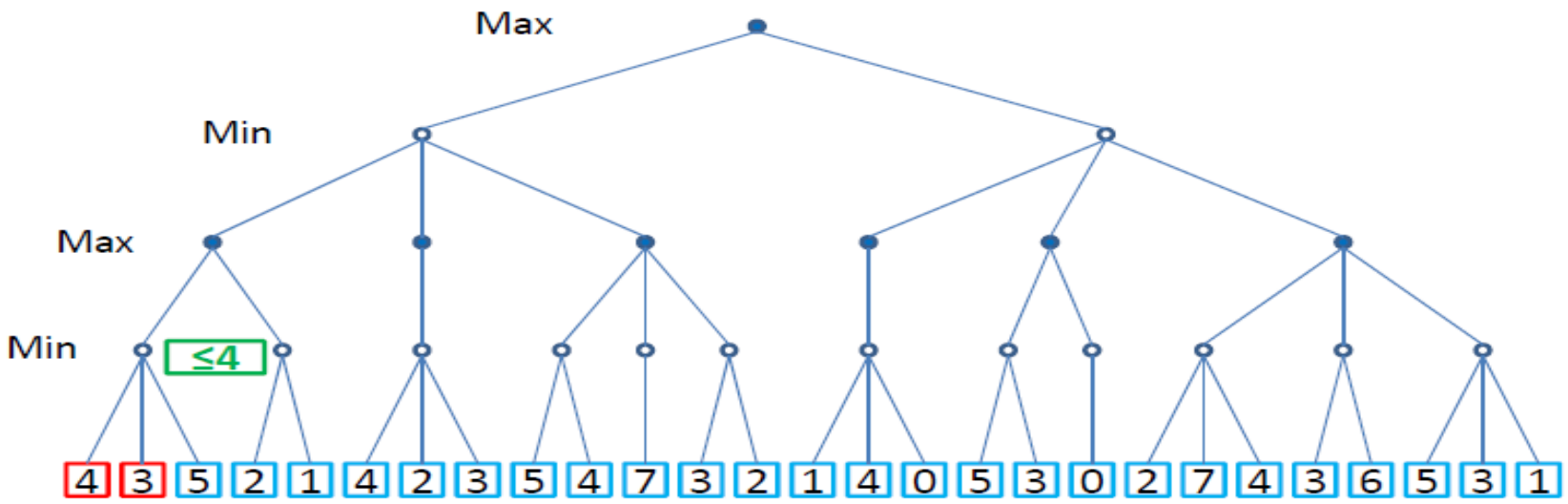
MiniMax with $\alpha\beta$-pruning

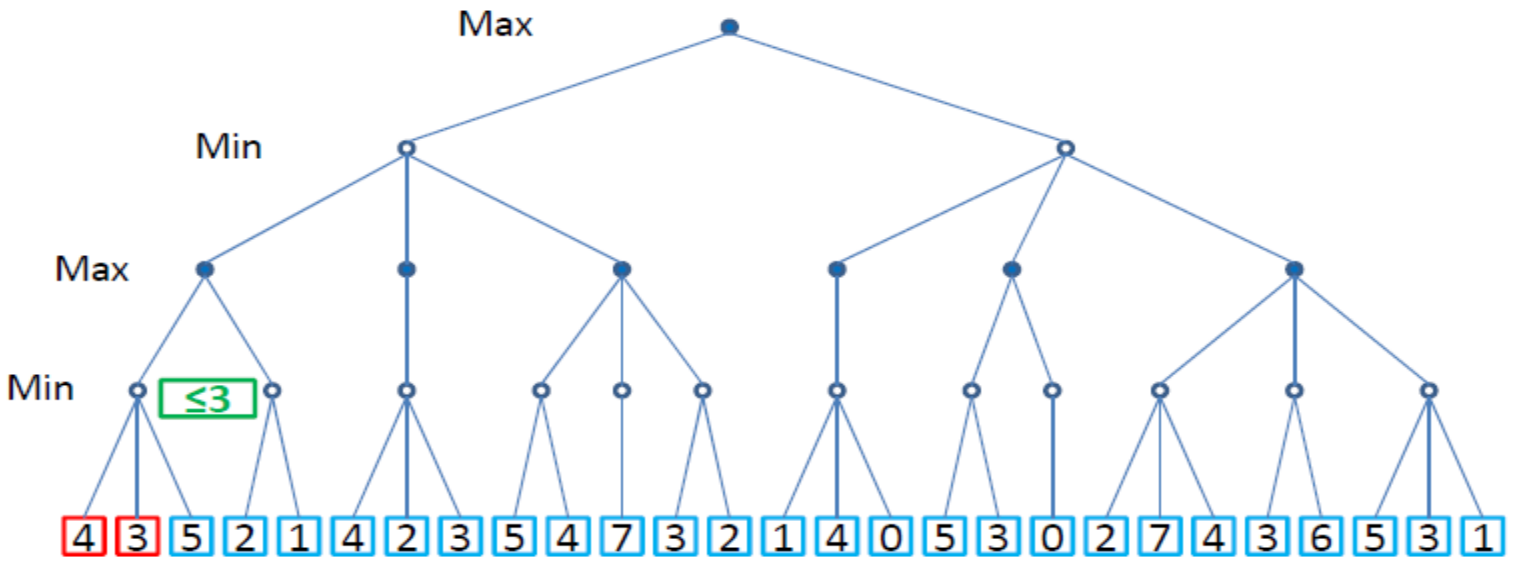


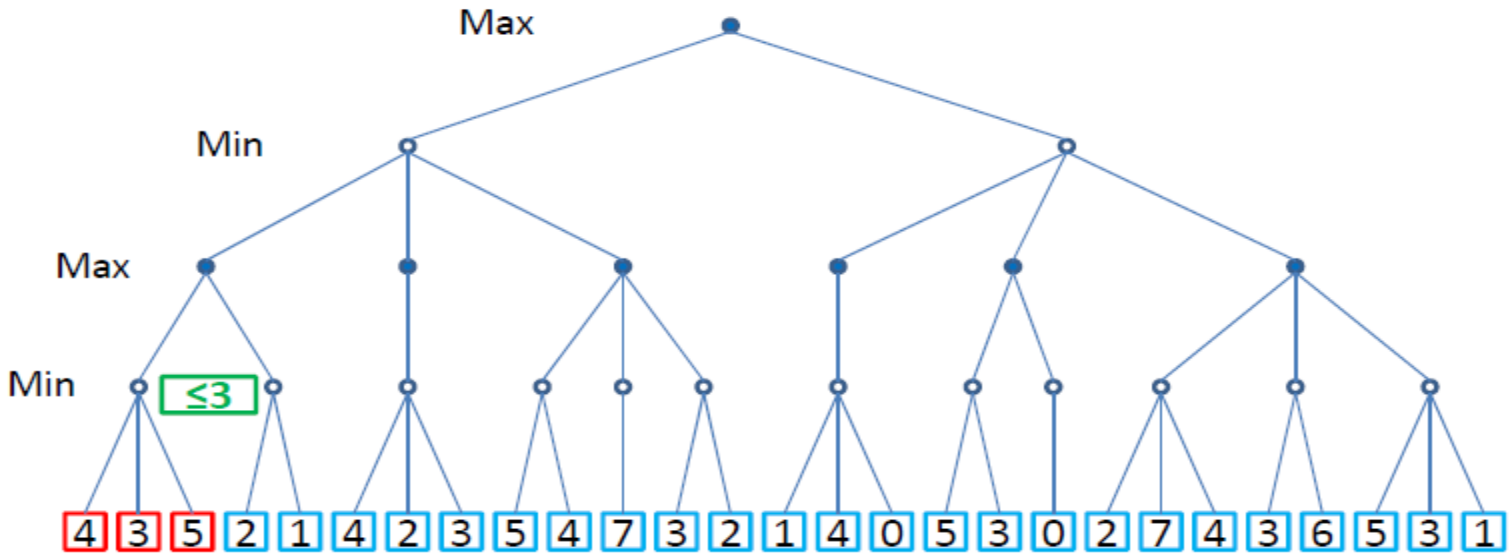
- **α-nodes**: Temporary values at MIN-nodes

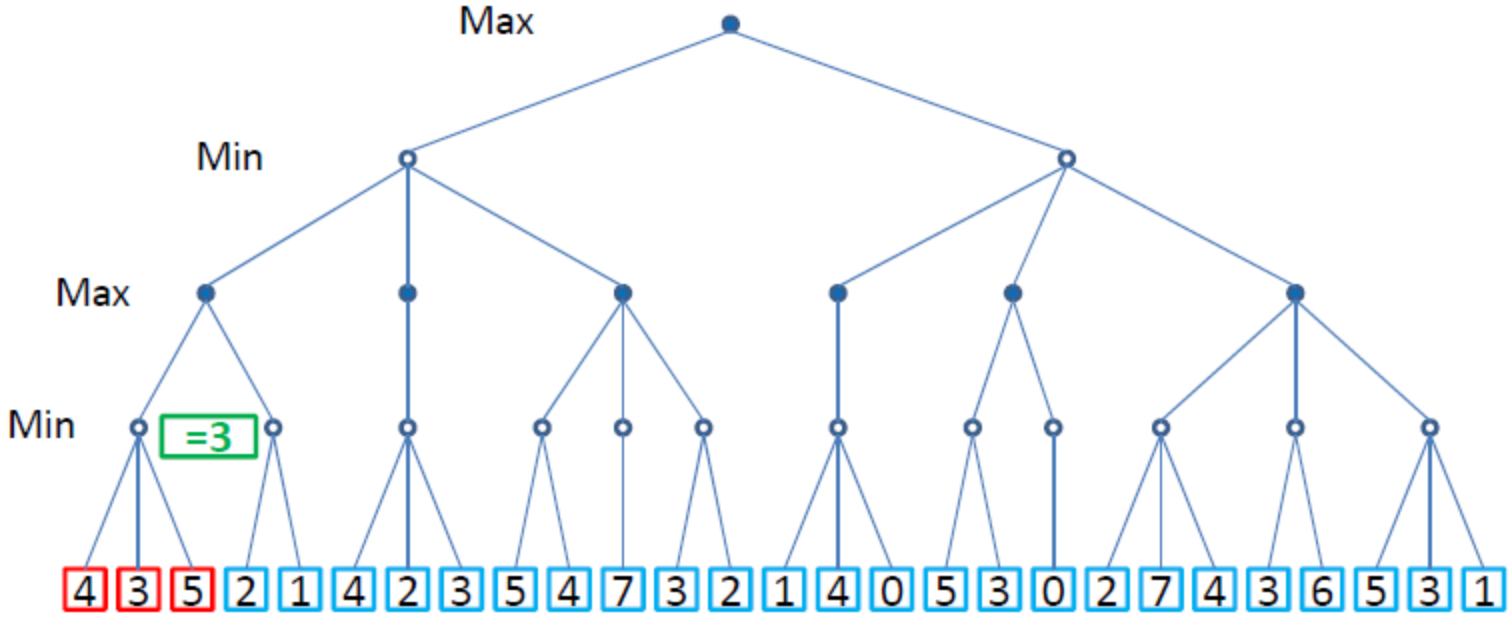


MiniMax with $\alpha\beta$-pruning

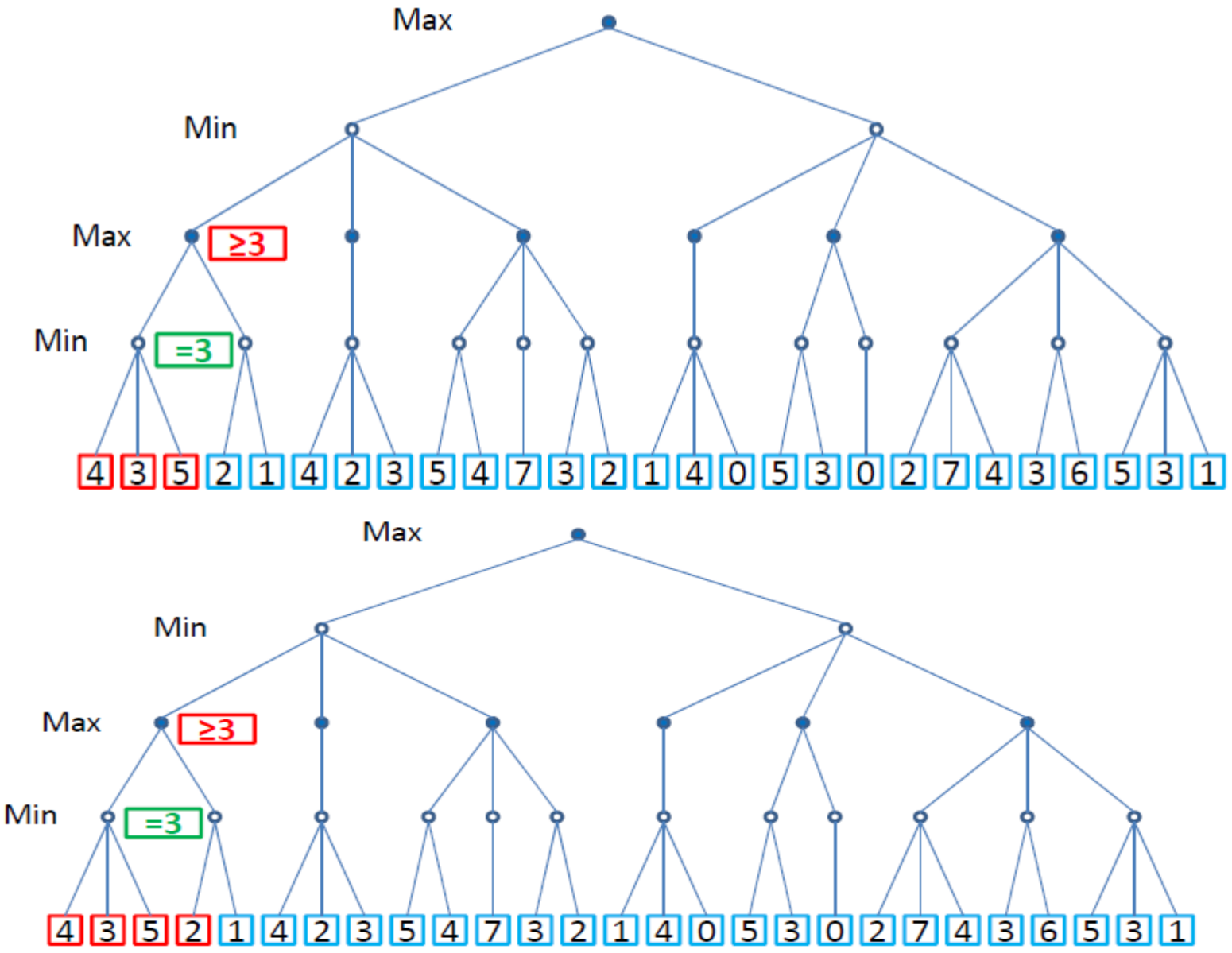


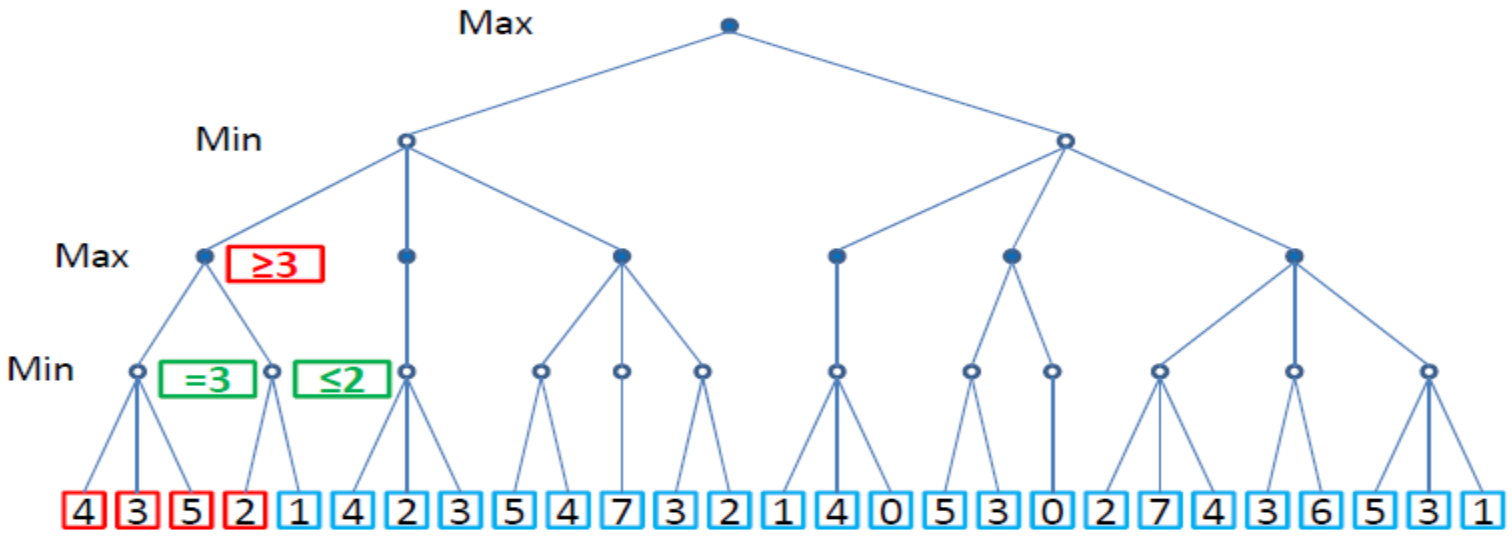



Max
Min
Max
Min
≤3
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

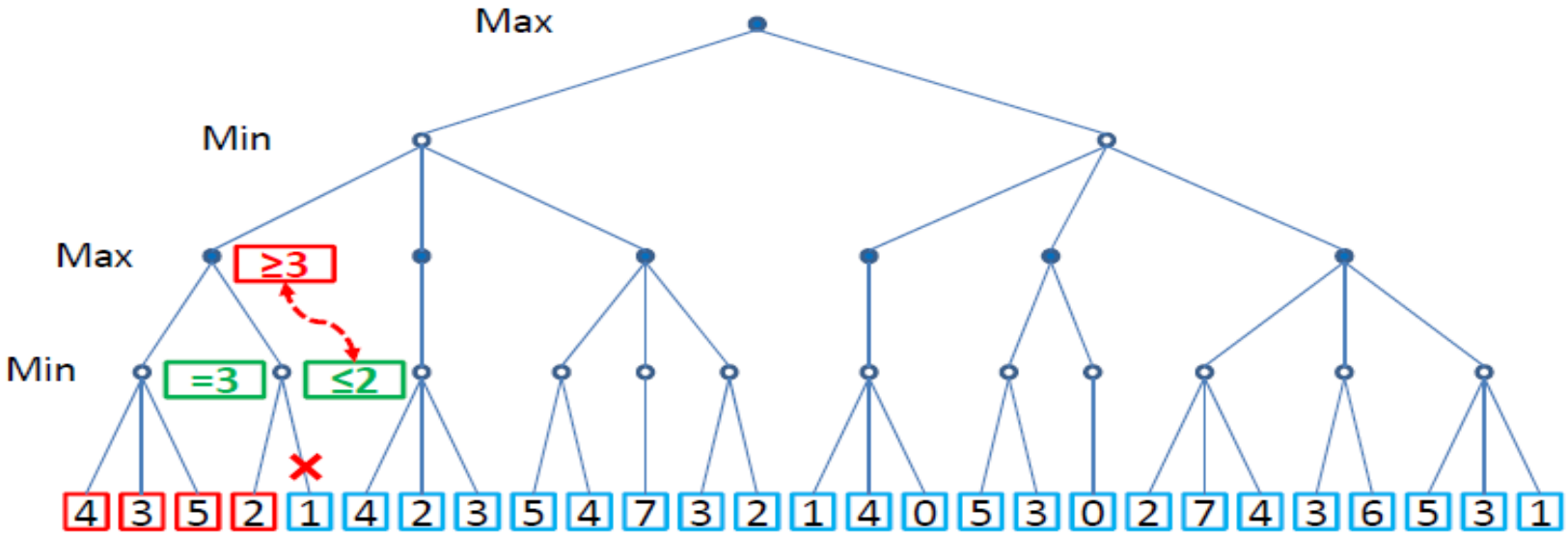

Max
Min
Max
Min
=3
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

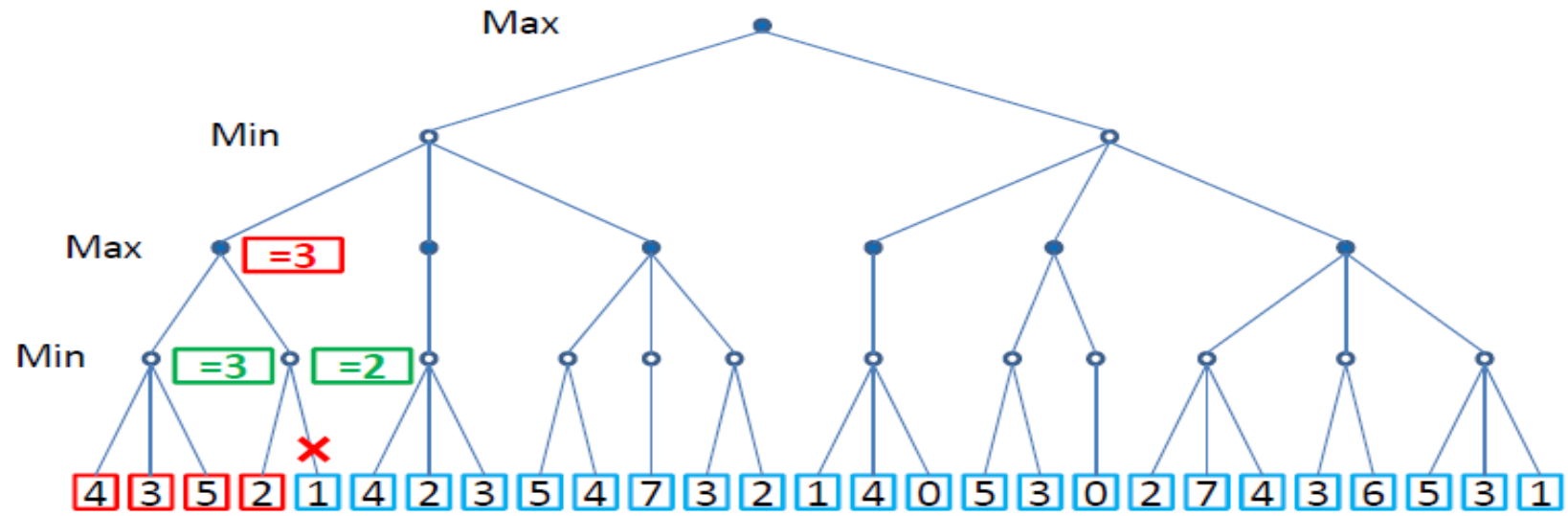
- **β-nodes**: Temporary values at MAX-nodes

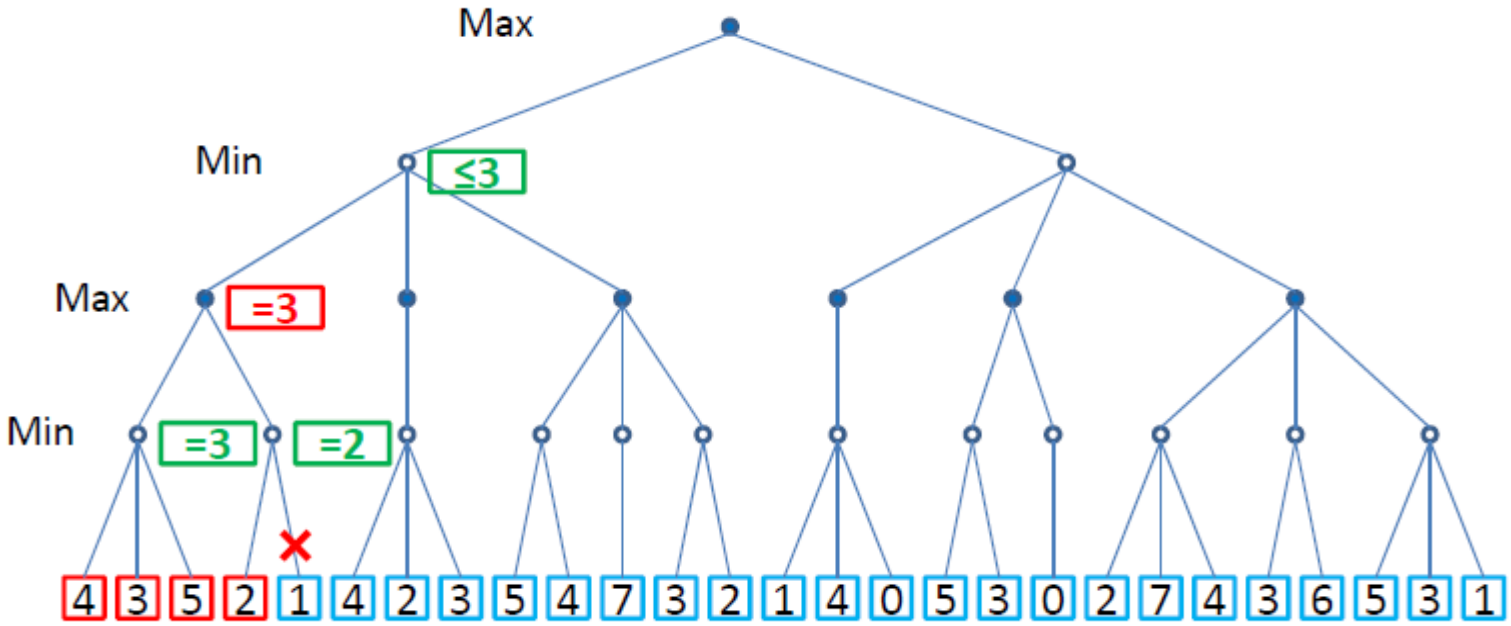


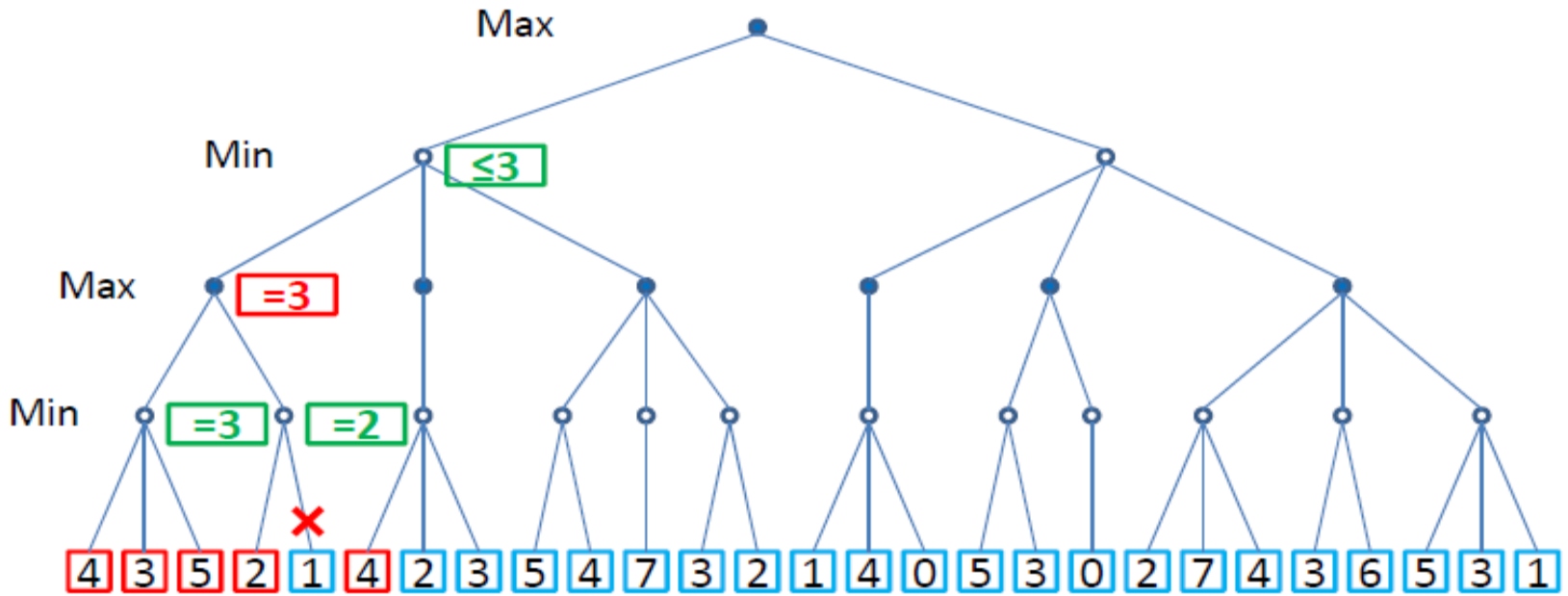


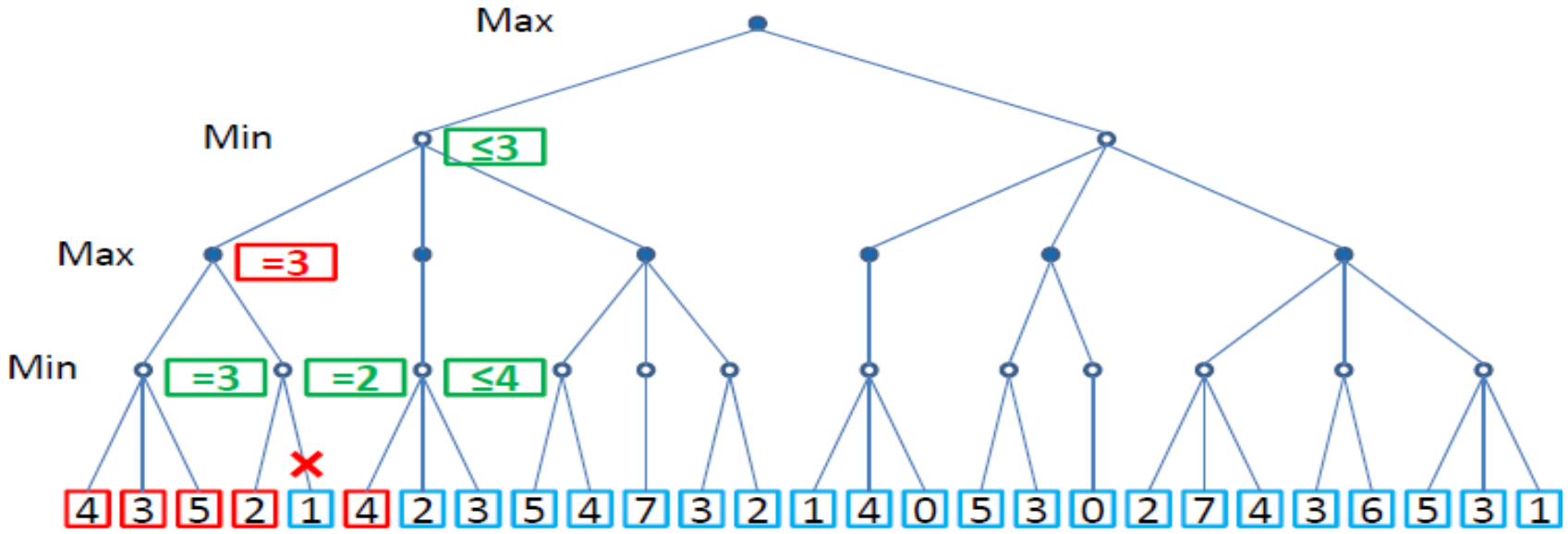
- **Prune: Parent β-node ≥ Child α-node**

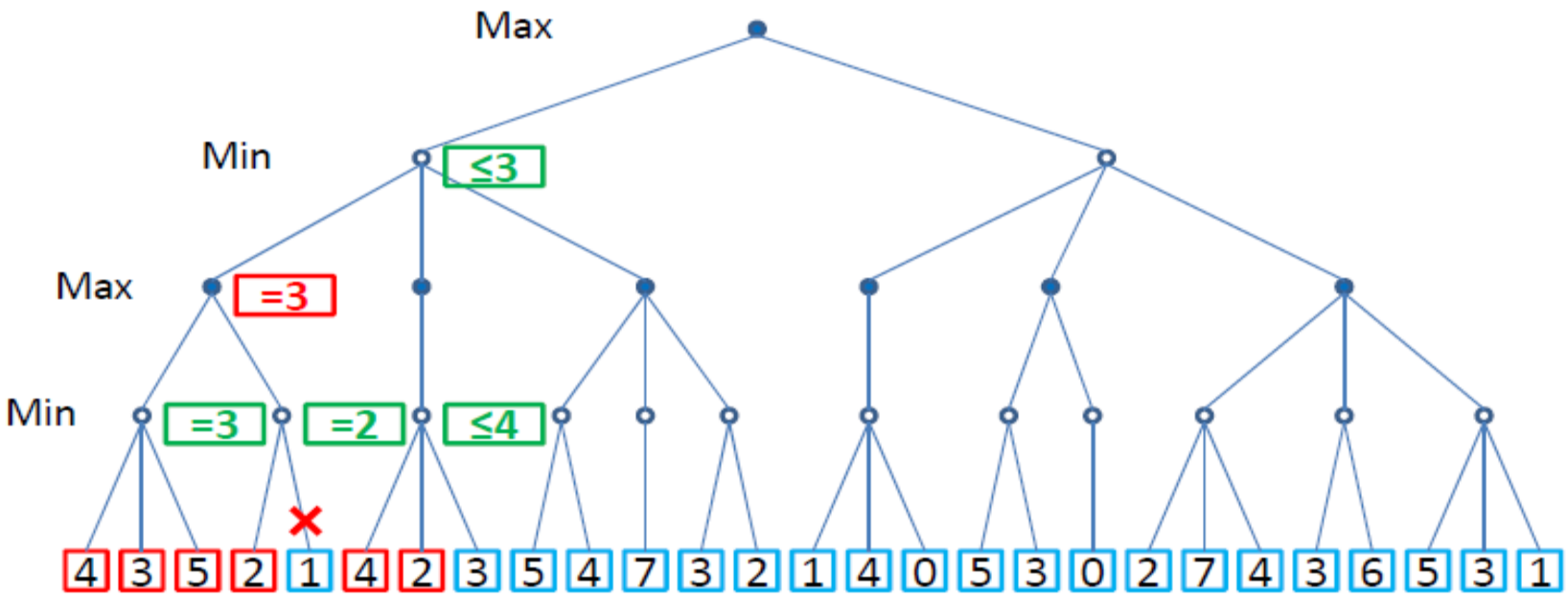


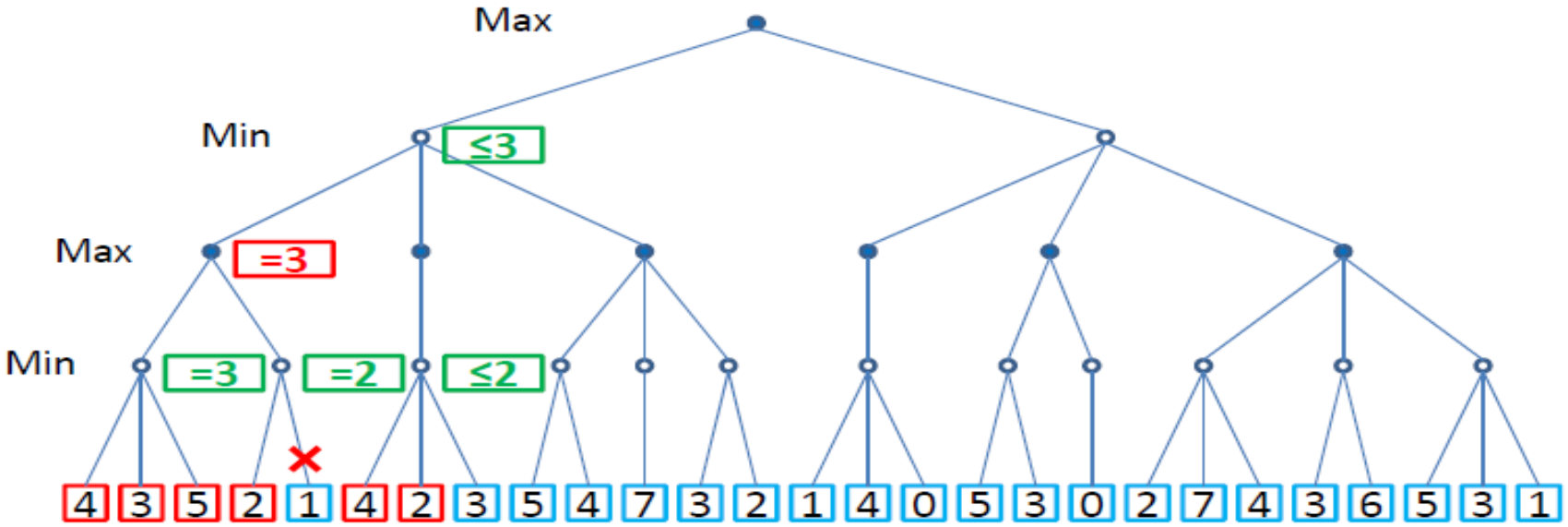

Max
Min
Max
=3
Min
=3
=2
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

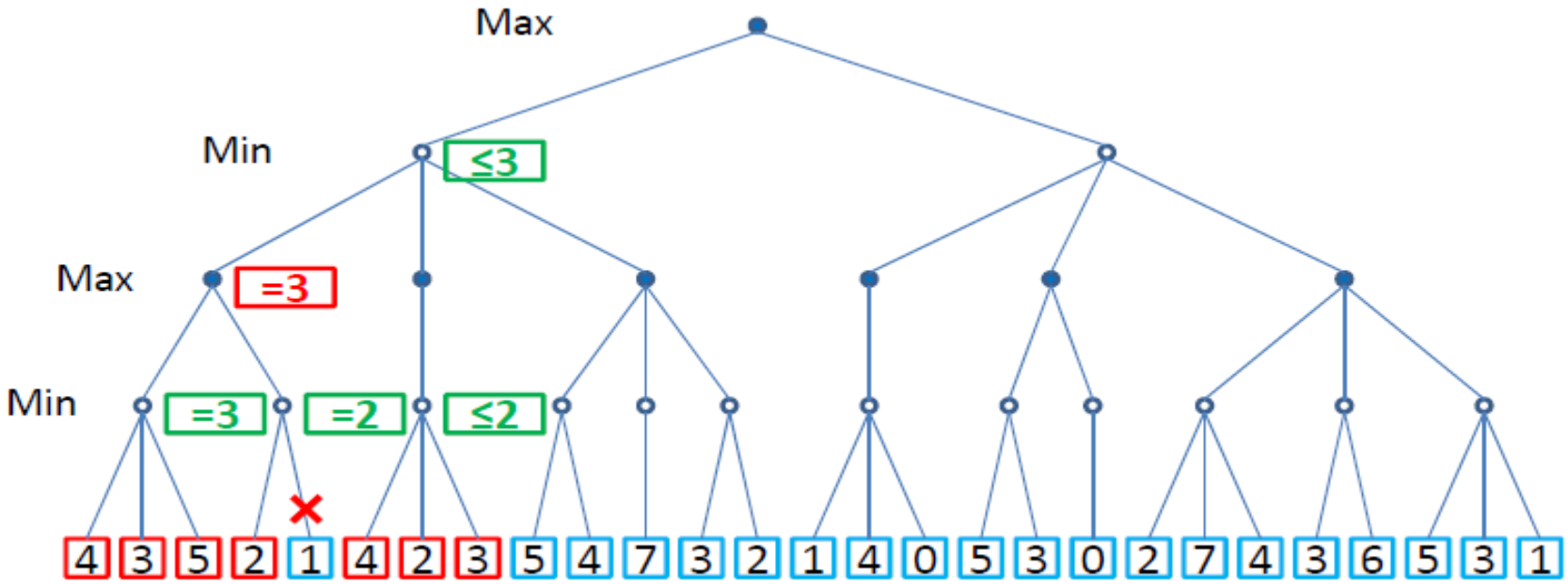

Max
Min
≤3
Max
=3
Min
=3
=2
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

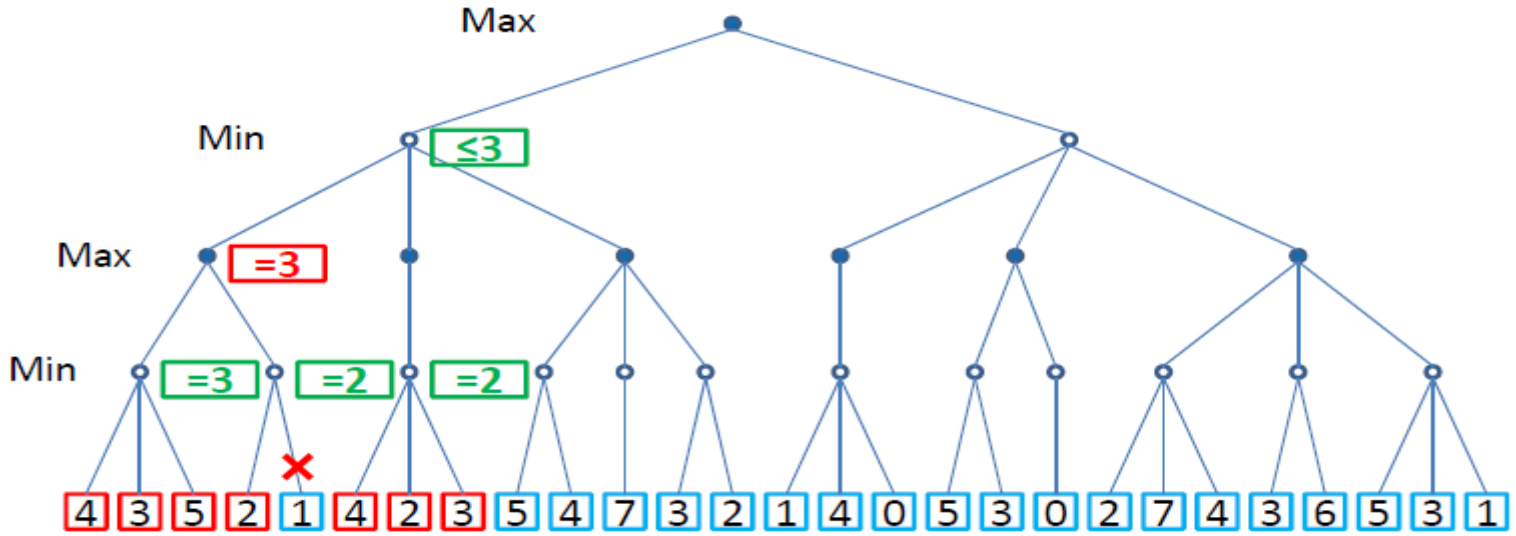

Max
Min
≤3
Max
=3
Min
=3
=2
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

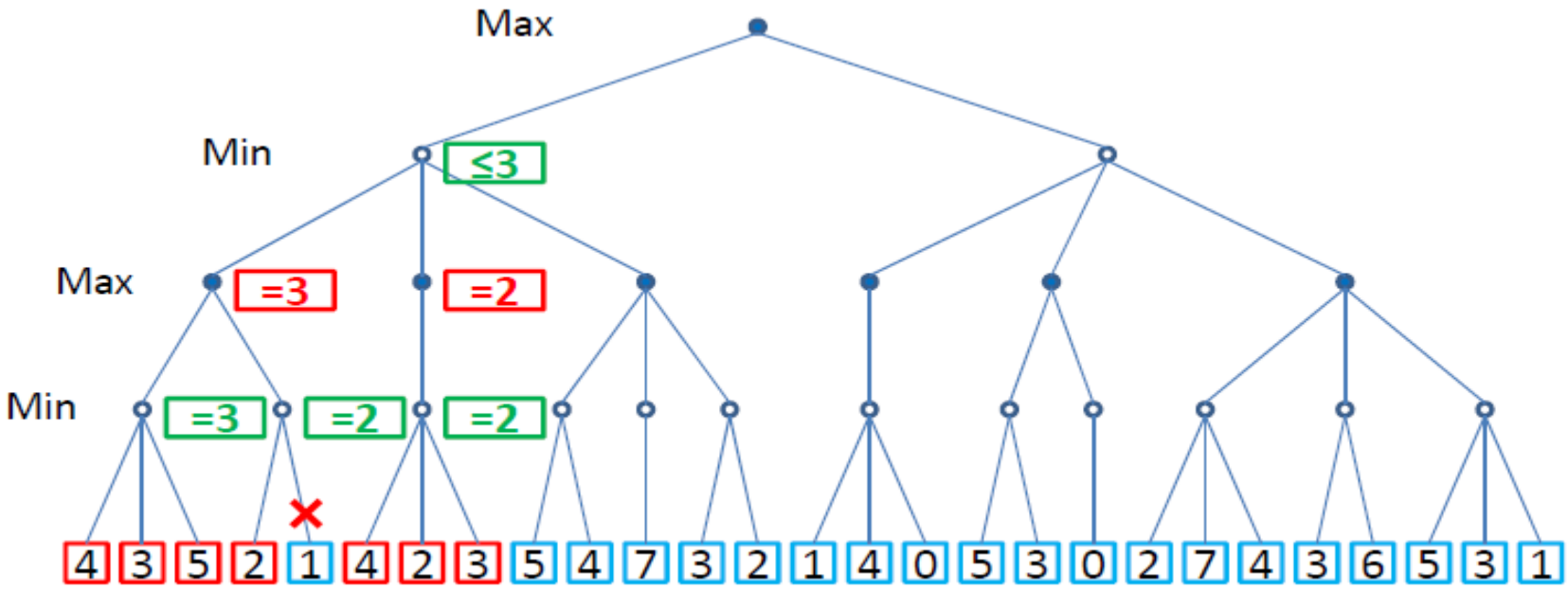

Max
Min
≤3
Max
=3
Min
=3
=2
≤4
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

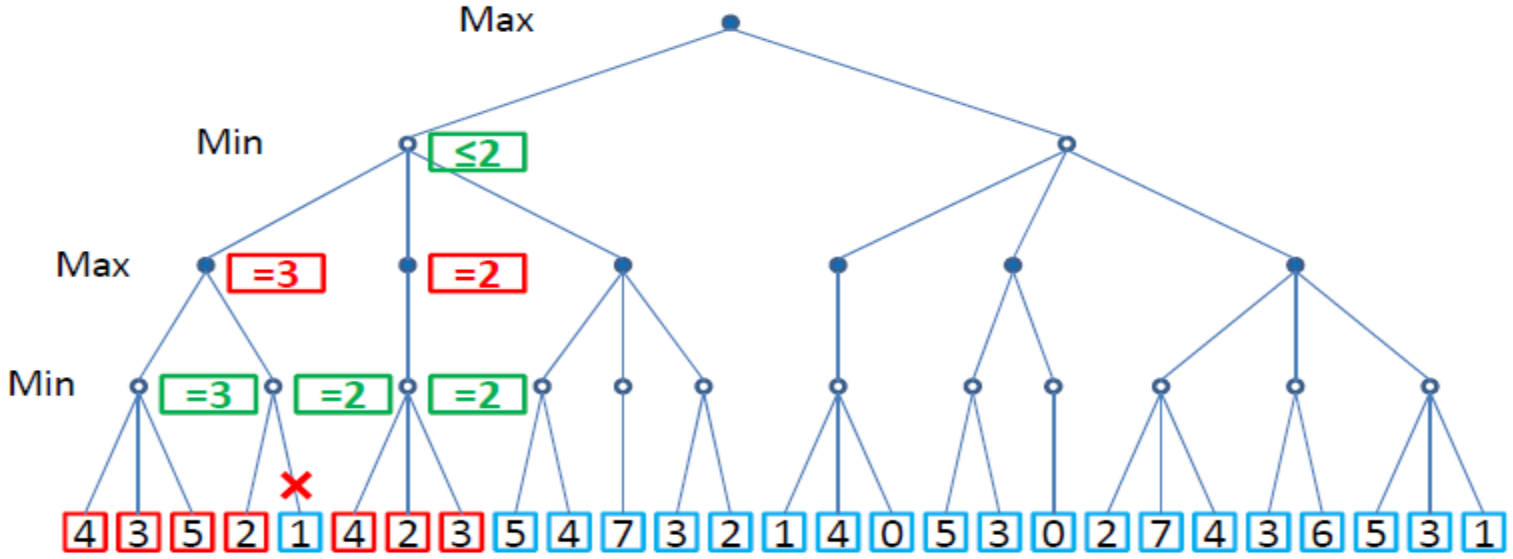

Max
Min
≤3
Max
=3
Min
=3
=2
≤4
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

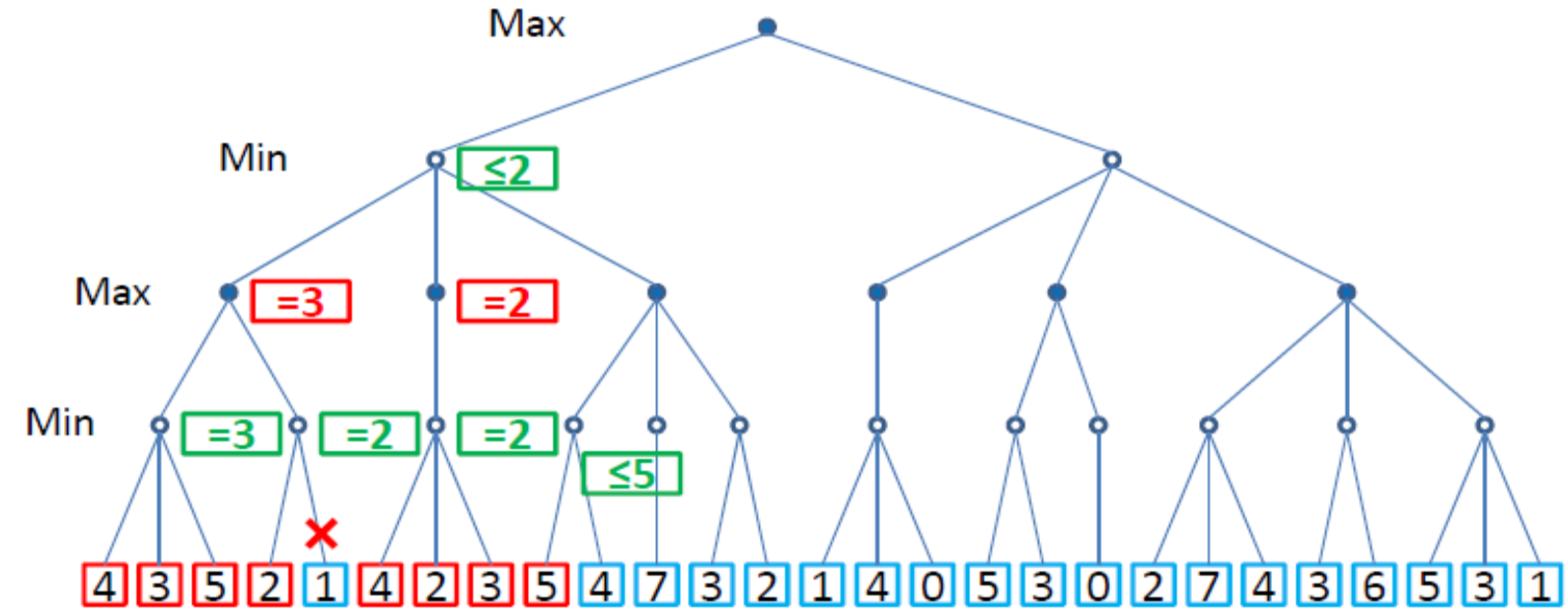

Max
Min
≤3
Max
=3
Min
=3
=2
≤2
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

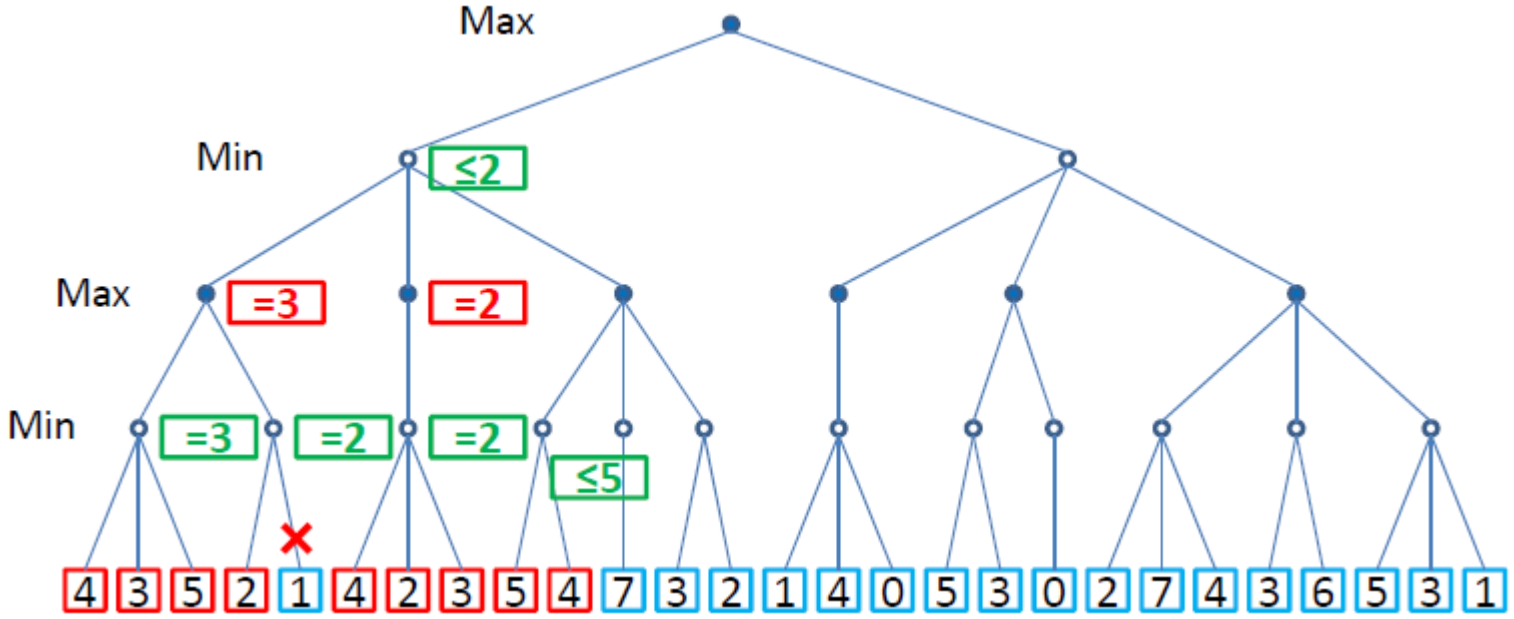

Max
Min
≤3
Max
=3
Min
=3
=2
≤2
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

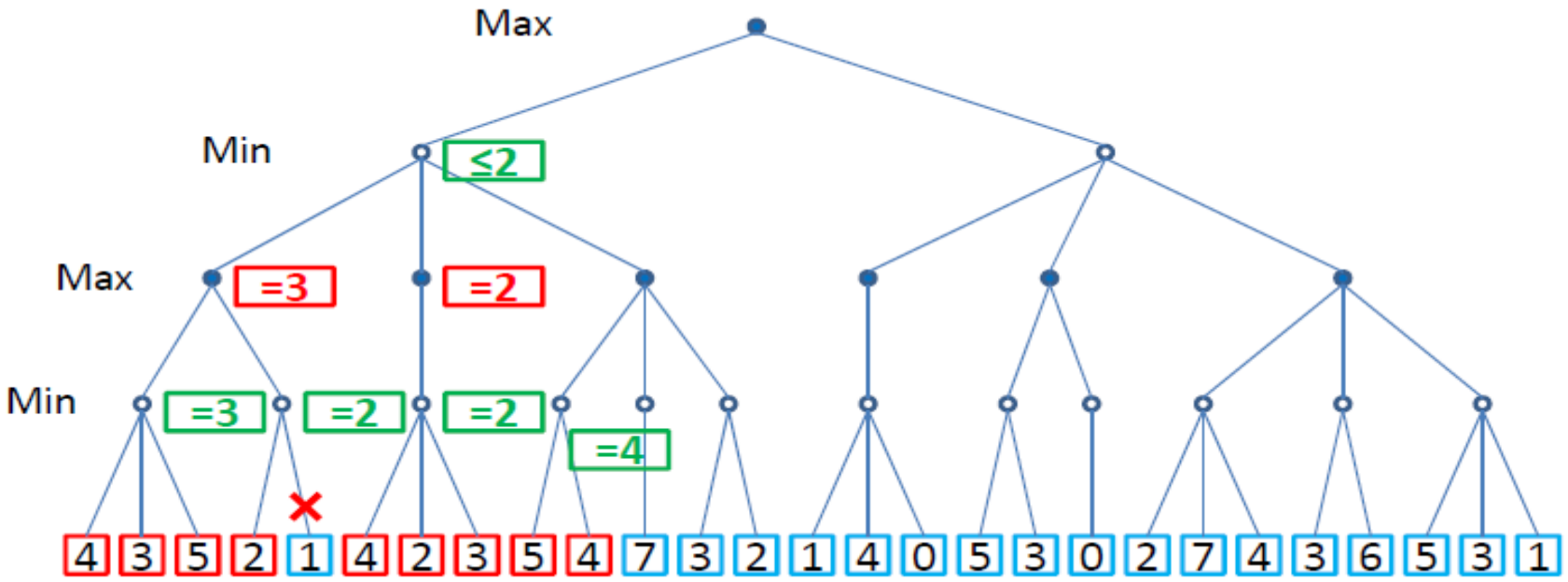

Max
Min
≤3
Max
=3
Min
=3
=2
=2
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

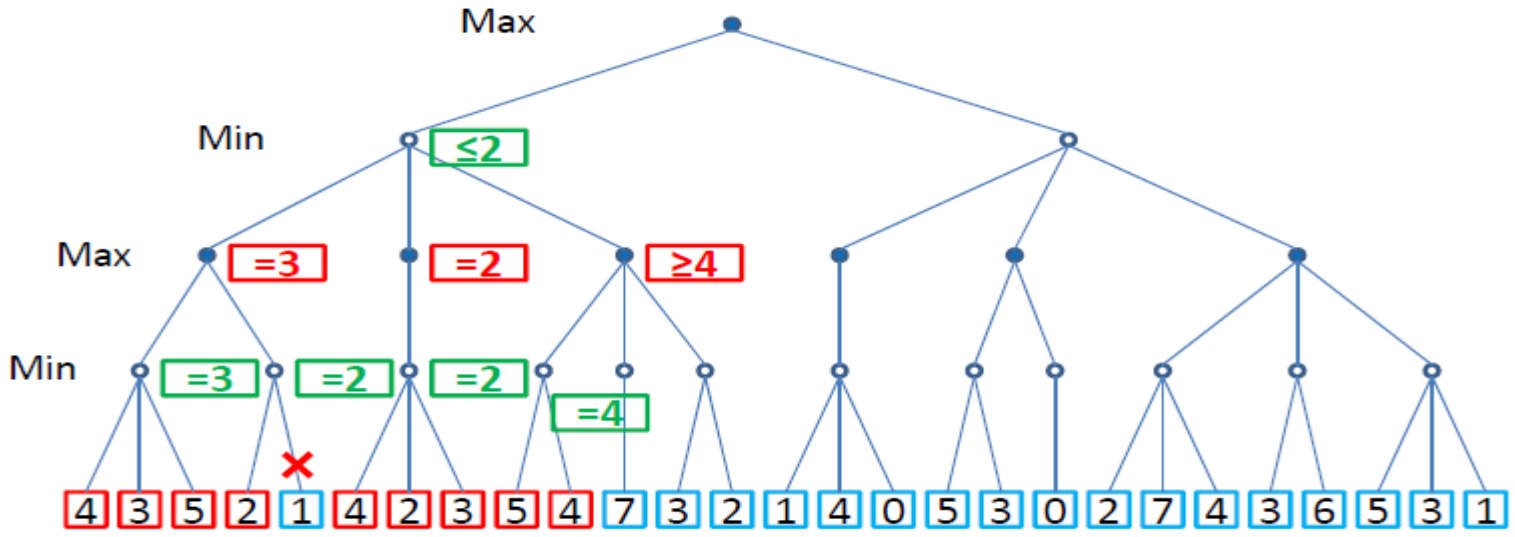

Max
Min
≤3
Max
=3
=2
Min
=3
=2
=2
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

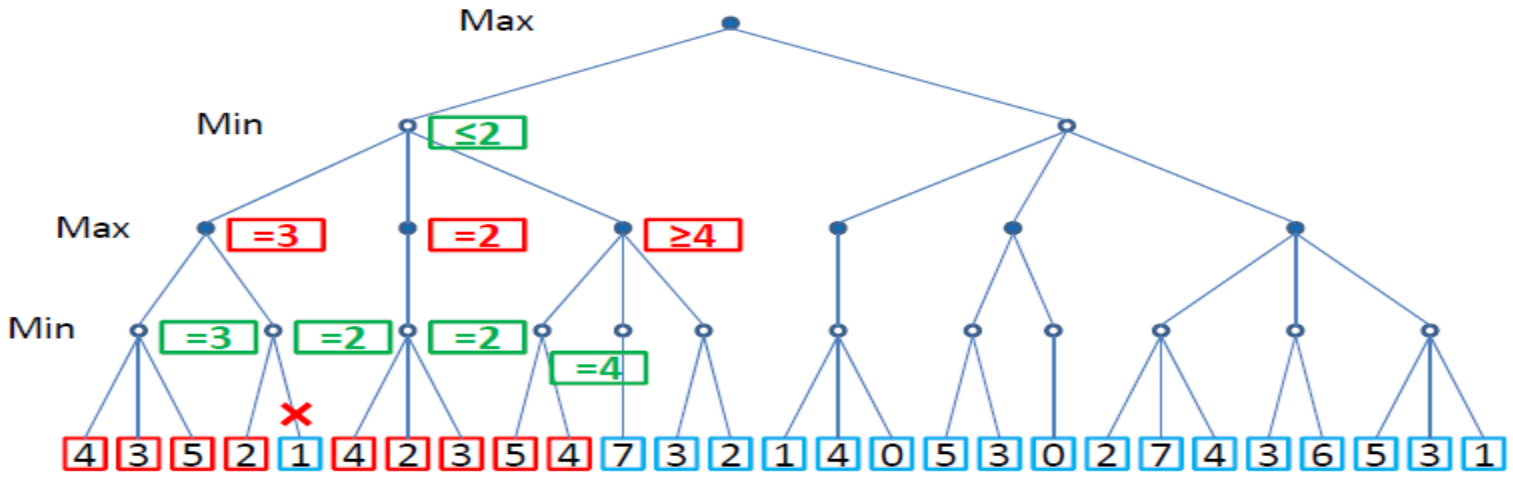

Max
Min
≤2
Max
=3
=2
Min
=3
=2
=2
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1


Max
Min
≤2
Max
=3
=2
Min
=3
=2
=2
≤5
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

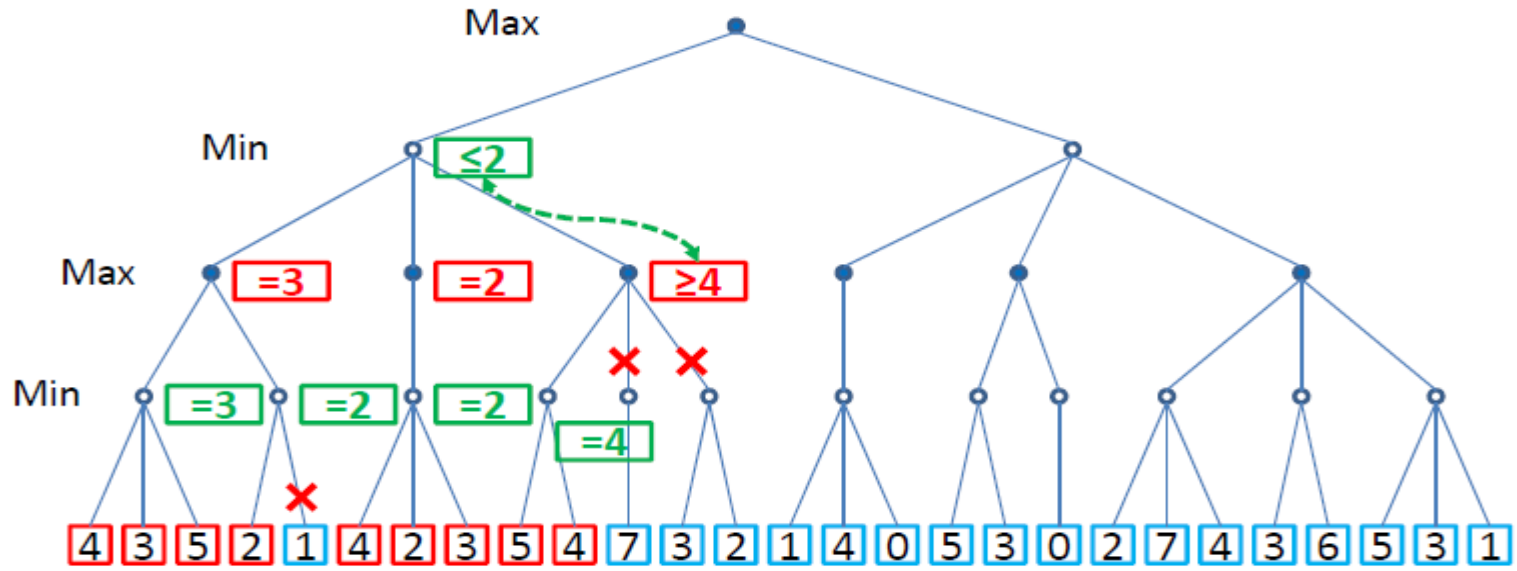

Max
Min
≤2
Max
=3
=2
Min
=3
=2
=2
≤5
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

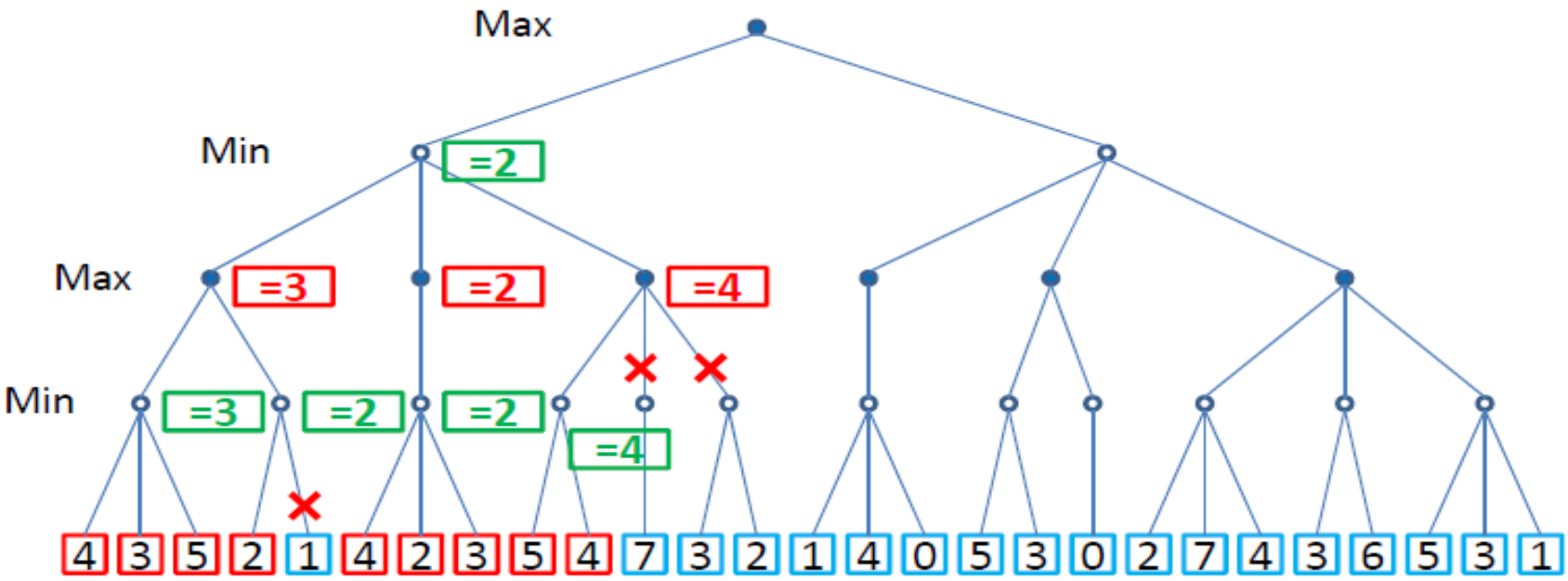

Max
Min
≤2
Max
=3
=2
Min
=3
=2
=2
=4
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

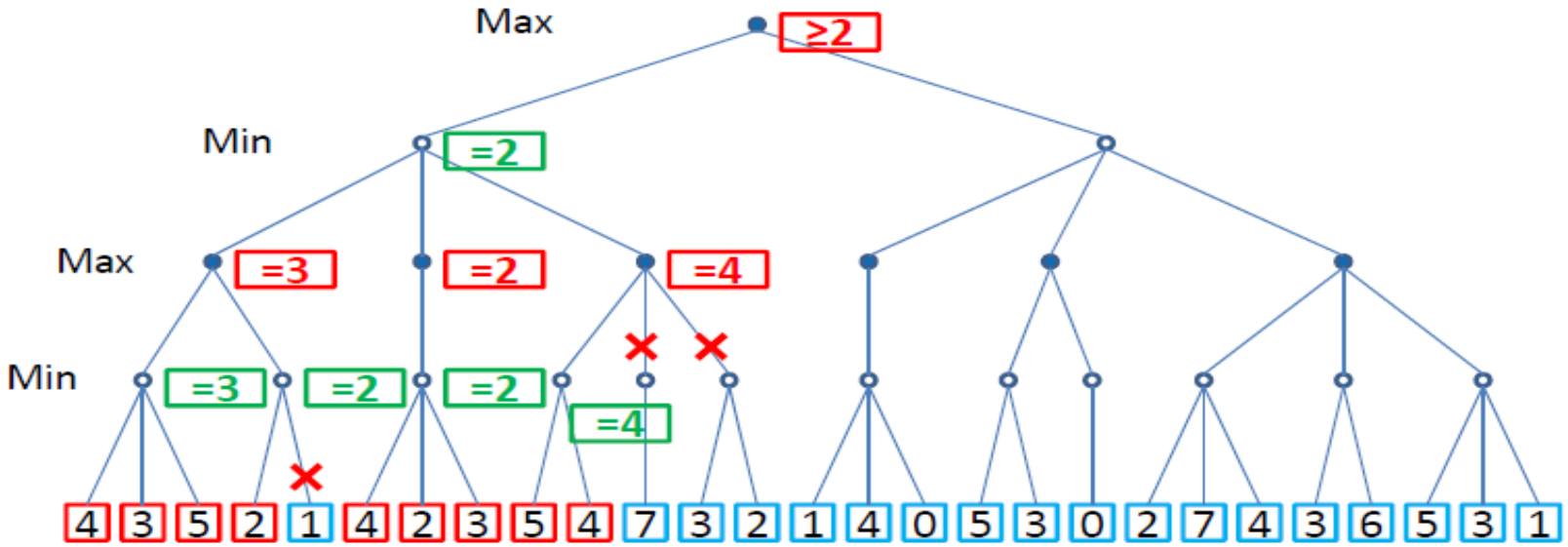

Max
Min
≤2
Max
=3
=2
≥4
Min
=3
=2
=2
=4
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1



- **Prune: Parent α-node ≤ Child β-node**




Max
Min
Max
Min
=2
=3
=2
=4
=3
=2
=2
=4
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1


Max
≥2
Min
=2
Max
=3
=2
=4
Min
=3
=2
=2
=4
4 3 5 2 1 4 2 3 5 4 7 3 2 1 4 0 5 3 0 2 7 4 3 6 5 3 1

Alpha-beta pruning

- *Depending ordering of expansion → Time complexity for perfect ordering* $O(b^{m/2})$-$O(b^{3m/4})$
- *Space complexity →* $O(bm)$

Constraint Satisfaction Problems

A CSP consists of:

- *Finite set of variables* $X_1, X_2, \ldots, X_n$
- *Nonempty* ***domain*** *of possible values* for each variable $D_1, D_2, \ldots D_n$ *where* $D_i = \{v_1, \ldots, v_k\}$
- *Finite set of constraints* $C_1, C_2, \ldots, C_m$
  - —Each *constraint* C_i limits the values that variables can take, e.g., $X_1 \neq X_2$ A *state* is defined as an *assignment* of values to some or all variables.

- **A *consistent assignment* does not violate the constraints.**
- **Example: Sudoku**

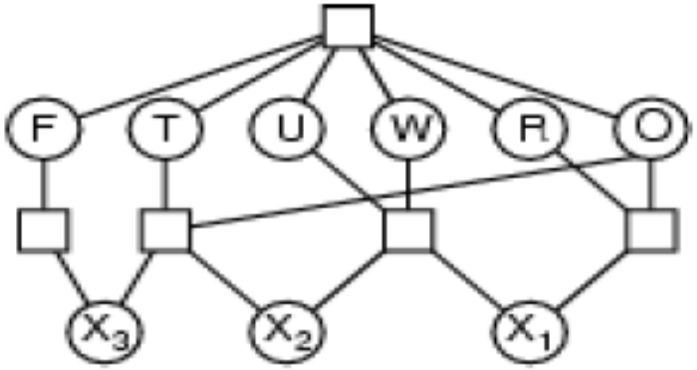
Example: Cryptarithmetic

X_3 X_2 X_1

T W O

\+ T W O

F O U R



- **Variables**: $F\ T\ U\ W\ R\ O, X_1\ X_2 X_3$
- **Domain**: $\{0,1,2,3,4,5,6,7,8,9\}$
- **Constraints**:
  - $Alldiff\ (F,T,U,W,R,O)$
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F,\ T \neq 0,\ F \neq 0$

Example

```
  S E N D
  M O R E
+---------
M O N E Y
```

Value of **M** = 1 [S<=9, M<=9, C1<=1, so S + M + C1 <=19]

Now, S = {9, 8}
when C1 = {0, 1}

Also, E + O + C2 = 10 + N, if C1 = 1
Else, E + O + C2 = N, if C1 = 0

Similarly, N + R + C3 = E + 10 , if C2 = 1
Else, N + R + C3 = E, if C2 = 0

And, D + E = 10 + Y, if C3 = 1
Else, D + E = Y, if C3 = 0

Analysis cont....

- Now, analyzing and deducing values from Right to Left left to right, we get

If C1 = 1, S = 9 Then C1 + S + M = 11, which makes O=1 [False].

if C1 = 1, S = 8, Then C1 + S + M = 10, which makes O = 0.

if C1 = 0, S = 9, then C1 + S + M = 10, which makes O = 0.

So, **O = 0** is valid.

Now, Both the above given alternatives look equally probable at the moment.

Since O = 0, E + O + C2 will give carry only when C2 = 1 and E = 9.
But, that will give E + O + C2 = 10, and N=0. [False, O = 0 is already established]

So, the second alternative must be correct, i.e. C1=0, S=9, O=0, M=1.

Analysis cont....

- Now, E + O + C2 = E + C2 = N.
  If C2 = 0, then E = N [Invalid]
  so, **C2 = 1 and N = E + 1**
  Also,

  N + R + C3 = E + 10
  E + 1 + R + C3 = E + 10
  or, R + C3 = 9

  Now if C3 = 0, R = 9 [Invalid, S = 9]
  So, C3 = 1 and R = 8.

  Now,
  Choices For E = {2, 3, 4, 5, 6, 7 } so N = {3, 4, 5, 6, 7, 8 }
  But 8 is already taken.

  So, E = {2, 3, 4, 5, 6} and N = {3, 4, 5, 6, 7}

Analysis cont....

- Now,
  D + E = 10 + Y >= 12

  so, (D,E) = { (5, 7), (6, 7), (7, 5), (7, 6) } which means N = { 6, 7, 8}
  But 8 is already taken so N = {6, 7}
  so, (D, E) = { (7, 5), (7, 6)} i.e D = {7} and E = {5, 6}

  Now, if E = 6, N=7 [invalid, 7 is already taken, D = 7]
  so , E = 5, N = 6 and Y = 2.

  Hence the Solution becomes ,

  S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2.

  i.e.
  ```
    9 5 6 7
  + 1 0 8 5
  ---------
  1 0 6 5 2
  ```

  That's about it.

Constraint satisfaction problems

- **An assignment is *complete* when every variable is assigned a value.**
- **A *solution* to a CSP is a complete assignment that satisfies all constraints.**
- **Applications:**
  - Map coloring
  - Line Drawing Interpretation
  - Scheduling problems
    - —Job shop scheduling
    - —Scheduling the Hubble Space Telescope
  - Floor planning for VLSI
- **Beyond our scope: CSPs that require a solution that maximizes an *objective function*.**

The water jug problem answer

Action / Successor Functions

1. $(x, y \mid x < 4) \rightarrow (4, y)$	"Fill *4*"
2. $(x, y \mid y < 3) \rightarrow (x, 3)$	"Fill *3*"
3. $(x, y \mid x > 0) \rightarrow (0, y)$	"Empty *4*"
4. $(x, y \mid y > 0) \rightarrow (x, 0)$	"Empty *3*"
5. $(x, y \mid x+y \geq 4$ *and* $y > 0)$	$\longrightarrow (4, y - (4 - x))$
	"Pour from *3* to *4* until *4* is full"
6. $(x, y \mid x+y \geq 3$ *and* $x > 0)$	$\longrightarrow (x - (3 - y), 3)$
	"Pour from *4* to *3* until *3* is full"
7. $(x, y \mid x+y \leq 4$ *and* $y > 0)$	$\longrightarrow (x+y, 0)$
	"Pour all water from *3* to *4*"

References

- Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach, 4th US ed. https://aima.cs.berkeley.edu/
- Bart Selman, CS 4700: Foundations of Artificial Intelligence,https://www.cs.cornell.edu/courses/cs4700/2013fa/
- Chuck Dyer CS 540: Introduction to Artificial Intelligence, https://pages.cs.wisc.edu/~dyer/cs540.html
- Mitch Marcus CIS 391: Introduction to Artificial Intelligence, https://www.cis.upenn.edu/~mitch/
- https://sites.cs.ucsb.edu/~yuxiangw/classes/AICourse-2022Spring/
- https://homes.cs.washington.edu/~shapiro/EE562/notes/BeyondClassicalSearch2020.pdf