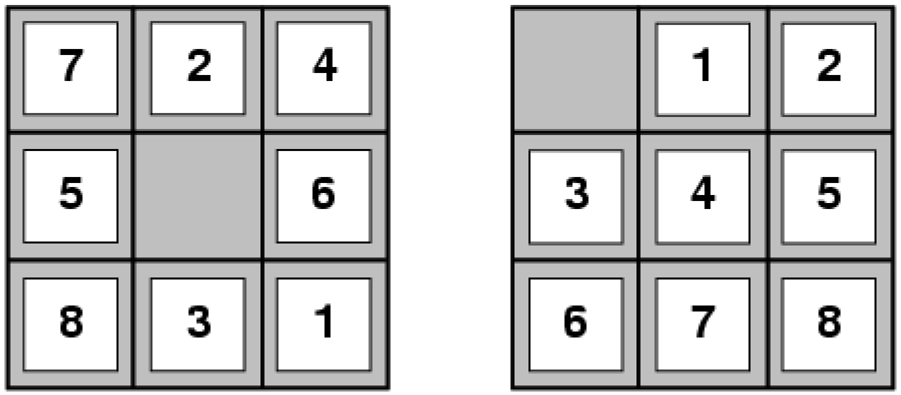
# Solving problems by searching

## Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms

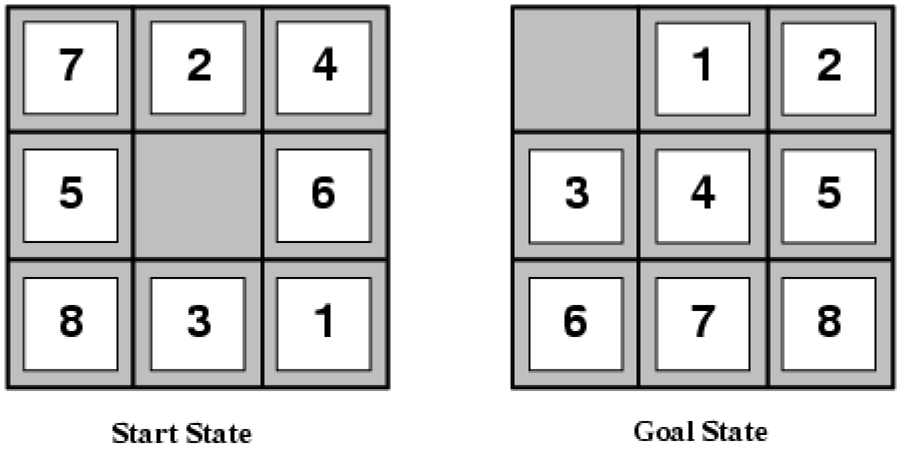
# Example: The 8-puzzle



Start State Goal State

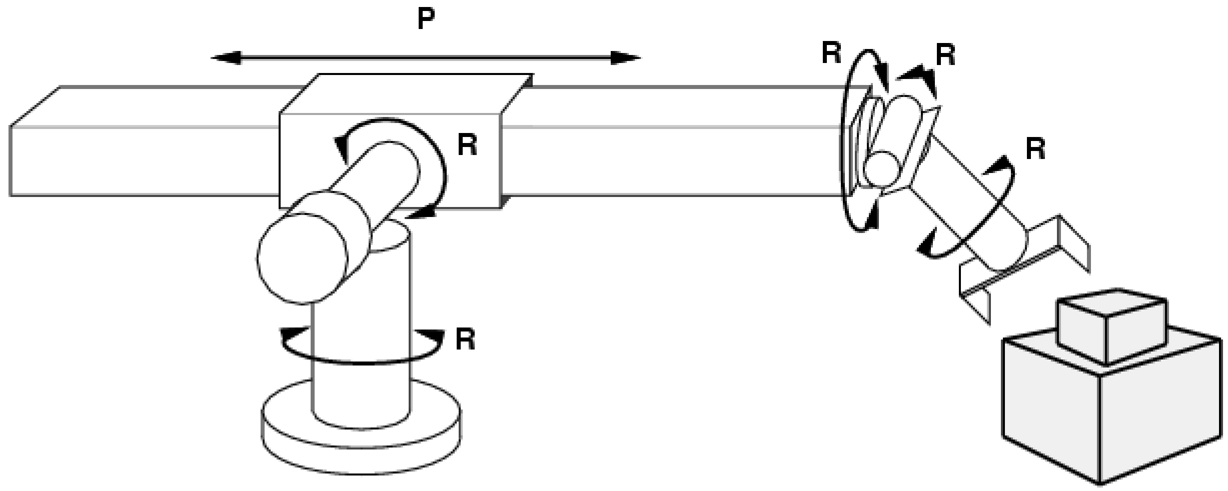
- states?
- actions?
- goal test?
- path cost?

# Example: The 8-puzzle



- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

# Example: robotic assembly



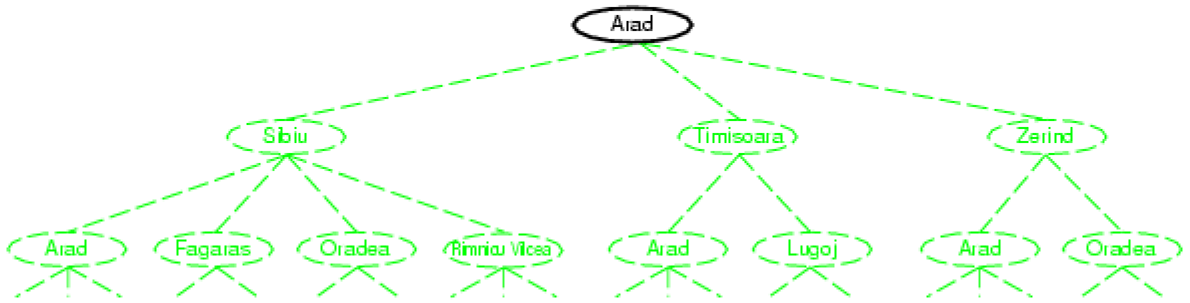
- states?: real-valued coordinates of robot joint angles parts of the object to be assembled
- actions?: continuous motions of robot joints
- goal test?: complete assembly
- path cost?: time to execute

## Tree search algorithms

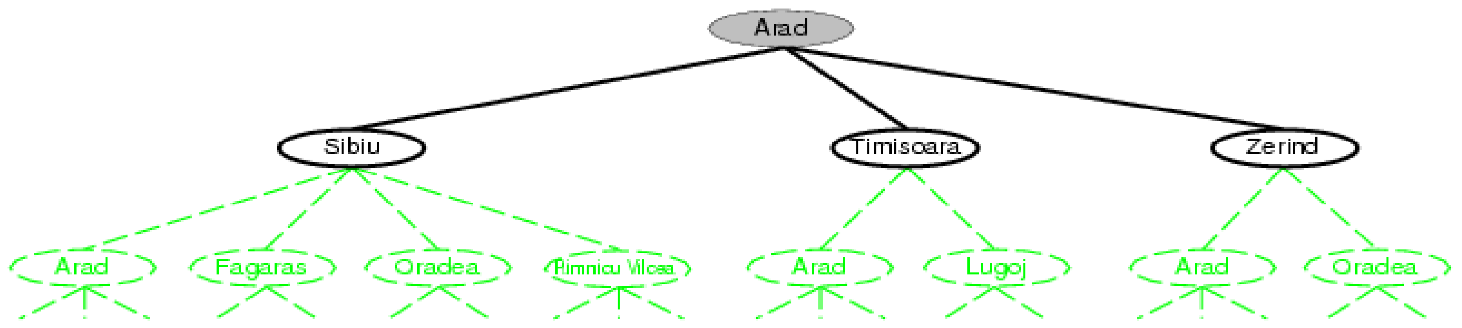
- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

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

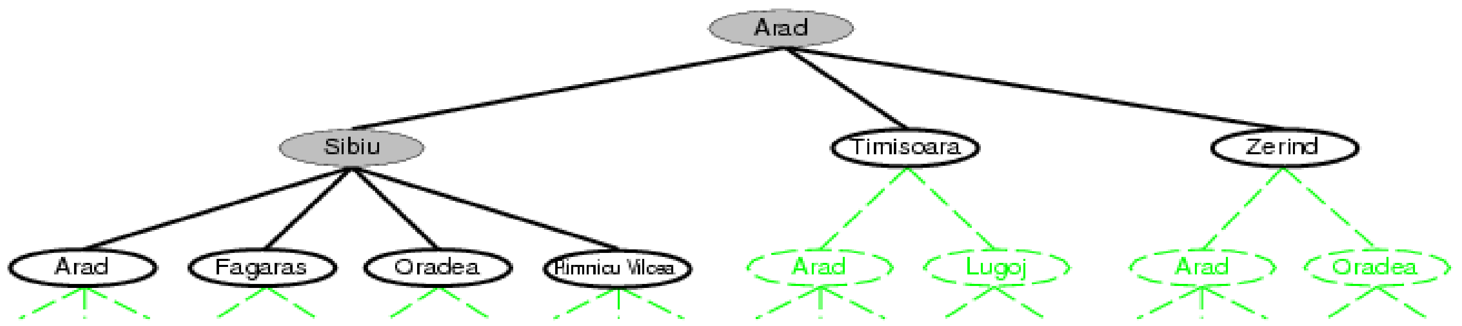
## Tree search example



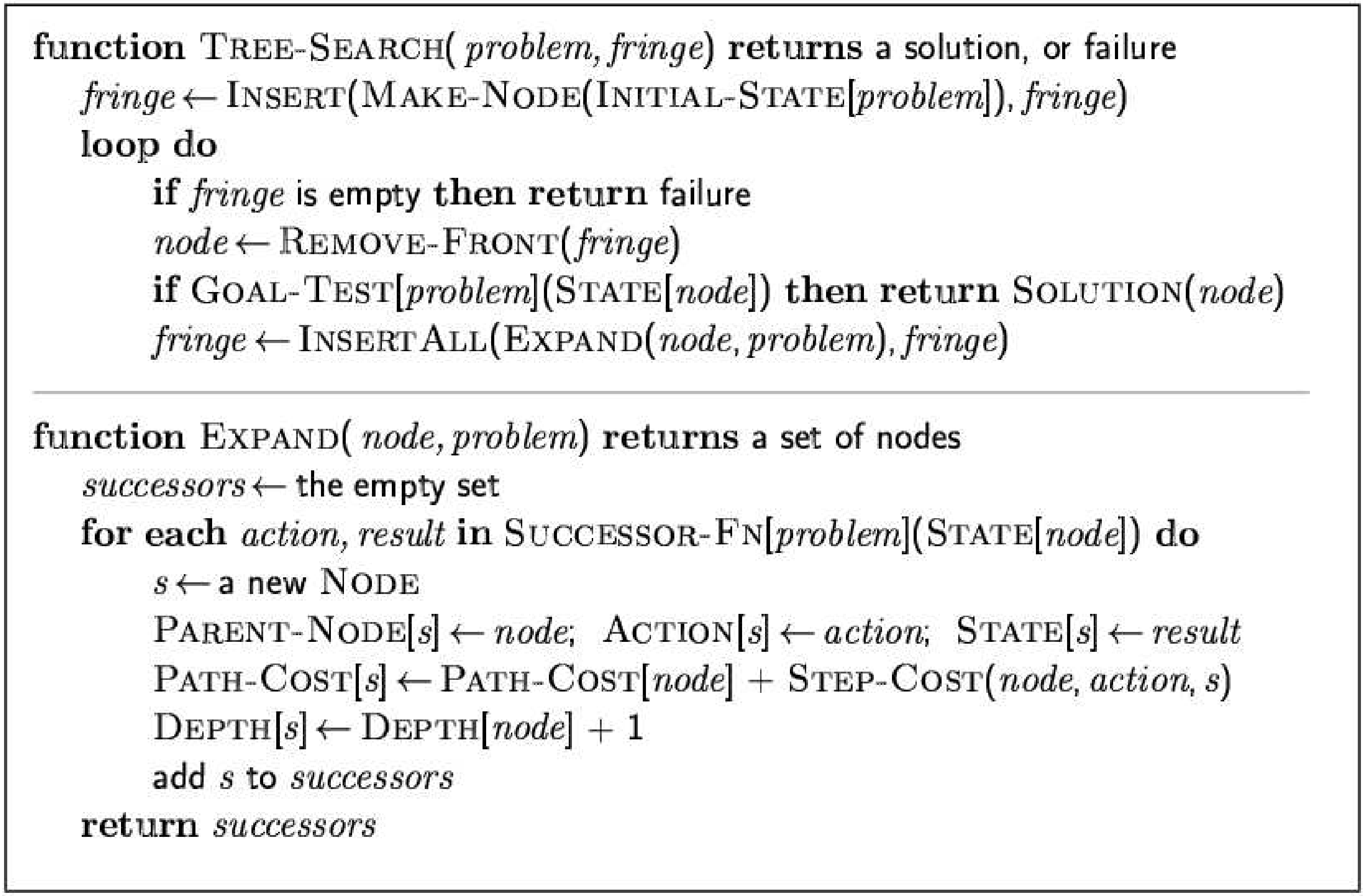
## Tree search example



# Tree search example



# Implementation: general tree search

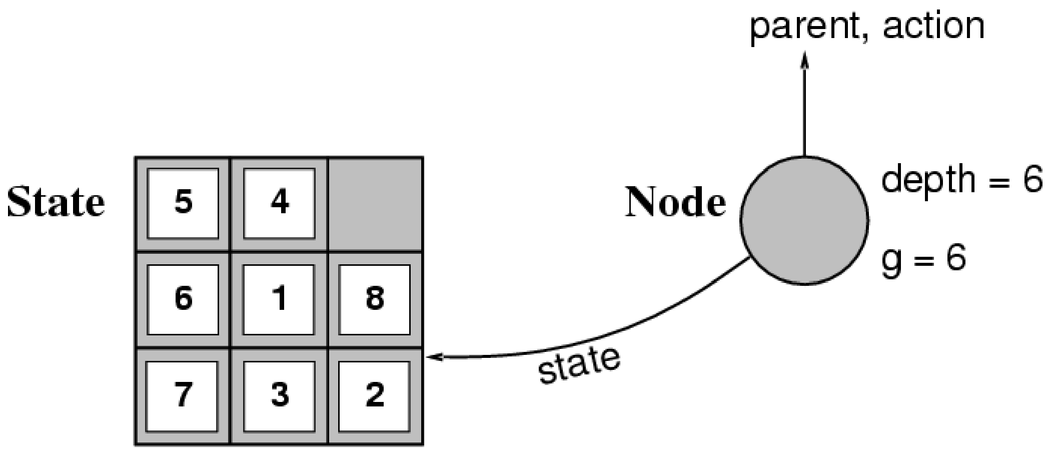


```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

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

## Implementation: states vs. nodes

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



- The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

## 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 ∞)

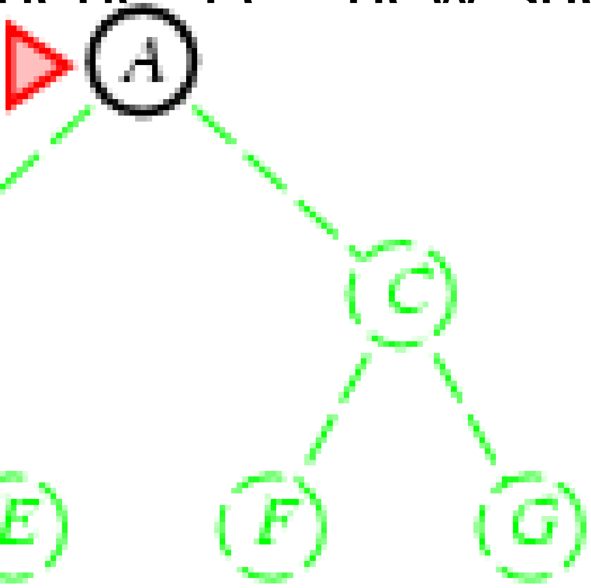
## Uninformed search strategies

- Uninformed 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

## Breadth-first search

Expand shallowest unexpanded node

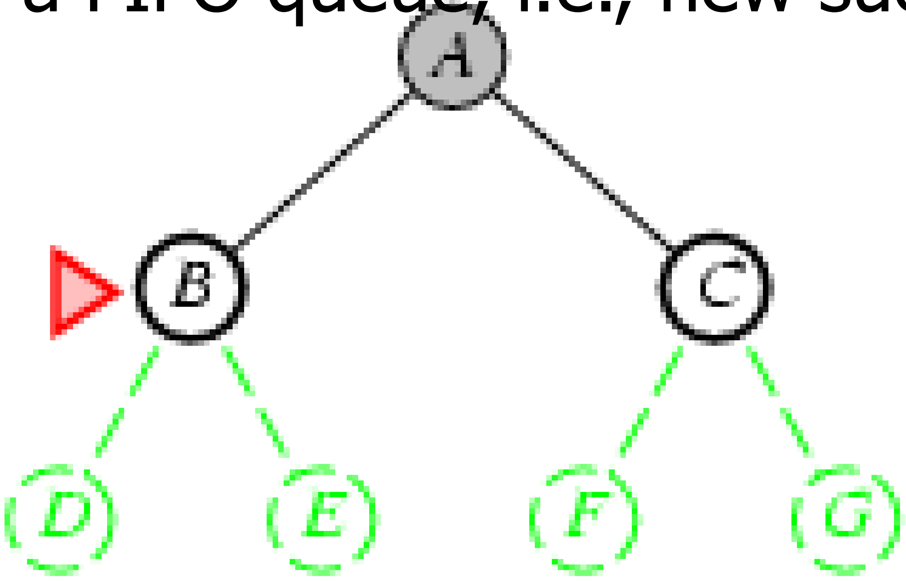
- 
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



## Breadth-first search

Expand shallowest unexpanded node

- 
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

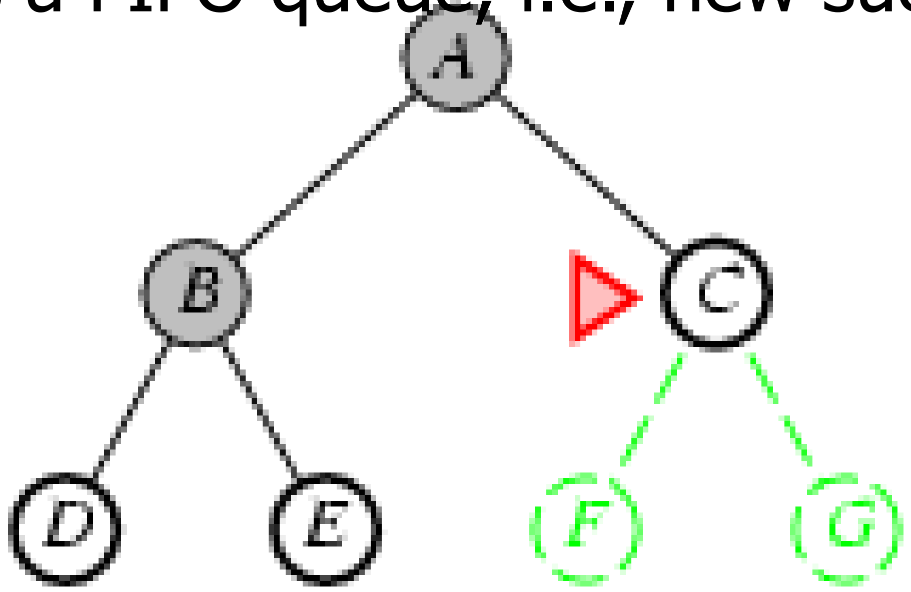


# Breadth-first search

Expand shallowest unexpanded node

-

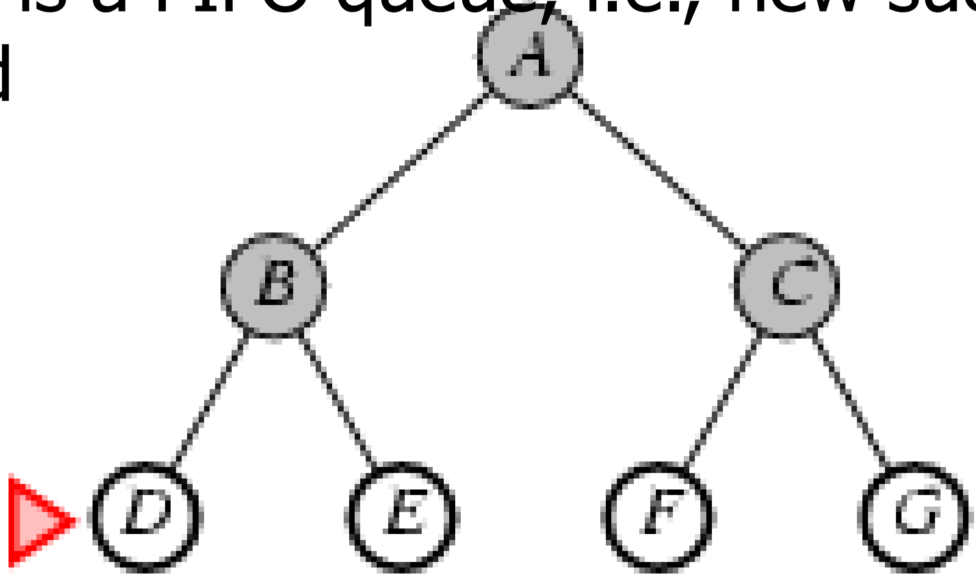
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



## Breadth-first search

Expand shallowest unexpanded node

- 
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



## Properties of breadth-first search

- Complete? Yes (if *b* is finite)
- Time? *1+b+b2+b3*+… +*bd* + *b(bd-1*) = O(bd+1)
- Space? *O(bd+1)* (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)

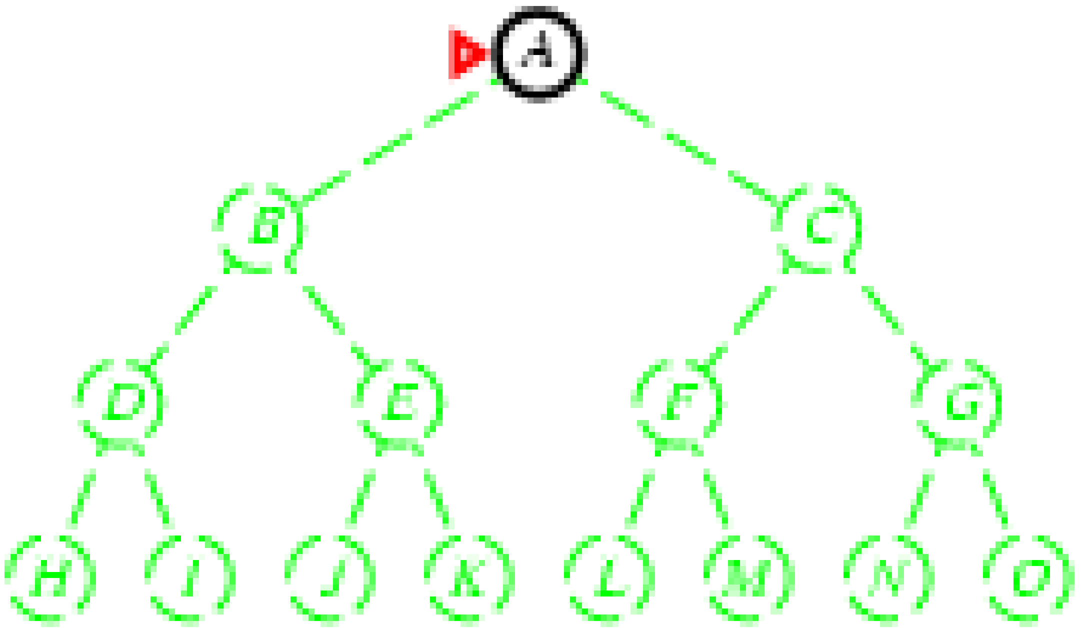
- Space is the bigger problem (more than time)

## Uniform-cost search

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

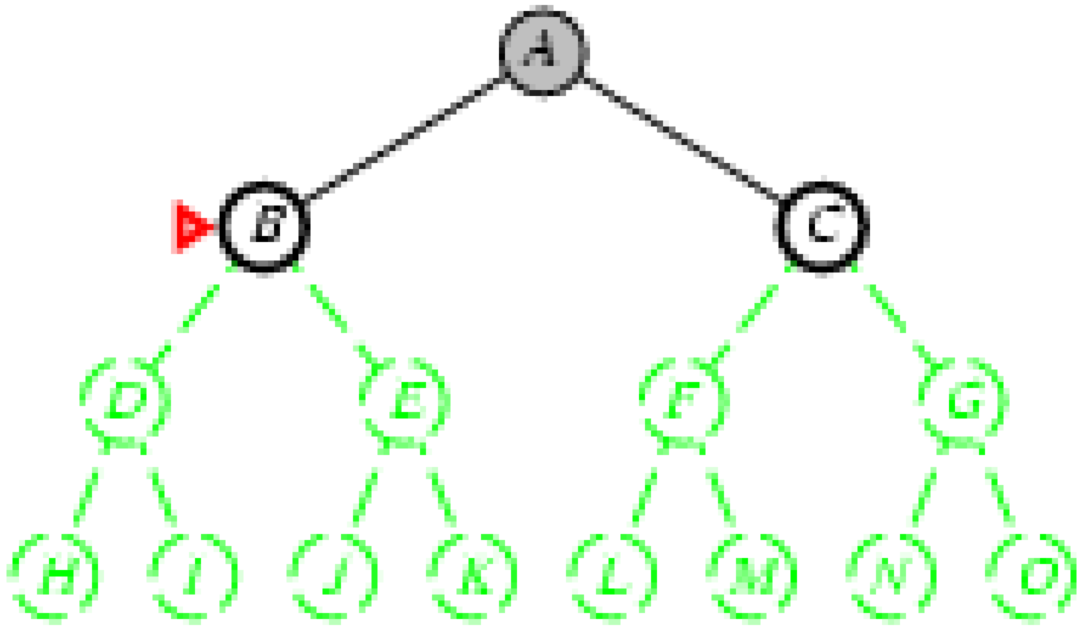
## Depth-first search

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



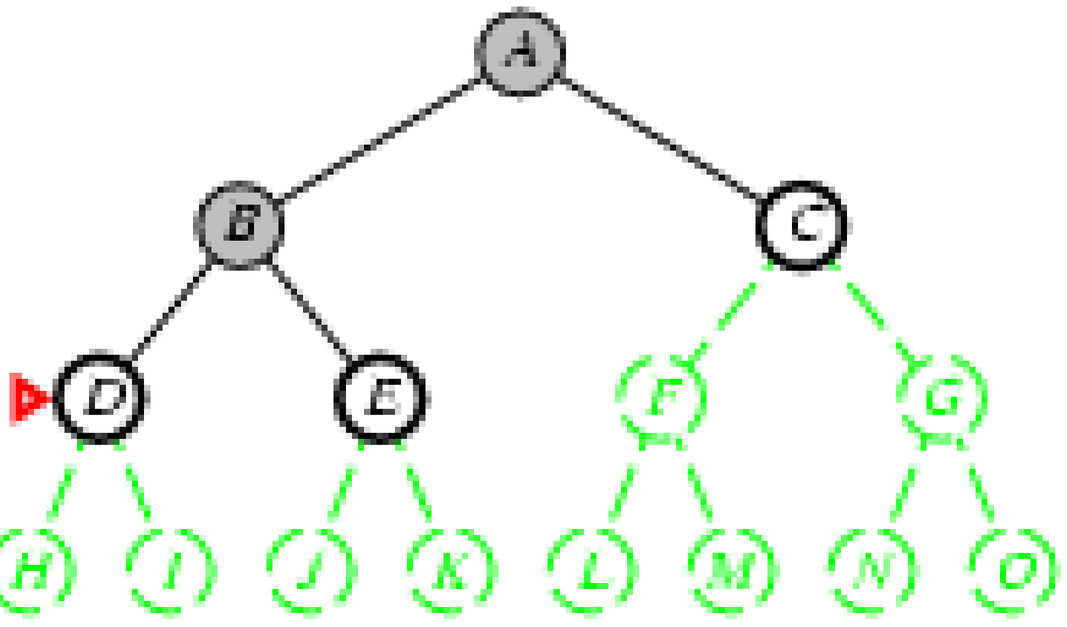
# Depth-first search

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



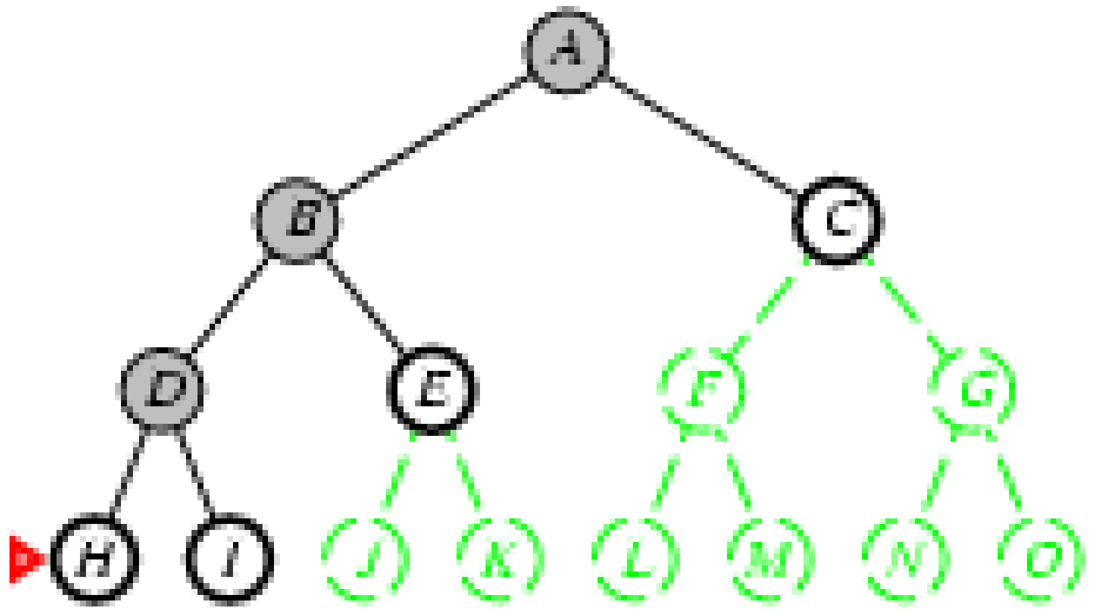
# Depth-first search

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



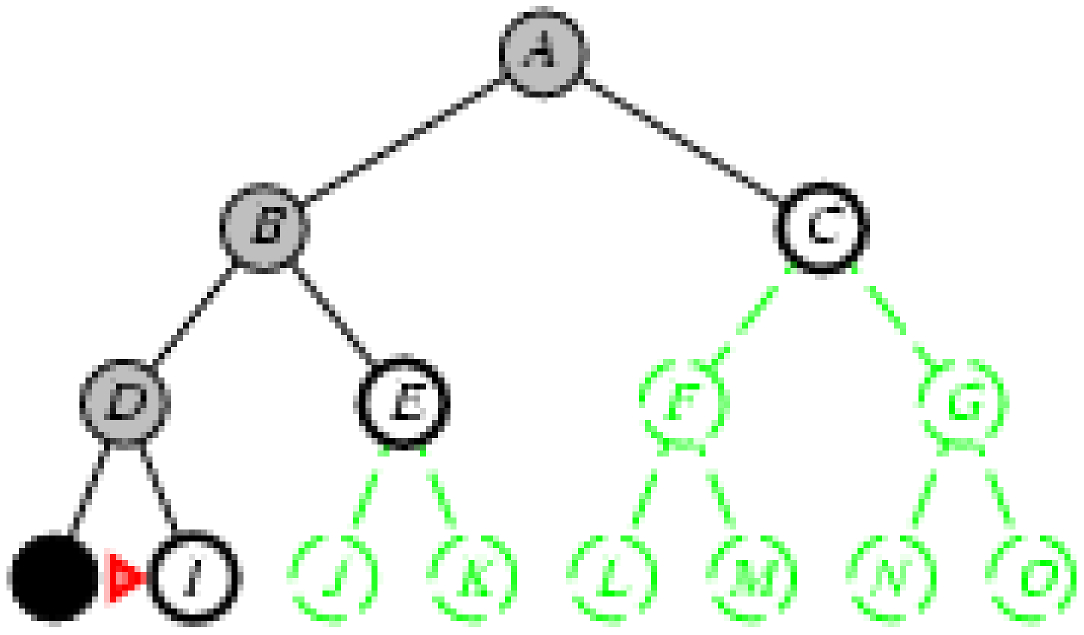
## Depth-first search

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



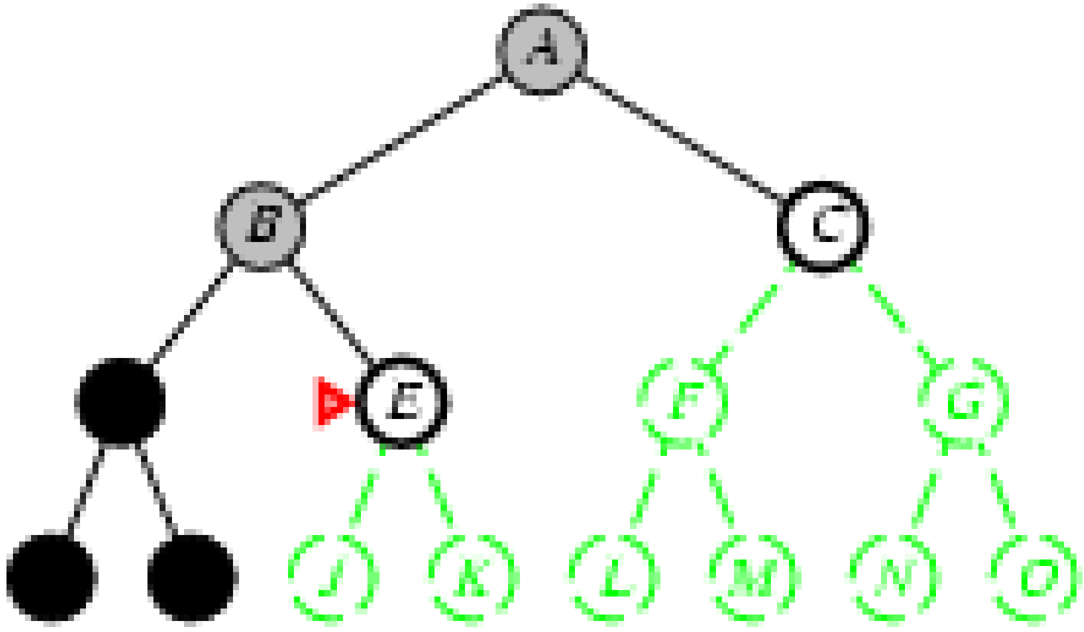
# Depth-first search

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



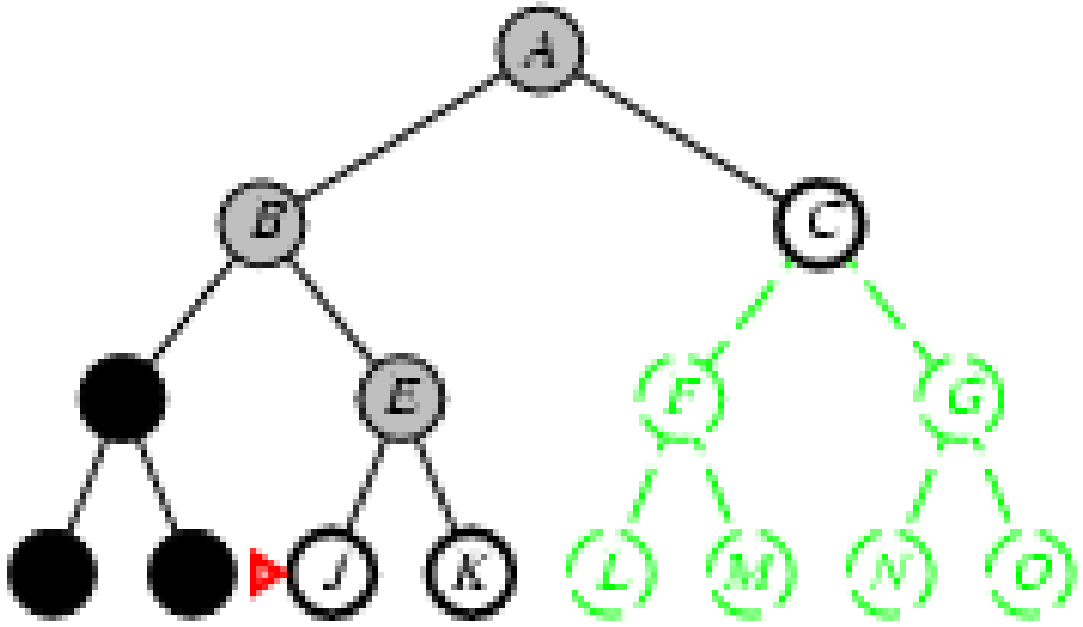
# Depth-first search

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



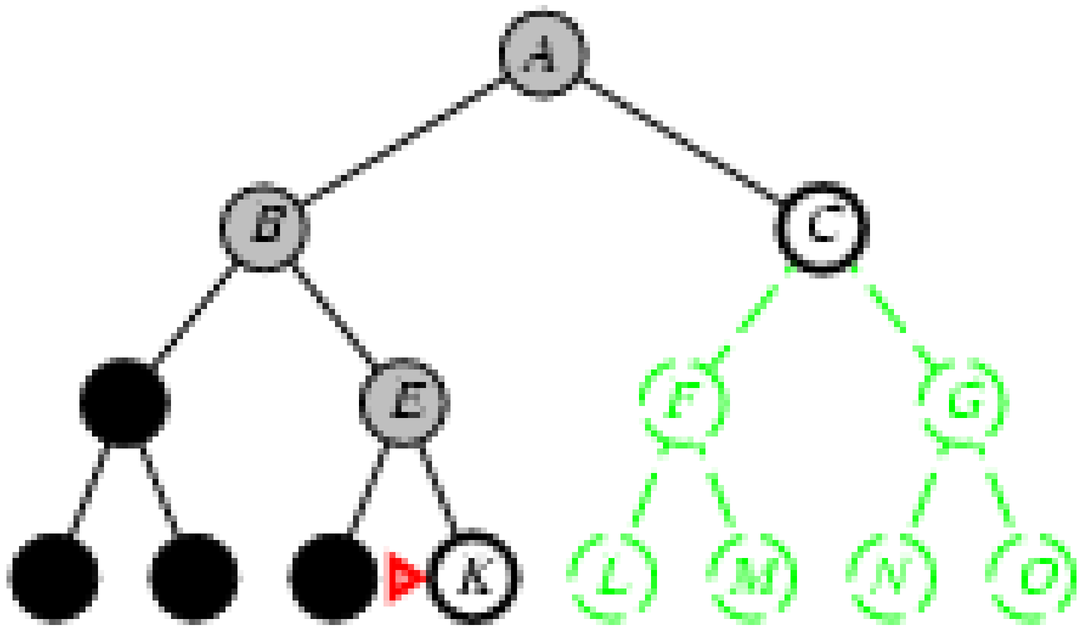
## Depth-first search

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



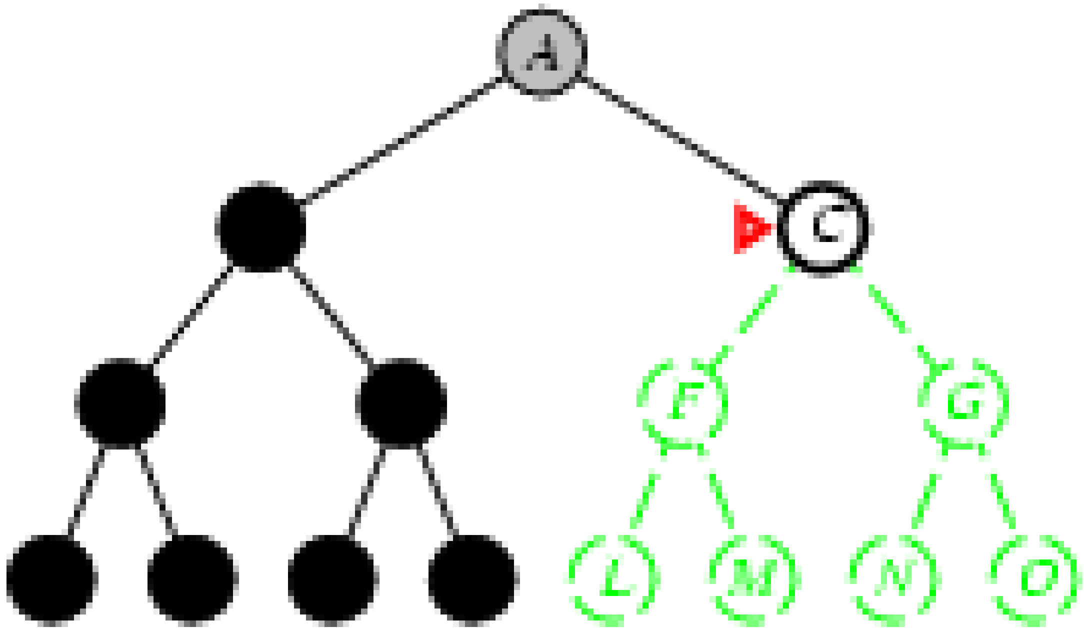
# Depth-first search

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



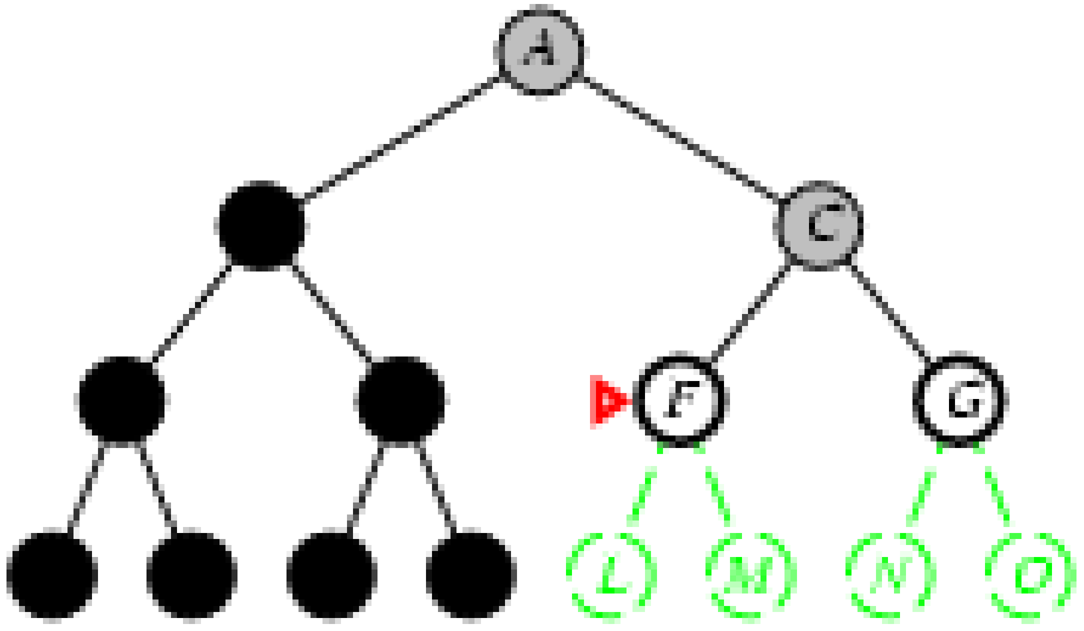
# Depth-first search

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



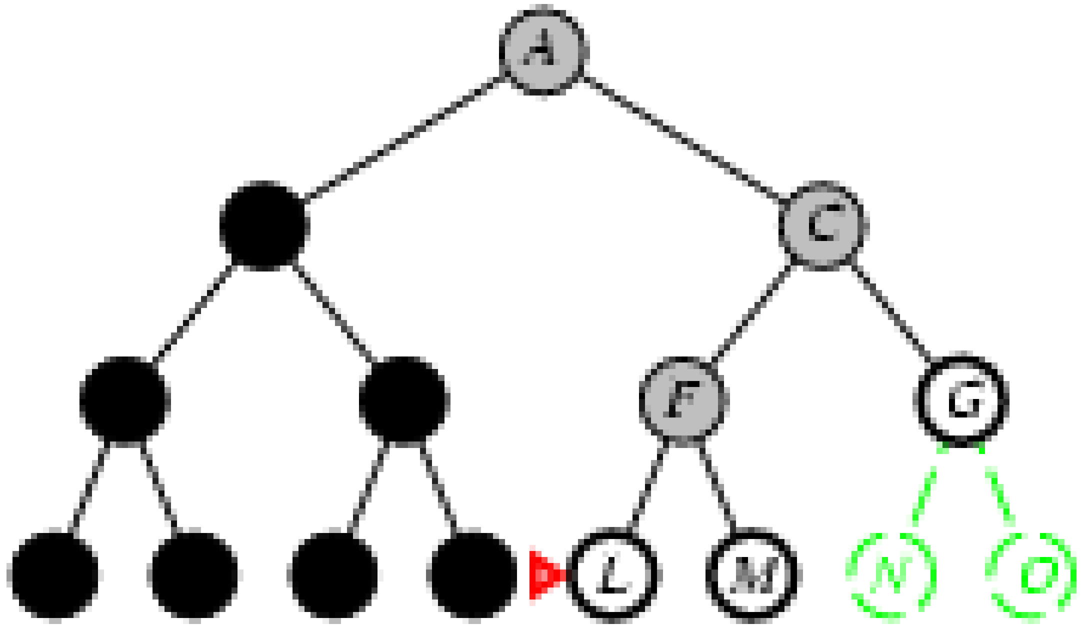
# Depth-first search

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



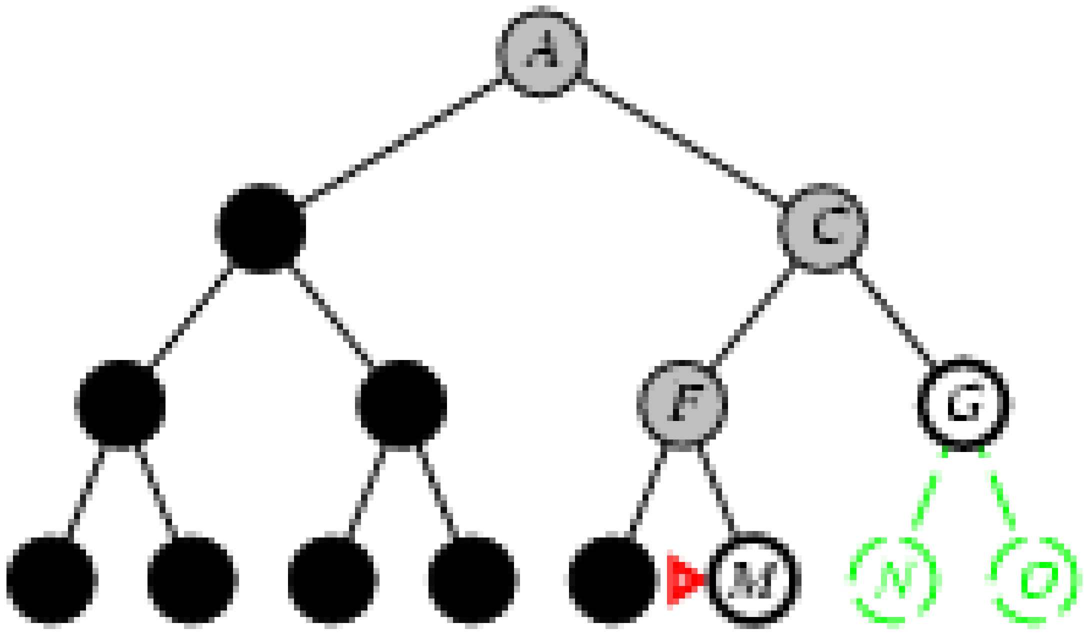
# Depth-first search

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



# Depth-first search

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



## Properties of depth-first search

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

→ complete in finite spaces

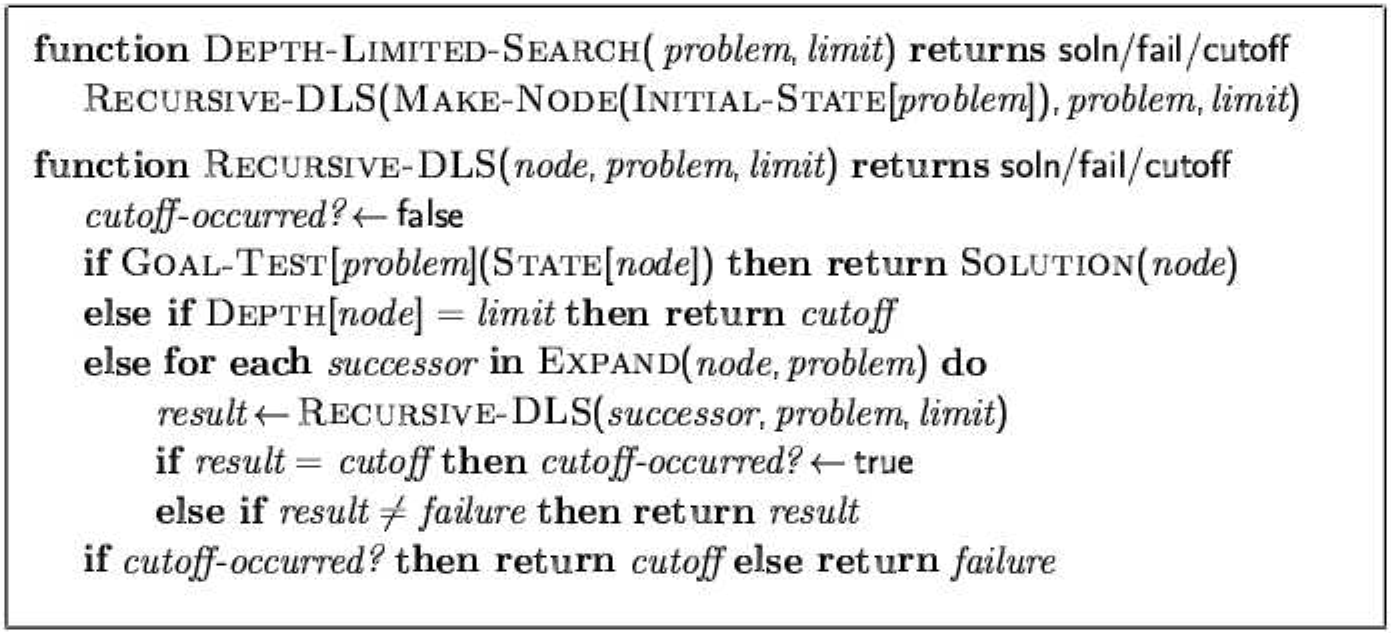
- Time? *O(bm)*: terrible if *m* is much larger than *d*
  - but if solutions are dense, may be much faster than breadth-first
- Space? *O(bm),* i.e., linear space!
- Optimal? No

# Depth-limited search

= depth-first search with depth limit *l*,

i.e., nodes at depth *l* have no successors

- Recursive implementation:



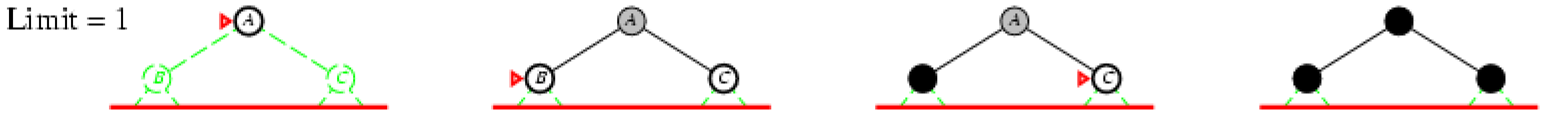
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

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

## Iterative deepening search

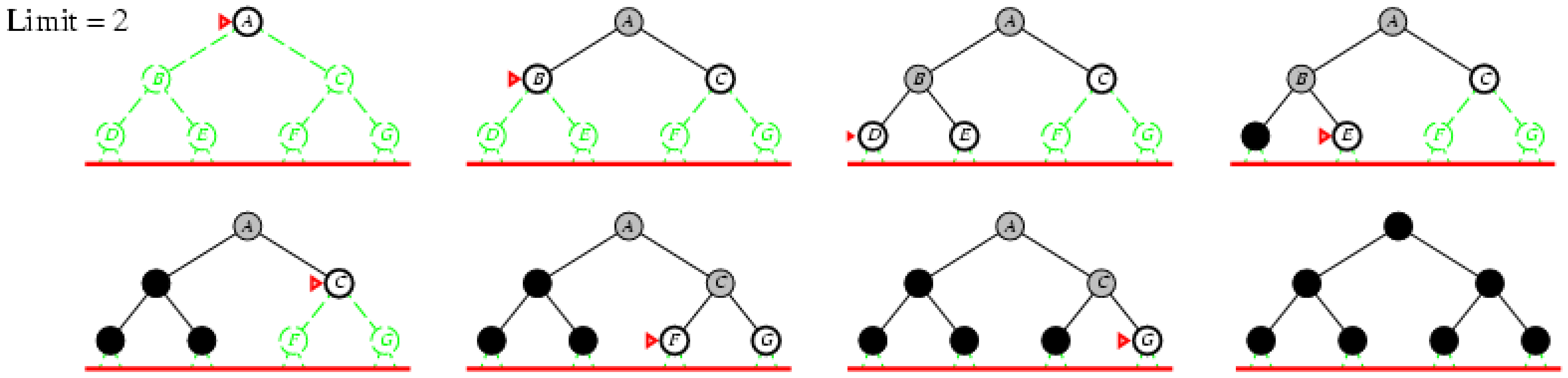
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    inputs: problem, a problem

    for depth ← 0 to $\infty$ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result $\neq$ cutoff then return result
```

## Iterative deepening search *l* =0

# Iterative deepening search *l* =1

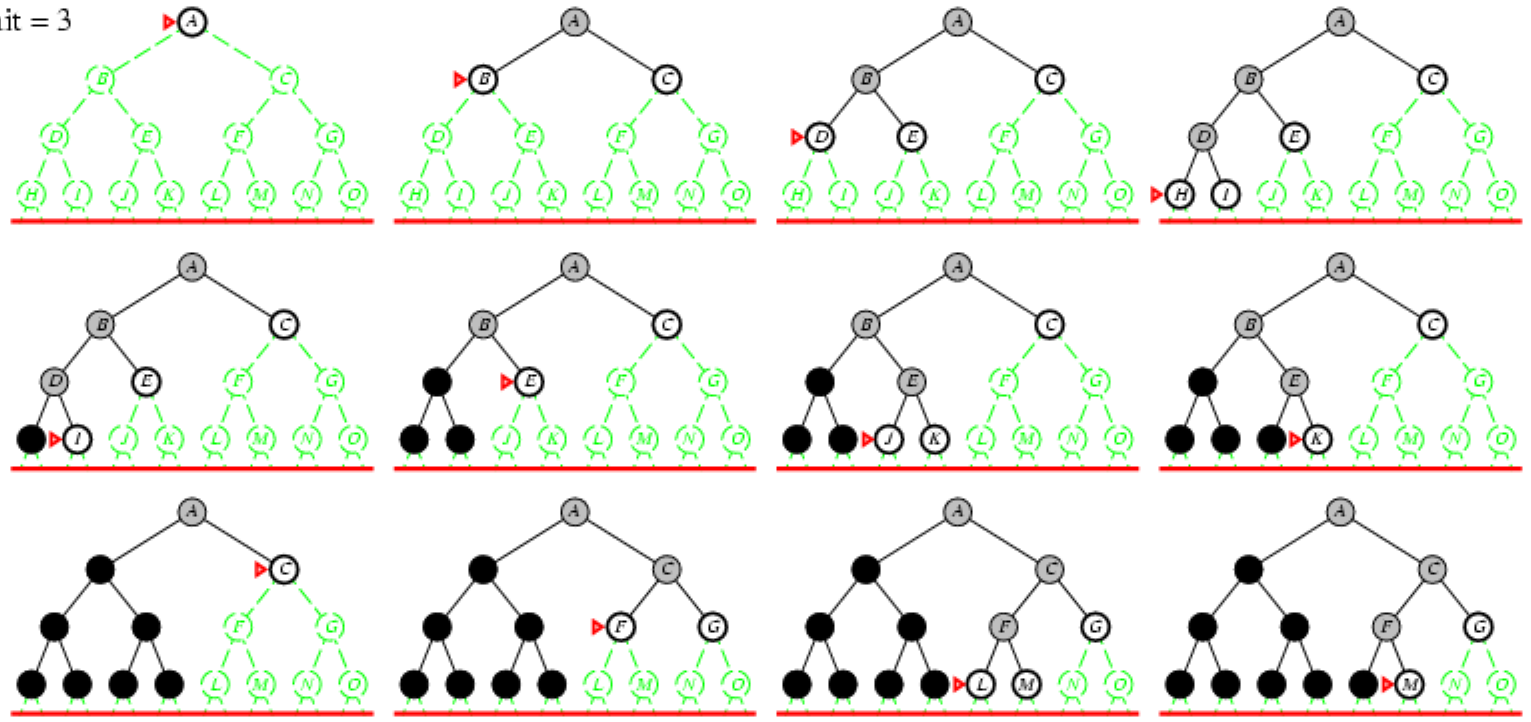


# Iterative deepening search *l* =2



# Iterative deepening search *l* =3

Limit = 3



# Iterative deepening search

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

  $N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

NIDS = (d+1)b0 + d b^1 + (d-1)b^2 + … + 3bd-2 +2bd-1 + 1bd

- For $b = 10$, $d = 5$,
  - NDLS = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
  - NIDS = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456

- Overhead = (123,456 - 111,111)/111,111 = 11%

## Properties of iterative deepening search

- Complete? Yes
- Time? *(d+1)b0 + d b1 + (d-1)b2 + … + bd = O(bd)*
- Space? *O(bd)*
- Optimal? Yes, if step cost = 1

## Summary of algorithms

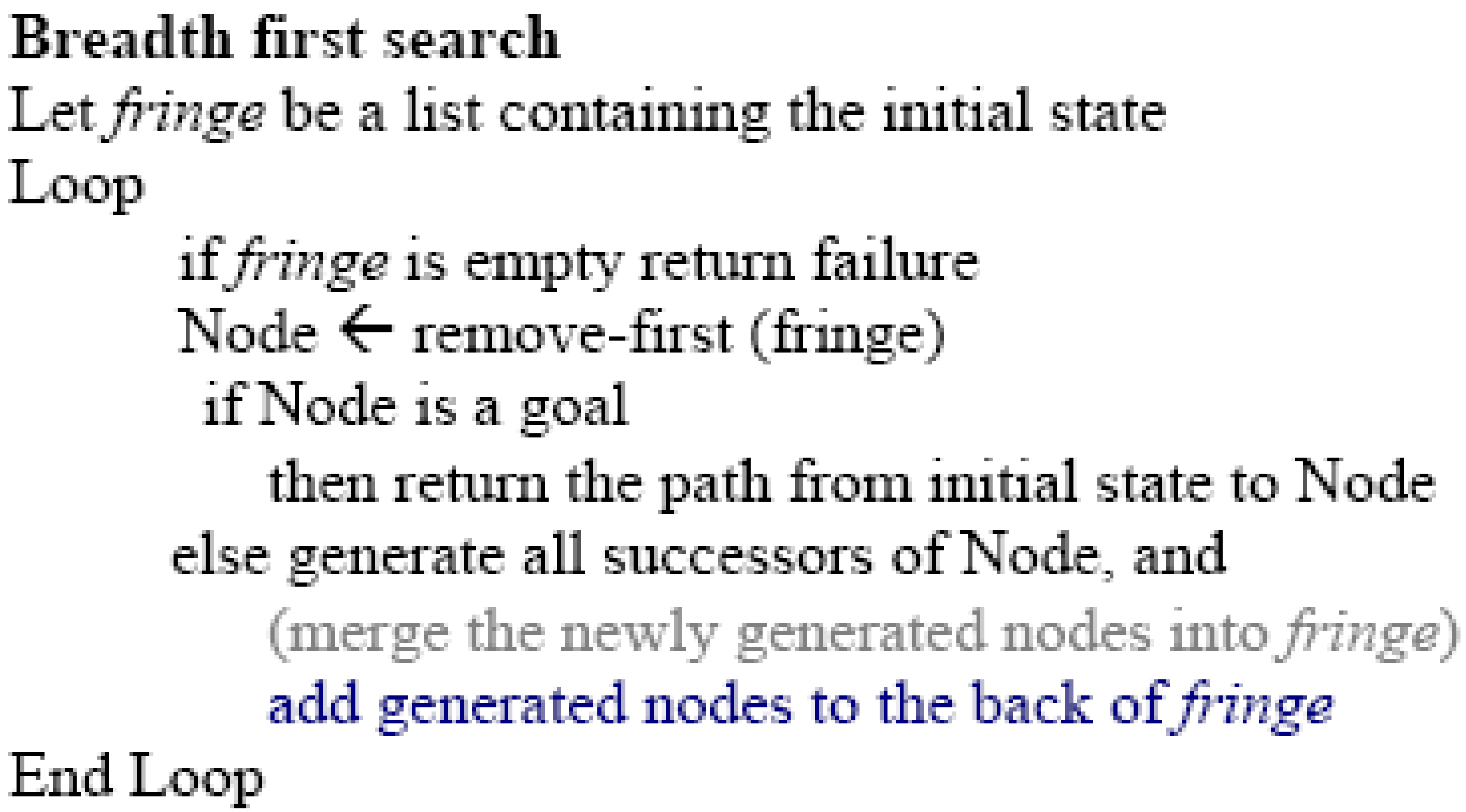
| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening |
|---|---|---|---|---|---|
| Complete? | Yes | Yes | No | No | Yes |
| Time | $O(b^{d+1})$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(b^m)$ | $O(b^l)$ | $O(b^d)$ |
| Space | $O(b^{d+1})$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(bm)$ | $O(bl)$ | $O(bd)$ |
| Optimal? | Yes | Yes | No | No | Yes |

# Uninformed Search Techniques
## BFS and DFS

# Uninformed Search

- Breadth First Search
- Uniform Cost Search
- Depth First Search
- Depth Limited Search
- Iterative deepening Depth First Search
- Bidirectional Search

# Breadth First Search



**Breadth first search**

```
Let *fringe* be a list containing the initial state
Loop
        if *fringe* is empty return failure
        Node ← remove-first (fringe)
         if Node is a goal
            then return the path from initial state to Node
        else generate all successors of Node, and
            (merge the newly generated nodes into *fringe*)
            add generated nodes to the back of *fringe*
End Loop
```

## Concept

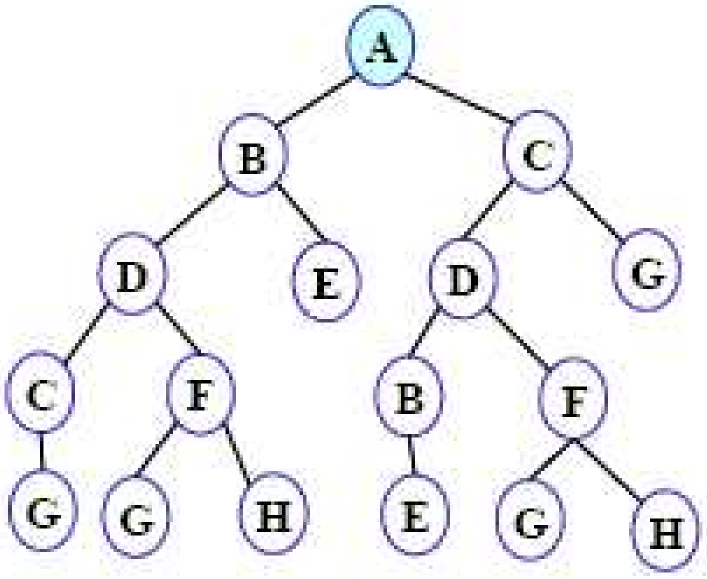
**Step 1:** Traverse the root node

**Step 2:** Traverse all neighbours of root node.

**Step 3:** Traverse all neighbours of neighbours of the root node.

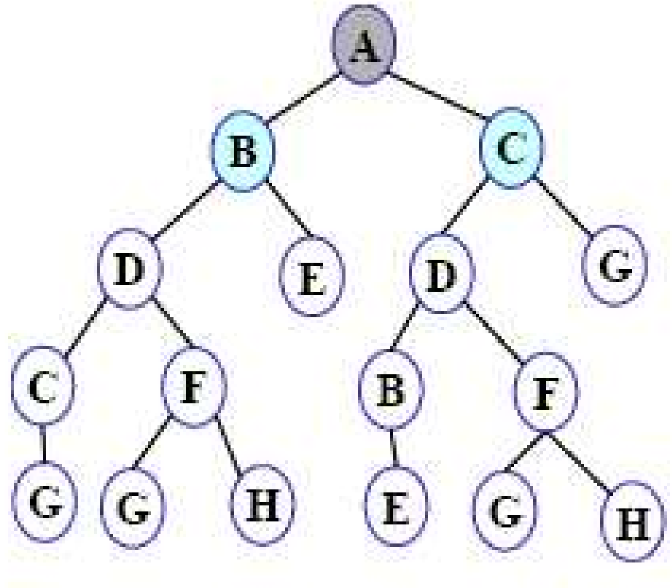
**Step 4:** This process will continue until we are getting the goal node.

Step 1: Initially fringe contains only one node corresponding to the source state A.



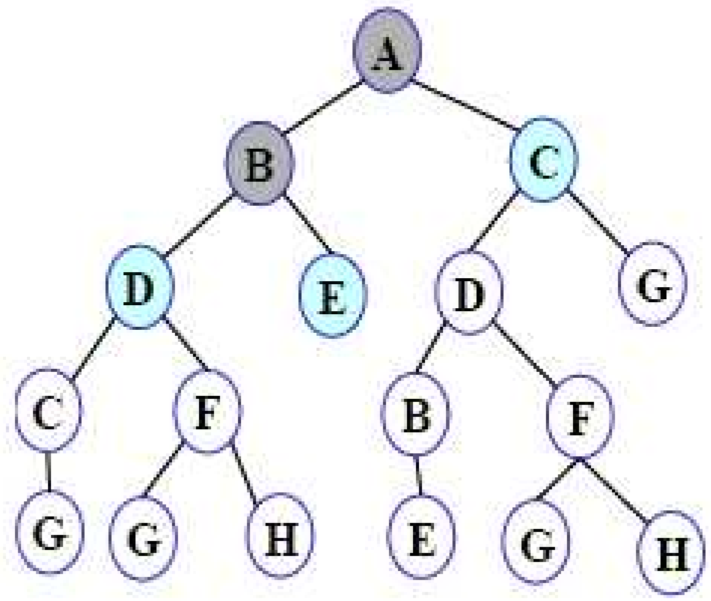
FRINGE: A

Step 2: A is removed from fringe. The node is expanded, and its children B and C are generated. They are placed at the back of fringe.



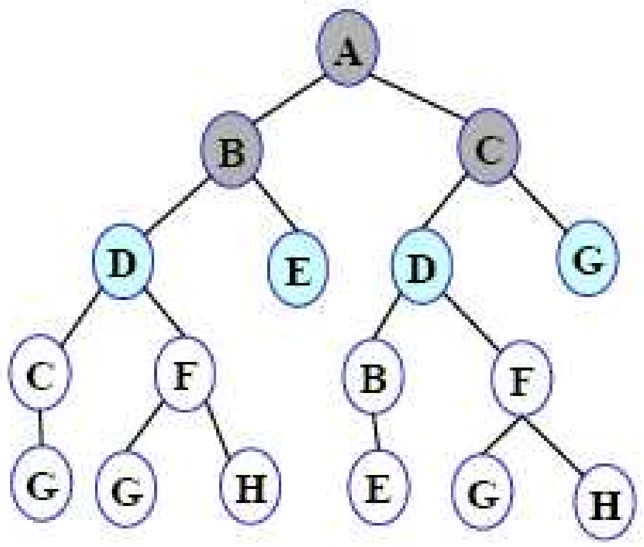
FRINGE: B C

Step 3: Node B is removed from fringe and is expanded. Its children D, E are generated and put at the back of fringe.



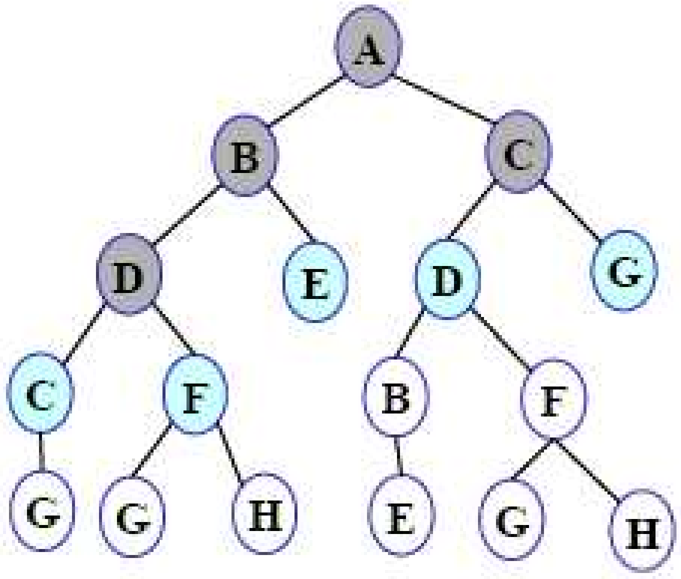
FRINGE: C D E

Step 4: Node C is removed from fringe and is expanded. Its children D and G are added to the back of fringe.



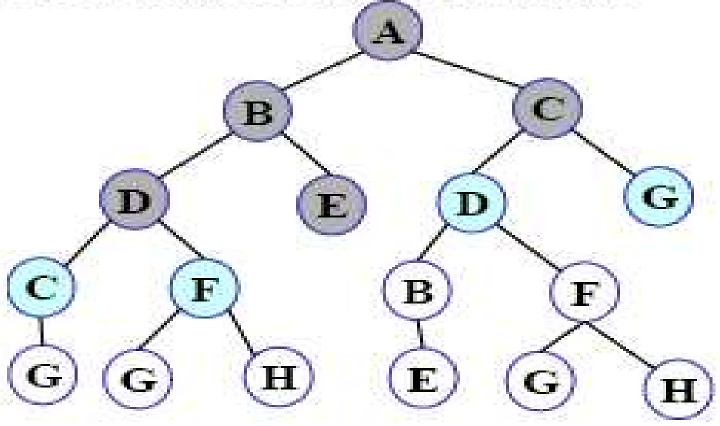
FRINGE: D E D G

Step 5: Node D is removed from fringe. Its children C and F are generated and added to the back of fringe.



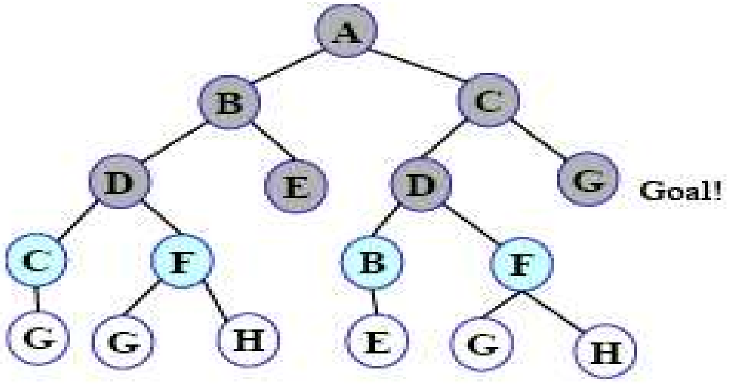
FRINGE: E D G C F

Step 6: Node E is removed from fringe. It has no children.



FRINGE: D G C F

Step 7: D is expanded, B and F are put in OPEN.



FRINGE: G C F B F

Step 8: G is selected for expansion. It is found to be a goal node. So the algorithm returns the path A C G by following the parent pointers of the node corresponding to G. The algorithm terminates.

## Properties of BFS

- Complete.
- The algorithm is optimal (i.e., admissible) if all operators have the same cost. Otherwise, breadth first search finds a solution with the shortest path length.
- The algorithm has exponential time and space complexity. Suppose the search tree can be modelled as a b-ary tree as shown in Figure 3. Then the time and space complexity of the algorithm is O(bd+1) where d is the depth of the solution and b is the branching factor (i.e., number of children) at each node.

**Advantages**

- In this procedure at any way it will find the goal.
- It does not follow a single unfruitful path for a long time.
- It finds the minimal solution in case of multiple paths.

**Disadvantages**

- BFS consumes large memory space.
- Its time complexity is more.
- It has long pathways, when all paths to a destination are on approximately the same search depth.

# Depth First Search

```
Depth First Search
Let fringe be a list containing the initial state
Loop
    if fringe is empty return failure
    Node ← remove-first (fringe)
     if Node is a goal
        then return the path from initial state to Node
    else generate all successors of Node, and
        merge the newly generated nodes into fringe
        add generated nodes to the front of fringe
End Loop
```

## Concept

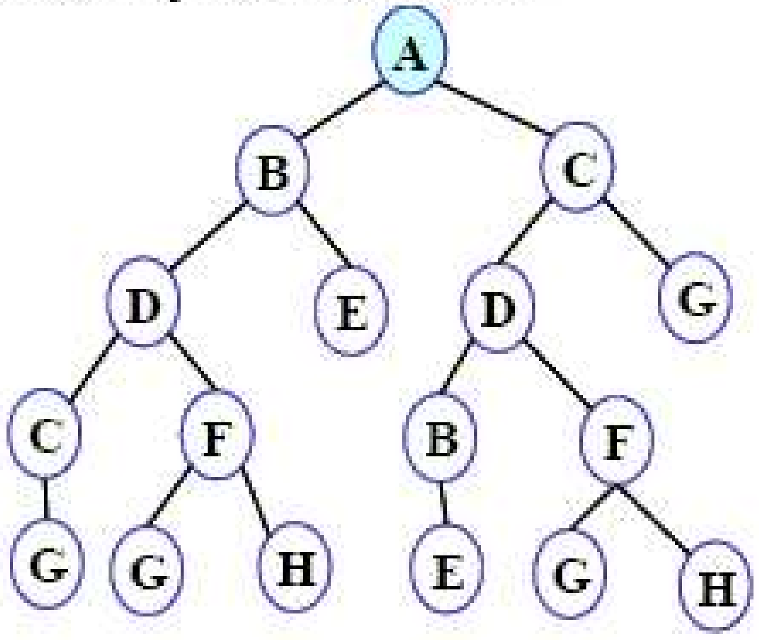
**Step 1:** Traverse the root node.

**Step 2:** Traverse any neighbour of the root node.

**Step 3:** Traverse any neighbour of neighbour of the root node.

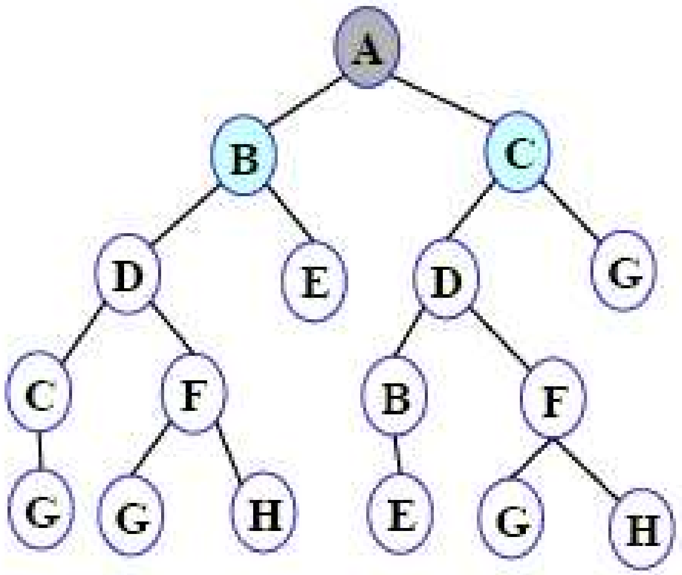
**Step 4:** This process will continue until we are getting the goal node.

Step 1: Initially fringe contains only the node for A.



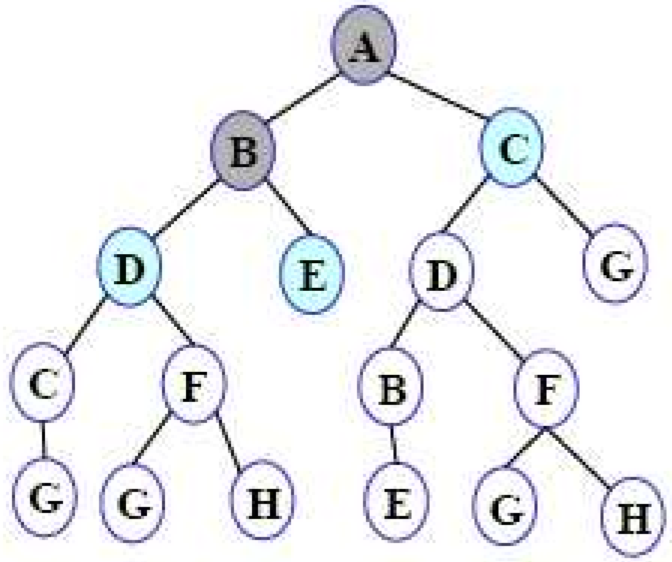
FRINGE: A

Step 2: A is removed from fringe. A is expanded and its children B and C are put in front of fringe.



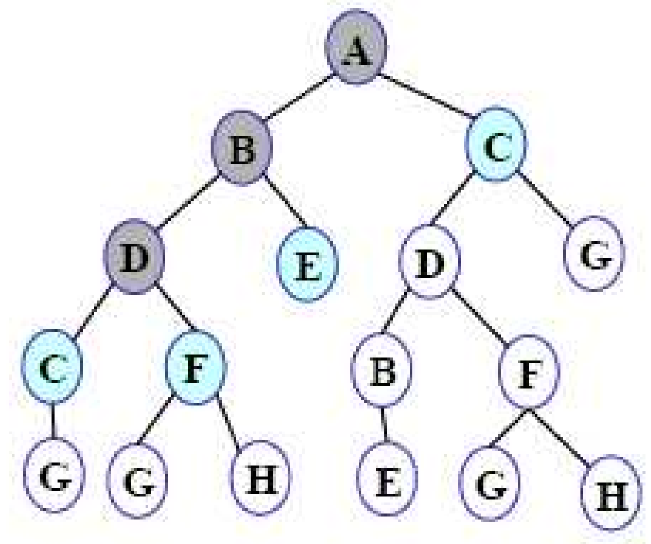
FRINGE: B C

Step 3: Node B is removed from fringe, and its children D and E are pushed in front of fringe.



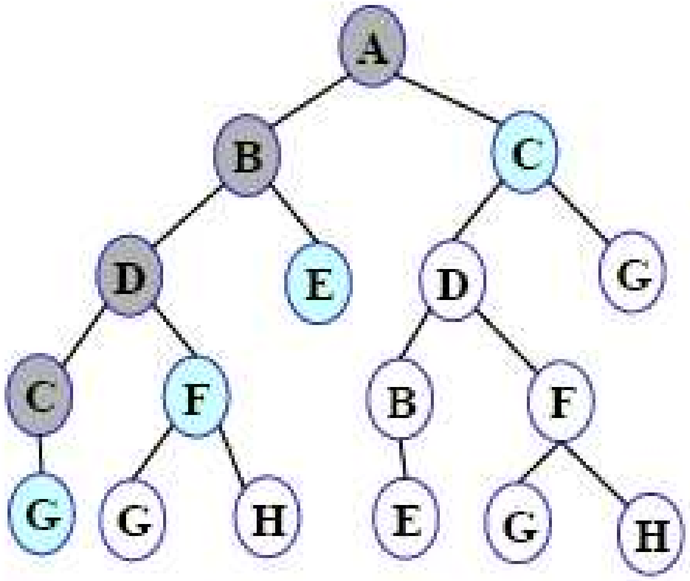
FRINGE: D E C

Step 4: Node D is removed from fringe. C and F are pushed in front of fringe



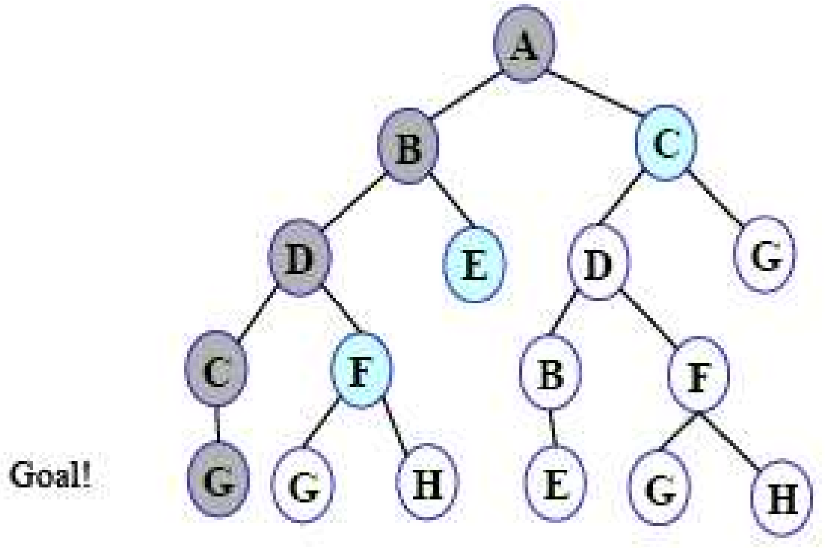
FRINGE: C F E C

Step 5: Node C is removed from fringe. Its child G is pushed in front of fringe.



FRINGE: G F E C

Step 6: Node G is expanded and found to be a goal node. The solution path A-B-D-C-G is returned and the algorithm terminates.



FRINGE: G F E C

## Properties of DFS

- The algorithm takes exponential time. If N is the maximum depth of a node in the search space, in the worst case the algorithm will take time O(bd). However the space taken is linear in the depth of the search tree, O(bN).

- Note that the time taken by the algorithm is related to the maximum depth of the search tree. If the search tree has infinite depth, the algorithm may not terminate. This can happen if the search space is infinite. It can also happen if the search space contains cycles. The latter case can be handled by checking for cycles in the algorithm. Thus Depth First Search is not complete.

**Advantages:**

- DFS consumes very less memory space.
- It will reach at the goal node in a less time period than BFS if it traverses in a right path.
- It may find a solution without examining much of search because we may get the desired solution
- in the very first go.

**Disadvantages:**

- It is possible that may states keep reoccurring.
- There is no guarantee of finding the goal node.
- Sometimes the states may also enter into infinite loops.

## Difference between BFS and DFS

**BFS**

- It uses the data structure queue.
- BFS is complete because it finds the solution if one exists.
- BFS takes more space i.e. equivalent to O(bd+1) where b is the maximum breath exist in a search
- tree and d is the maximum depth exit in a search tree.
- In case of several goals, it finds the best one.

**DFS**

- It uses the data structure stack.
- It is not complete because it may take infinite loop to reach at the goal node.
- The space complexity is O (d).
- In case of several goals, it will terminate the solution in any order.

## Uniform Cost Search

- This algorithm is by Dijkstra [1959]. The algorithm expands nodes in the order of their cost from the source.
- In uniform cost search the newly generated nodes are put in OPEN according to their path costs. This ensures that when a node is selected for expansion it is a node with the cheapest cost among the nodes in OPEN.
- Let g(n) = cost of the path from the start node to the current node n. Sort nodes by increasing value of g.

# Algorithm- Uniform Cost Search

1. **Initialize**: set OPEN=s, CLOSED={} and c(s)=0
2. **Fail:** If OPEN={}, terminate with failure
3. **Select:** Select a state with the minimum cost ,n, from OPEN and save in CLOSED
4. **Terminate**: If n∈G, terminate with success
5. **Expand:** generate the successor of n

```
For each successor, m, :
If m∈[OPEN∪CLOSED]
    set C(m)= C(n)+C(n,m) and insert m in OPEN
If m∈[OPEN∪CLOSED]
Set C(m)= min{ C(m),C(n)+C(n,m)}
```

## Properties of UCS

Some properties of this search algorithm are:

- Complete
- Optimal/Admissible
- Exponential time and space complexity, O(bd)

## Uninformed Search

## Building Goal-Based Agents

- We have a **goal** to reach
  - Driving from point A to point B
  - Put 8 queens on a chess board such that no one attacks another
  - Prove that John is an ancestor of Mary
- We have information about the current **state**, where we are now at the beginning and after each action
- We have a set of **actions** we can take to move around (change from where we are) if the preconditions are met
- **Objective**: find a sequence of legal actions which will bring us from the start point to a goal

## What is the goal to be achieved?

- Could describe a situation we want to achieve, a set of properties that we want to hold, etc.
- Requires defining a **“goal test”** so that we know what it means to have achieved/satisfied our goal.
- This is a hard part that is rarely tackled in AI, usually assuming that the system designer or user will specify the goal to be achieved.

## What are the actions?

- Quantify all of the primitive actions or events that are sufficient to describe all necessary changes in solving a task/goal.
- No uncertainty associated with what an action does to the world. That is, given an action (aka operator or move) and a description of the current state of the world, the action completely specifies
  - **Precondition:** if that action CAN be applied to the current world (i.e., is it applicable and legal), and
  - **Effect:** what the exact state of the world will be after the action is performed in the current world (i.e., no need for "history" information to be able to compute what the new world looks like).

# Actions

- Note also that actions can all be considered as **discrete events** that can be thought of as occurring at an **instant of time**.
  - That is, the world is in one situation, then an action occurs and the world is now in a new situation. For example, if "Mary is in class" and then performs the action "go home," then in the next situation she is "at home." There is no representation of a point in time where she is neither in class nor at home (i.e., in the state of "going home").
- The number of operators needed depends on the **representation** used in describing a state.
- Actions often are associated with **costs**

## Representing states

- At any moment, the relevant world is represented as a **state**
  - **Initial (start) state**: S
  - An action (or an operation) changes the current state to another state (if it is applied): state transition
  - An action can be taken (**applicable**) only if the its precondition is met by the current state
  - For a given state, there might be **more than one** applicable actions
  - **Goal state**: a state satisfies the goal description or passes the goal test
  - Dead-end state: a non-goal state to which no action is applicable

## Representing states

- **Stat space**:
  - Includes the initial state S and all other states that are reachable  from S by a sequence of actions
  - A state space can be organized as a graph:

  nodes: states in the space

  arcs: actions/operations

- The **size of a problem** is usually described in terms of the **number of states** (or the size of the state space) that are possible.
  - Tic-Tac-Toe has about 3^9 states.
  - Checkers has about 10^40 states.
  - Rubik's Cube has about 10^19 states.
  - Chess has about 10^120 states in a typical game.
  - GO has more states than Chess

## Closed World Assumption

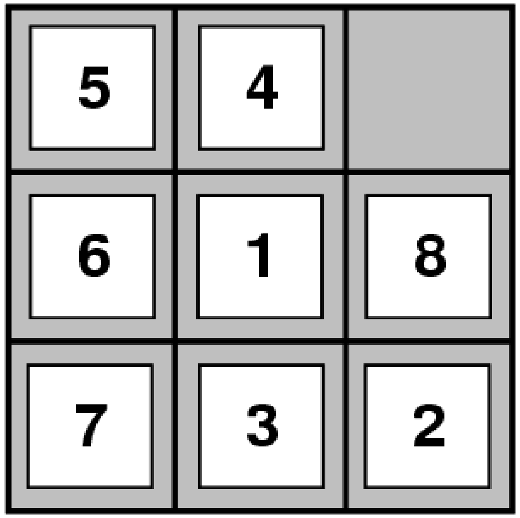
- We will generally use the **Closed World Assumption**.
- All necessary information about a problem domain is available in each percept so that each state is a complete description of the world.
- There is no incomplete information at any point in time.

## Some example problems

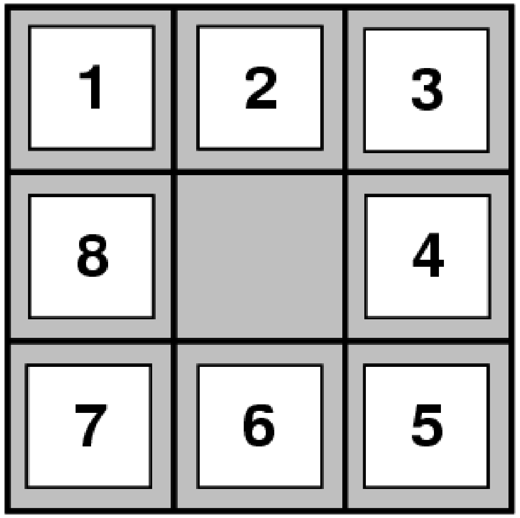
- Toy problems and micro-worlds
  - 8-Puzzle
  - Missionaries and Cannibals
  - Cryptarithmetic
  - Remove 5 Sticks
  - Traveling Salesman Problem (TSP)
- Real-world-problems

# 8-Puzzle

**Given an initial configuration of 8 numbered tiles on a 3 x 3 board, move the tiles in such a way so as to produce a desired goal configuration of the tiles.**



Start State

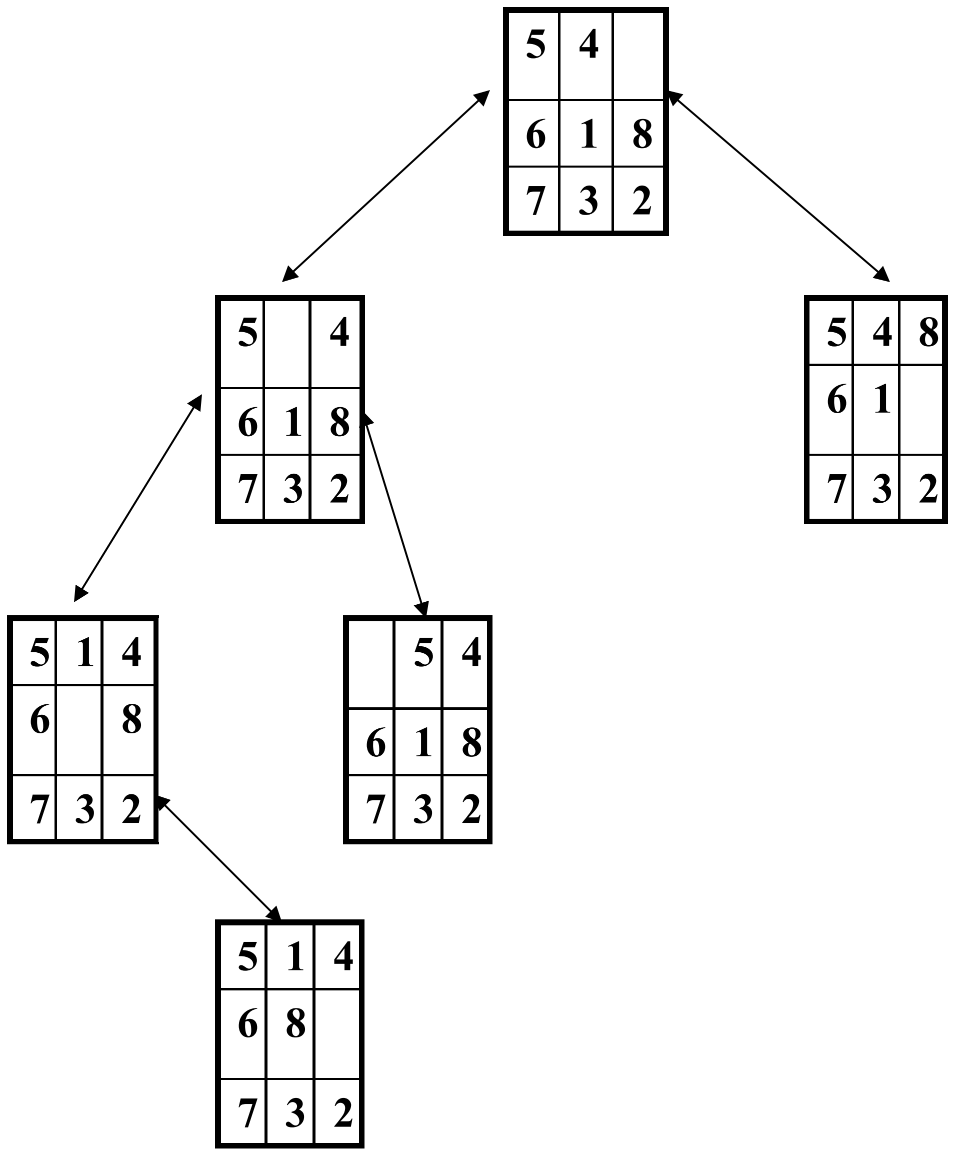


Goal State

# 8 puzzle

- **State:** 3 x 3 array configuration of the tiles on the board.
- **Operators:** Move Blank square Left, Right, Up or Down.
  - This is a more efficient encoding of the operators than one in which each of four possible moves for each of the 8 distinct tiles is used.
- **Initial State:** A particular configuration of the board.
- **Goal:** A particular configuration of the board.
- The state space is partitioned into two subspaces
- NP-complete problem, need to examine O(2^k) states where k is the length of the solution path.
- 15-puzzle problems (4 x 4 grid with 15 numbered tiles), and N-puzzles (N = n^2-1)

## A portion of the state space of a 8-Puzzle problem



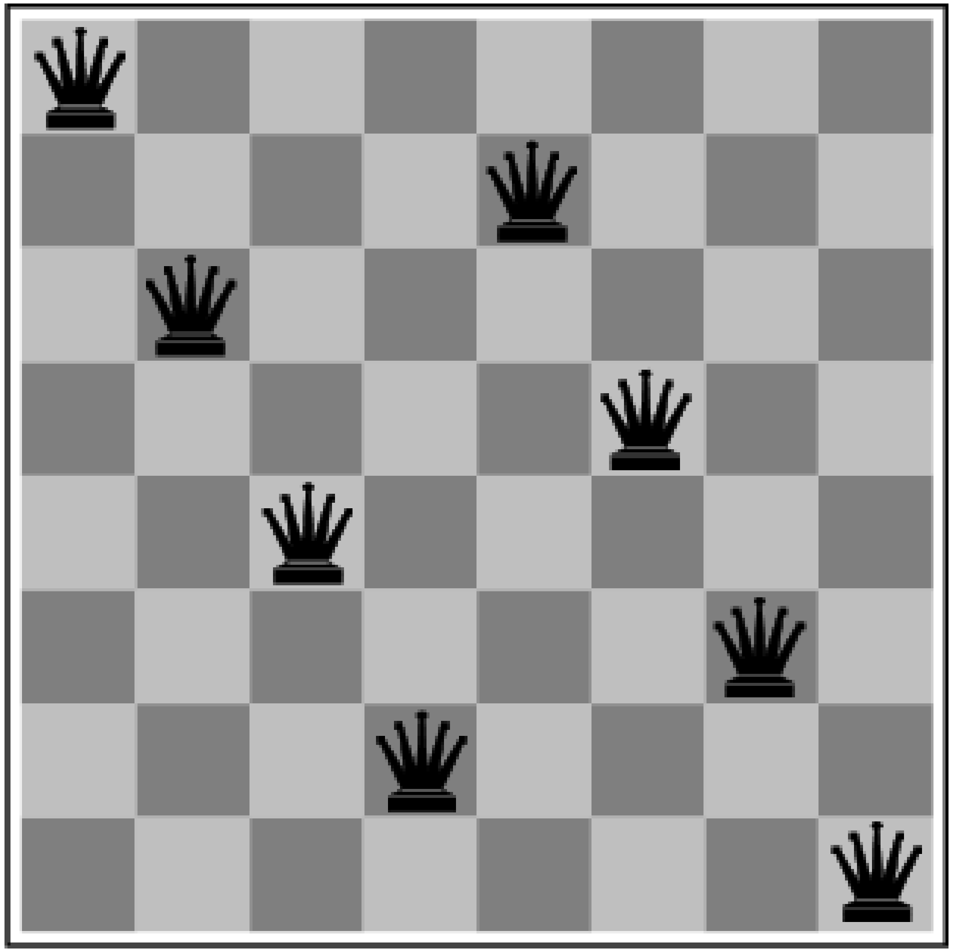
## The 8-Queens Problem

Place eight queens on a chessboard such that no queen attacks any other!

Total # of states: 4.4x10^9

Total # of solutions:

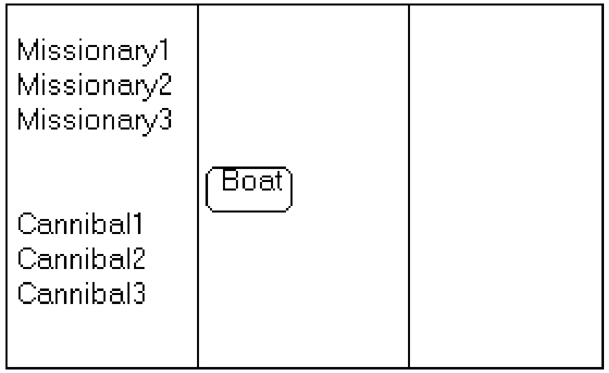
12 (or 96)



# Missionaries and Cannibals

**There are 3 missionaries, 3 cannibals, and 1 boat that can carry up to two people on one side of a river.**

- **Goal**: Move all the missionaries and cannibals across the river.
- **Constraint:** Missionaries can never be outnumbered by cannibals on either side of river, or else the missionaries are killed.
- **State:** configuration of missionaries and cannibals and boat on each side of river.
- **Operators:** Move boat containing 1 or 2 occupants across the river (in either direction) to the other side.



3 Missionaries and 3 Cannibals wish to cross the river. They have a boat that will carry two people. Everyone can navigatethe boat. If at any time the Cannibals outnumber the missionaries on either bank of the river, they will eat the Missionaries. Find the smallest number of crossings that will allow everyone to cross the river safely.
The problem can be solved in 11 moves. But people rarely get the optimal solution, because the MC problem contains a 'tricky' state at the end, where two people move back across the river.

# Missionaries and Cannibals Solution

| | ***Near side*** | | ***Far side*** | |
|---|---|---|---|---|
| 0 Initial setup: | MMMCCC | B | | - |
| 1 Two cannibals cross over: | MMMC | | B | CC |
| 2 One comes back: | MMMCC | B | | C |
| 3 Two cannibals go over again: | MMM | | B | CCC |
| 4 One comes back: | MMMC | B | | CC |
| 5 Two missionaries cross: | MC | | B | MMCC |
| 6 A missionary & cannibal return: | MMCC | B | | MC |
| 7 Two missionaries cross again: | CC | | B | MMMC |
| 8 A cannibal returns: | CCC | B | | MMM |
| 9 Two cannibals cross: | C | | B | MMMCC |
| 10 One returns: | CC | B | | MMMC |
| 11 And brings over the third: | - | | B | MMMCCC |

# Cryptarithmetic

- Find an assignment of digits (0, ..., 9) to letters so that a given arithmetic expression is true. examples: SEND + MORE = MONEY and

```
 FORTY     Solution:  29786
+  TEN                  850
+  TEN                  850
 -----                -----
 SIXTY                31486
F=2, O=9, R=7, etc.
```

- Note: In this problem, the solution is NOT a sequence of actions that transforms the initial state into the goal state, but rather the solution is simply finding a goal node that includes an assignment of digits to each of the distinct letters in the given problem

## Remove 5 Sticks

- Given the following configuration of sticks, remove exactly 5 sticks in such a way that the remaining configuration forms exactly 3 squares.

## Traveling Salesman Problem

- Given a road map of n cities, find the **shortest** tour which visits every city on the map exactly once and then return to the original city (*Hamiltonian circuit*)
- (Geometric version):
  - a complete graph of n vertices.
  - n!/2n legal tours
  - Find one legal tour that is shortest

## Formalizing Search in a State Space

- A state space is a **graph**, (V, E) where V is a set of **nodes** and E is a set of **arcs**, where each arc is directed from a node to another node
- **node:** corresponds to a **state**
  - state description
  - plus optionally other information related to the parent of the node, operation to generate the node from that parent, and other bookkeeping data)
- **arc:** corresponds to an applicable action/operation.
  - the source and destination nodes are called as **parent (immediate predecessor)** and **child** (**immediate successor**) nodes with respect to each other
  - ancestors( (predecessors) and descendents (successors)
  - each arc has a *fixed, non-negative* **cost** associated with it, corresponding to the cost of the action

- **node generation:** making explicit a node by applying an action to another node which has been made explicit
- **node expansion:** generating **all** children of an explicit node by applying **all** applicable operations to that node
- One or more nodes are designated as **start nodes**
- A **goal test** predicate is applied to a node to determine if its associated state is a goal state
- A **solution** is a sequence of operations that is associated with a path in a state space from a start node to a goal node
- The **cost of a solution** is the sum of the arc costs on the solution path

- **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 to include a goal node.
  - Hence, initially V={S}, where S is the start node; when S is expanded, its successors are generated and those nodes are added to V and the associated arcs are added to E. This process continues until a goal node is generated (included in V) and identified (by goal test)
- During search, a node can be in one of the three categories:
  - Not generated yet (has not been made explicit yet)
  - **OPEN**: generated but not expanded
  - **CLOSED**: expanded
  - Search strategies differ mainly on **how to select an OPEN node** for expansion at each step of search

# A General State-Space Search Algorithm

- Node n
  - state description
  - parent (may use a backpointer) (if needed)
  - Operator used to generate n (optional)
  - Depth of n (optional)
  - Path cost from S to n (if available)
- **OPEN** list
  - initialization: {S}
  - node insertion/removal depends on specific search strategy
- **CLOSED** list
  - initialization: {}
  - organized by backpointers

# A General State-Space Search Algorithm

```
open := {S}; closed :={};
repeat
    n := select(open);       /* select one node from open for expansion */
       if n is a goal
             then exit with success;  /* delayed goal testing */
       expand(n)
              /* generate all children of n
                  put these newly generated nodes in open (check duplicates)
                  put n in closed (check duplicates) */
until open = {};
exit with failure
```

## Some Issues

- Search process constructs a search tree, where
  - **root** is the initial state S, and
  - **leaf nodes** are nodes
    - not yet been expanded (i.e., they are in OPEN list) or
    - having no successors (i.e., they're "deadends")
- Search tree may be infinite because of loops even if state space is small
- Search strategies mainly differ on *select*(open)
- Each node represents a *partial solution path* (and cost of the partial solution path) from the start node to the given node.
  - in general, from this node there are many possible paths (and therefore solutions) that have this partial path as a prefix.

## Evaluating Search Strategies

- **Completeness**
  - Guarantees finding a solution whenever one exists
- **Time Complexity**
  - How long (worst or average case) does it take to find a solution? Usually measured in terms of the **number of nodes expanded**
- **Space Complexity**
  - How much space is used by the algorithm? Usually measured in terms of the **maximum size that the “OPEN" list** becomes during the search
- **Optimality/Admissibility**
  - If a solution is found, is it guaranteed to be an optimal one? For example, is it the one with minimum cost?

## Uninformed vs. Informed Search

- **Uninformed Search Strategies**
  - Breadth-First search
  - Depth-First search
  - Uniform-Cost search
  - **Depth-First Iterative Deepening search**
- **Informed Search Strategies**
  - Hill climbing
  - Best-first search
  - Greedy Search
  - Beam search
  - Algorithm A
  - **Algorithm A***

## Breadth-First

- Algorithm outline:
  - Always select from the OPEN the node with the **smallest depth** for expansion, and put all newly generated nodes into OPEN
  - OPEN is organized as **FIFO** (first-in, first-out) list, i.e., a **queue**.
  - Terminate if a node selected for expansion is a goal
- **Properties**
  - **Complete**
  - **Optimal** (i.e., admissible) if all operators have the same cost. Otherwise, not optimal but finds solution with **shortest path length** (shallowest solution).
  - **Exponential time and space complexity**,

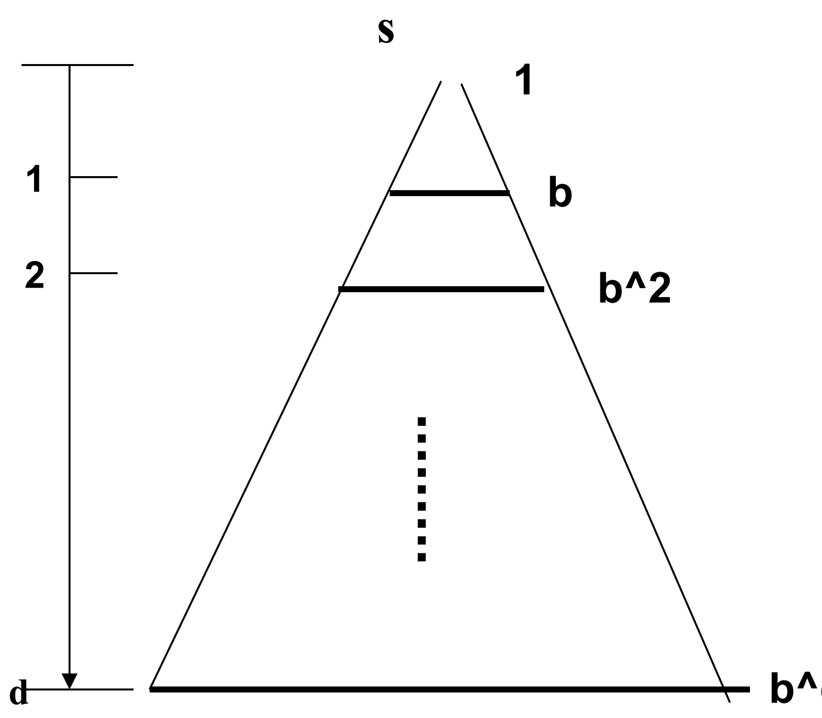
  O(b^d) nodes will be generated, where

  d is the depth of the solution and

  b is the branching factor (i.e., number of children) at each node

# Breadth-First

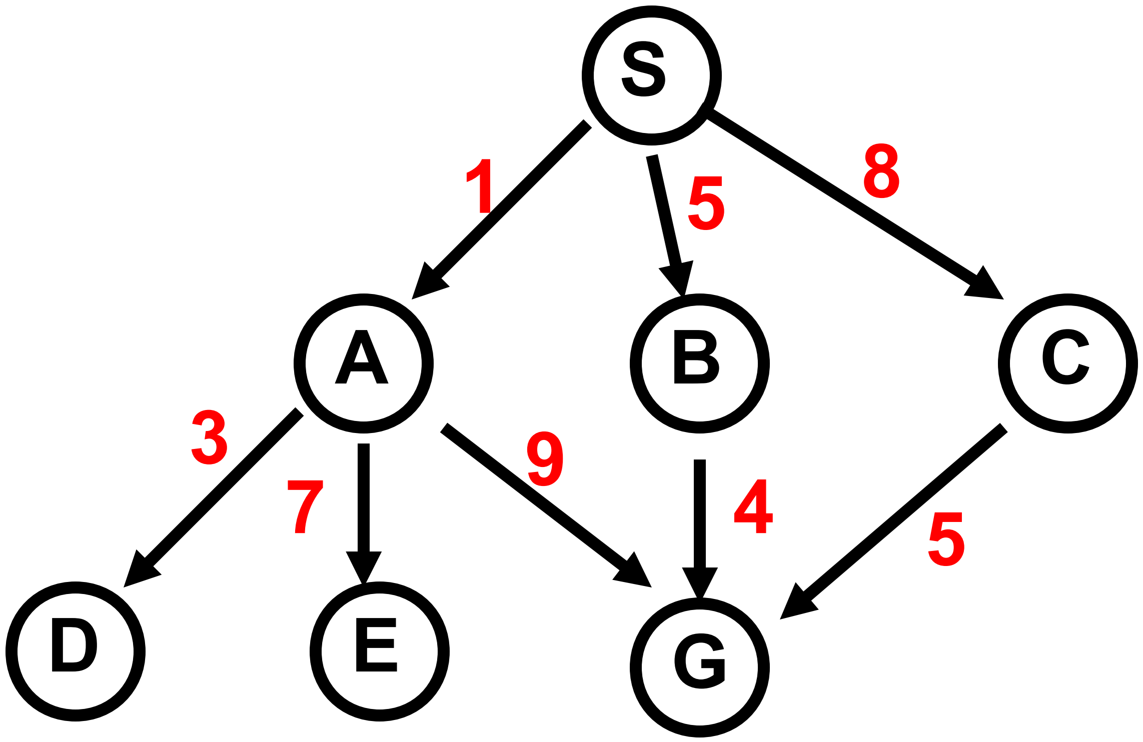
- A complete search tree of depth d where each non-leaf node has b children, has a total of 1 + b + b^2 + ... + b^d = (b^(d+1) - 1)/(b-1) nodes
- Time complexity (# of nodes generated): O(b^d)
- Space complexity (maximum length of OPEN): O(b^d)



- For a complete search tree of depth 12, where every node at depths 0, ..., 11 has 10 children and every node at depth 12 has 0 children, there are 1 + 10 + 100 + 1000 + ... + 10^12 = (10^13 - 1)/9 = O(10^12) nodes in the complete search tree.

• BFS is suitable for problems with shallow solutions

## Example Illustrating Uninformed Search Strategies



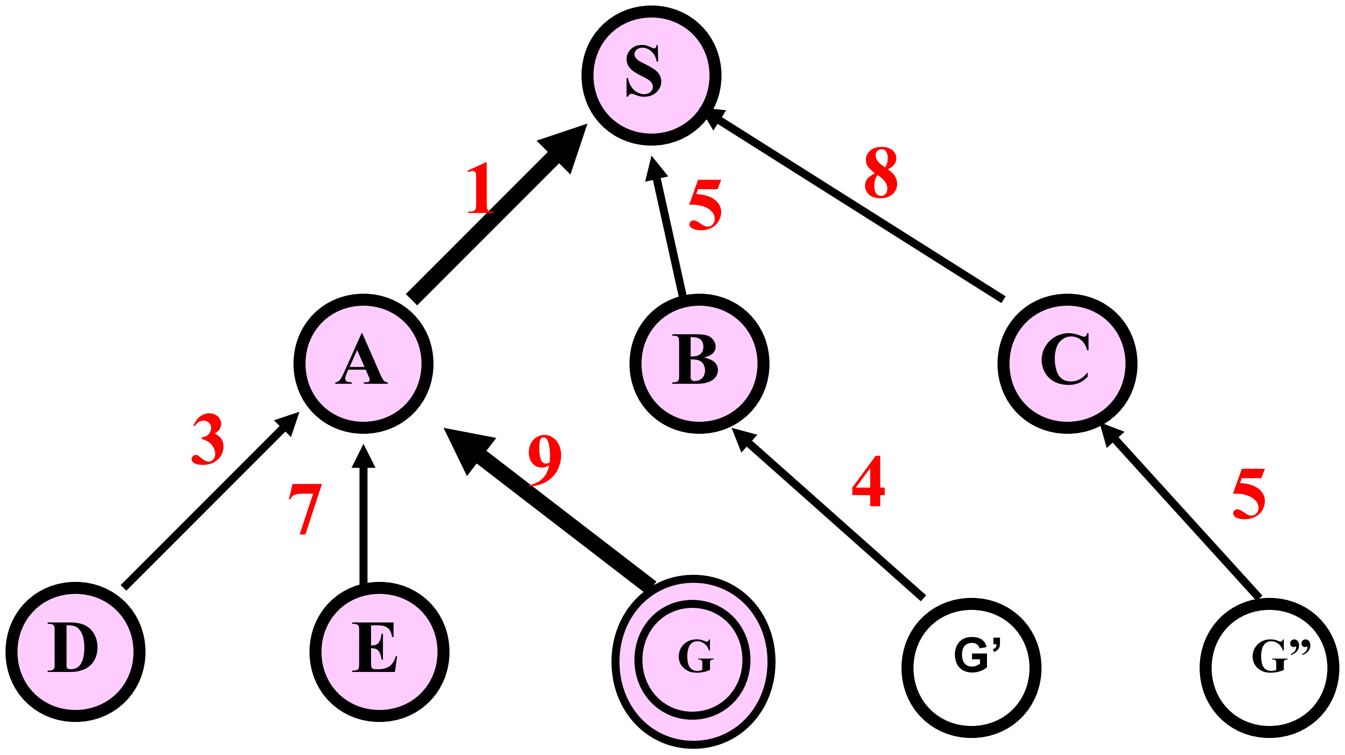
# Breadth-First Search

| exp. node | OPEN list | CLOSED list |
|---|---|---|
| | { S } | {} |
| S | { A B C } | {S} |
| A | { B C D E G } | {S A} |
| B | { C D E G G' } | {S A B} |
| C | { D E G G' G" } | {S A B C} |
| D | { E G G' G" } | {S A B C D} |
| E | { G G' G" } | {S A B C D E} |
| G | { G' G" } | {S A B C D E} |

Solution path found is S A G <-- this G also has cost 10

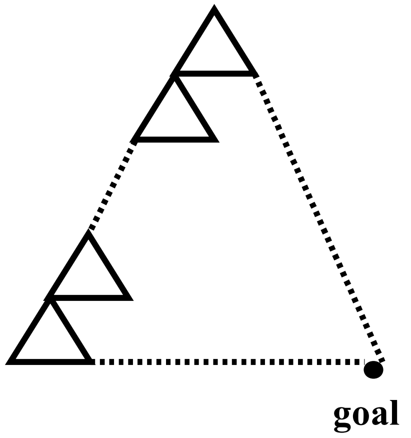
Number of nodes expanded (including goal node) = 7

## CLOSED List: the search tree connected by backpointers



## Depth-First (DFS)

- Algorithm outline:
  - Always select from the OPEN the node with the **greatest depth** for expansion, and put all newly generated nodes into OPEN
  - OPEN is organized as **LIFO** (last-in, first-out) list.
  - Terminate if a node selected for expansion is a goal

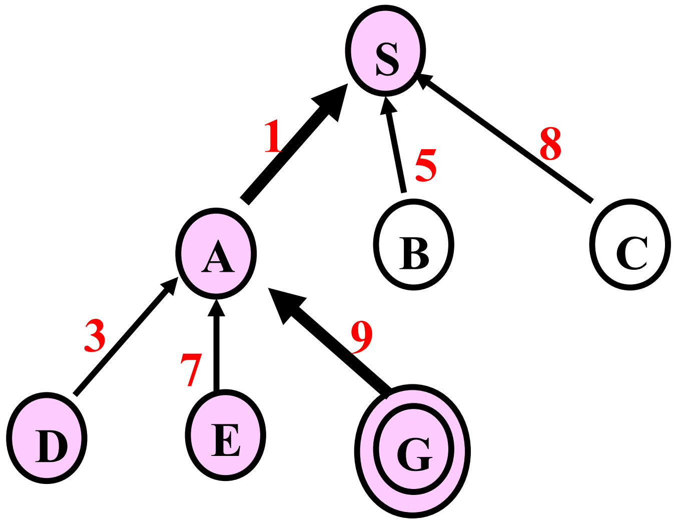


- **May not terminate** without a "depth bound," i.e., cutting off search below a fixed depth D (How to determine the depth bound?)
- **Not complete** (with or without cycle detection, and with or without a cutoff depth)
- **Exponential time**, O(b^d), but only **linear space**, O(bd), required
- Can find deep solutions quickly if lucky
- When search hits a deadend, can only back up one level at a time even if the "problem" occurs because of a bad operator choice near the top of the tree. Hence, only does "chronological backtracking"

# Depth-First Search

return GENERAL-SEARCH(problem, ENQUEUE-AT-FRONT)

| **exp. node** | **OPEN list** | **CLOSED list** |
|---|---|---|
| | { S } | |
| S | { A B C } | |
| A | { D E G B C} | |
| D | { E G B C } | |
| E | { G B C } | |
| G | { B C } | |



Solution path found is S A G <-- this G has cost 10

Number of nodes expanded (including goal node) = 5

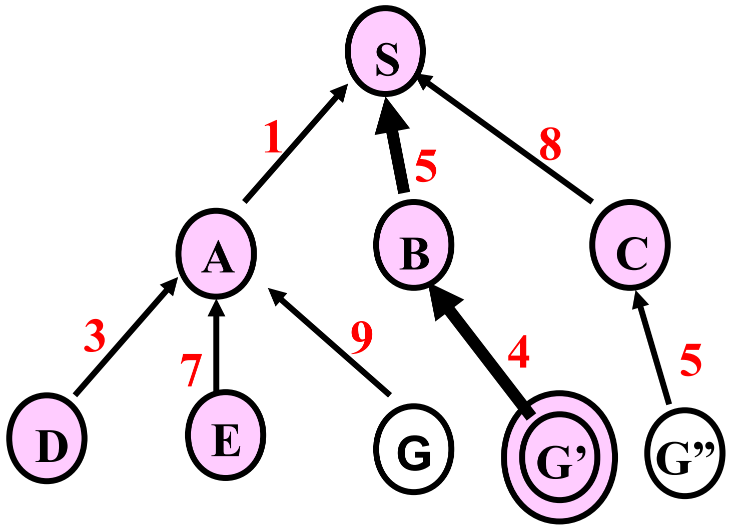
## Uniform-Cost (UCS)

- Let g(n) = cost of the path from the start node to an open node n
- Algorithm outline:
  - Always select from the OPEN the node with the **least g(.) value** for expansion, and put all newly generated nodes into OPEN
  - Nodes in OPEN are sorted by their g(.) values (in ascending order)
  - Terminate if a node selected for expansion is a goal
- Called *“Dijkstra's Algorithm”* in the algorithms literature and similar to *“Branch and Bound Algorithm”* in operations research literature

# Uniform-Cost Search

GENERAL-SEARCH(problem, ENQUEUE-BY-PATH-COST)

| **exp. node** | **nodes list** | **CLOSED list** |
|---|---|---|
| | {S(0)} | |
| S | {A(1) B(5) C(8)} | |
| A | {D(4) B(5) C(8) E(8) G(10)} | |
| D | {B(5) C(8) E(8) G(10)} | |
| B | {C(8) E(8) G'(9) G(10)} | |
| C | {E(8) G'(9) G(10) G"(13)} | |
| E | {G'(9) G(10) G"(13) } | |
| G' | {G(10) G"(13) } | |



Solution path found is S B G <-- this G has cost 9, not 10

Number of nodes expanded (including goal node) = 7

# Uniform-Cost (UCS)

- **Complete** (if cost of each action is not infinitesimal)
  - The total # of nodes n with g(n) <= g(goal) in the state space is finite
  - If n' is a child of n, then g(n') = g(n) + c(n, n') > g(n)
  - Goal node will eventually be generated (put in OPEN) and selected for expansion (and passes the goal test)
- **Optimal/Admissible**
  - Admissibility depends on the goal test being applied when a node is removed from the OPEN list, not when it's parent node is expanded and the node is first generated (delayed goal testing)
  - Multiple solution paths (following different backpointers)
  - Each solution path that can be generated from an open node n will have its path cost >= g(n)
  - When the first goal node is selected for expansion (and passes the goal test), its path cost is less than or equal to g(n) of every OPEN node n (and solutions entailed by n)
- **Exponential time and space complexity**,
  - worst case: becomes BFS when all arcs cost the same

# Depth-First Iterative Deepening (DFID)

- BF and DF both have exponential time complexity O(b^d)

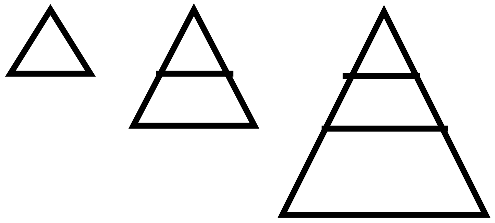
  BF is **complete** but has exponential space complexity (**conservative**)

  DF has **linear space complexity** but is incomplete (**radical**)
- Space is often a **harder** resource constraint than time
- Can we have an algorithm that
  - Is complete
  - Has linear space complexity, and
  - Has time complexity of O(b^d)
- DFID by Korf in 1985 (17 years after A*)

  First do DFS to depth 0 (i.e., treat start node as

  having no successors), then, if no solution found,

  do DFS to depth 1, etc.



```
until solution found do
    DFS with depth bound $c$
    $c = c+1$
```

# Depth-First Iterative Deepening (DFID)

- **Complete** (iteratively generate all nodes up to depth d)
- **Optimal/Admissible** if all operators have the same cost. Otherwise, not optimal but does guarantee finding solution of shortest length (like BF).
- **Linear space complexity:** O(bd), (like DF)
- **Time complexity** is a little worse than BFS or DFS because nodes near the top of the search tree are generated multiple times, but because almost all of the nodes are near the bottom of a tree, the worst case time complexity is still exponential, O(b^d)

# Depth-First Iterative Deepening

- If branching factor is b and solution is at depth d, then nodes at depth d are generated once, nodes at depth d-1 are generated twice, etc., and node at depth 1 is generated d times.

Hence

$$\text{total}(d) = b^d + 2b^{(d-1)} + \ldots + db$$

$$\le b^d / (1 - 1/b)^2 = O(b^d).$$

  - If $b=4$, then worst case is $1.78 * 4^d$, i.e., 78% more nodes searched than exist at depth d (in the worst case).

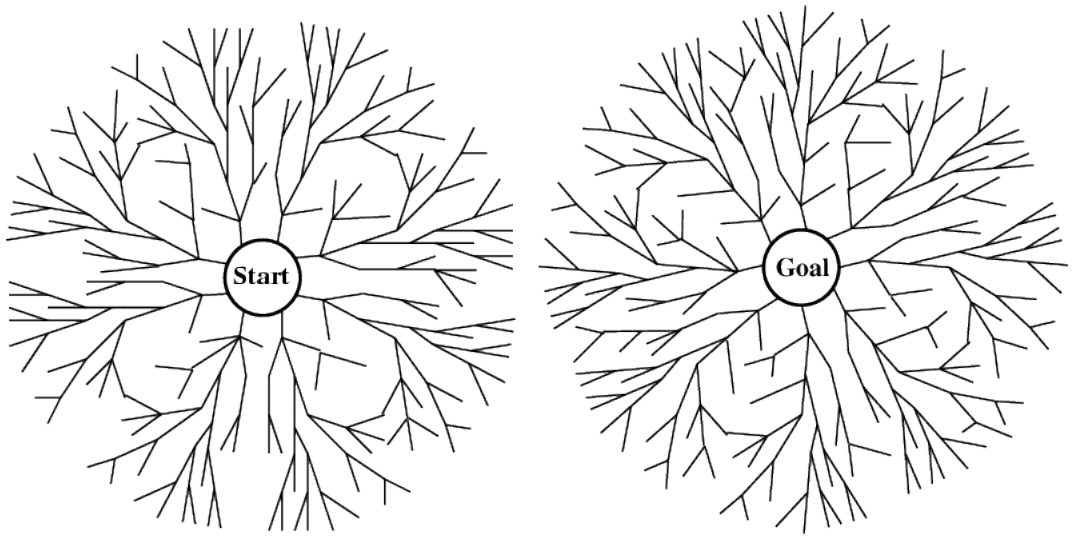
$$tota(d) = 1 \cdot b^d + 2 \cdot b^{d-1} + \cdots + (d-1) \cdot b^2 + d \cdot b$$
$$= b^d (1 + 2 \cdot b^{-1} + \cdots + (d-1) \cdot b^{2-d} + d \cdot b^{1-d})$$

Let $x = b^{-1}$, then

$$tota(d) = b^d (1 + 2 \cdot x^1 + \cdots + (d-1) \cdot x^{d-2} + d \cdot x^{d-1})$$
$$= b^d \frac{d}{dx} (x + x^2 + \cdots + x^{d-1} + x^d)$$
$$= b^d \frac{d}{dx} \frac{(x - x^{d+1})}{1-x}$$
$$\le b^d \frac{d}{dx} \frac{x}{1-x} \qquad /* x^{d+1} << 1 \text{ when } d \text{ is large since } 1/b < 1 */$$
$$= b^d \frac{1 \cdot (1-x) - x \cdot (-1)}{(1-x)^2}. \text{ Therefore}$$

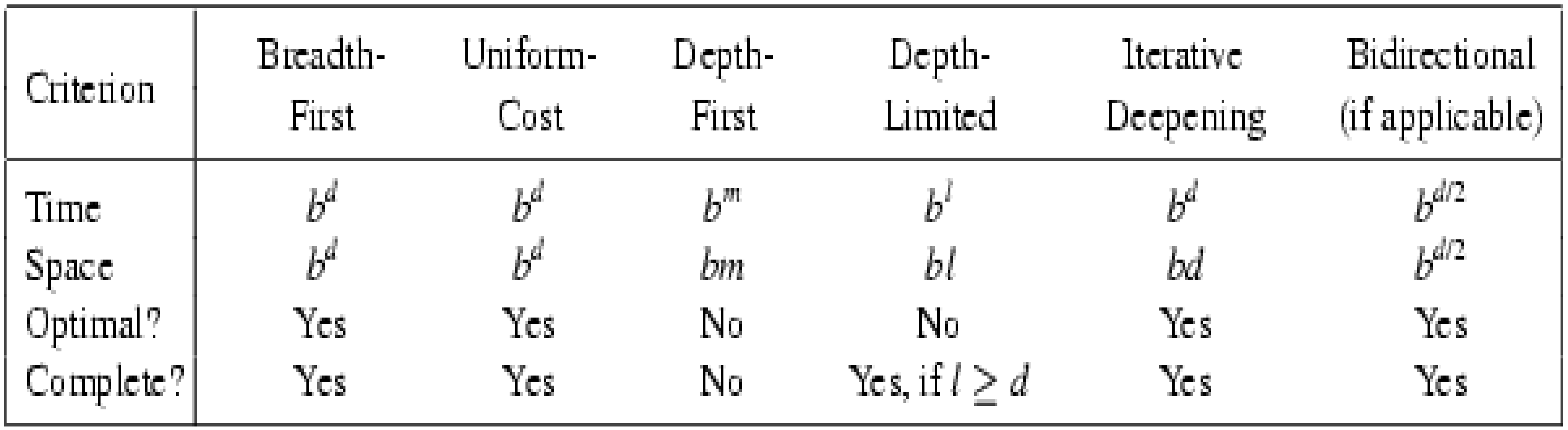
$$tota(d) \le b^d / (1-x)^2 = b^d / (1-b^{-1})^2$$

## Bi-directional search



- Alternate searching from the start state toward the goal and from the goal state toward the start.
- Stop when the frontiers intersect.
- Works well only when there are unique start and goal states and when actions are reversible
- Can lead to finding a solution more quickly (but watch out for pathological situations).

## Comparing Search Strategies



| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening | Bidirectional (if applicable) |
|---|---|---|---|---|---|---|
| Time | $b^d$ | $b^d$ | $b^m$ | $b^l$ | $b^d$ | $b^{d/2}$ |
| Space | $b^d$ | $b^d$ | $bm$ | $bl$ | $bd$ | $b^{d/2}$ |
| Optimal? | Yes | Yes | No | No | Yes | Yes |
| Complete? | Yes | Yes | No | Yes, if $l \geq d$ | Yes | Yes |

## When to use what

- **Depth-First Search:**
  - Many solutions exist
  - Know (or have a good estimate of) the depth of solution
- **Breadth-First Search**:
  - Some solutions are known to be shallow
- **Uniform-Cost Search**:
  - Actions have varying costs
  - Least cost solution is required

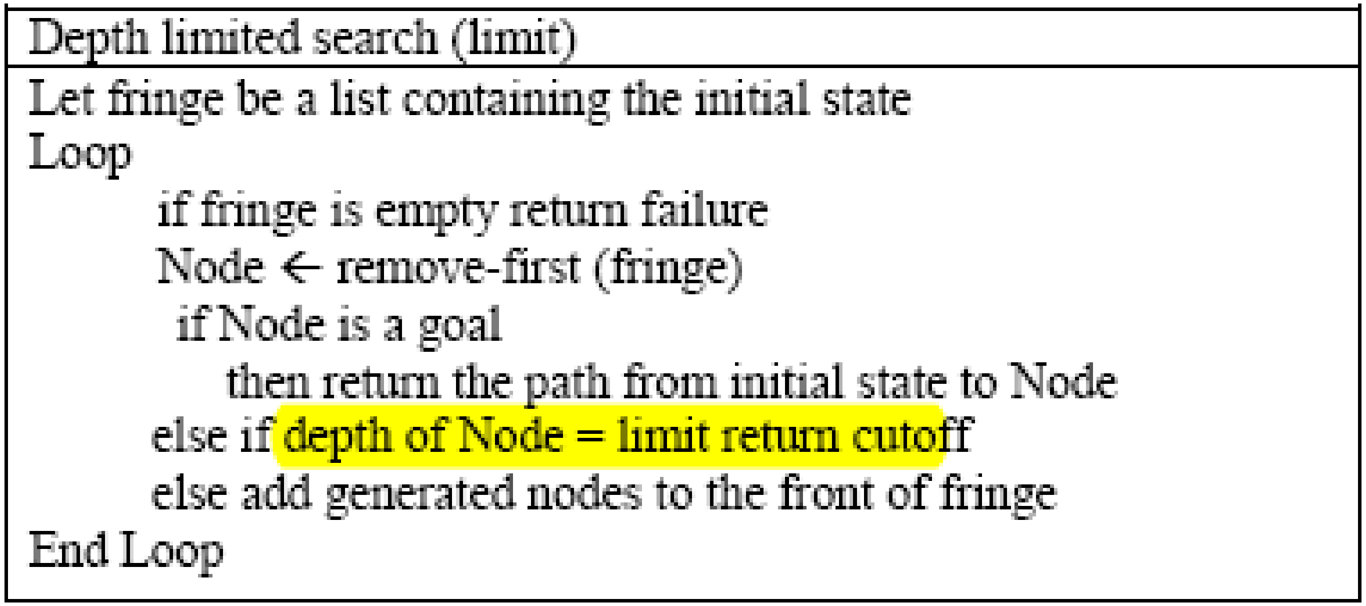
*This is the only uninformed search that worries about costs.*

- **Iterative-Deepening Search**:
  - Space is limited and the shortest solution path is required

# Uninformed Search Depth Limited and Iterative Deepening Search

## Depth Limited Search

- A variation of Depth First Search circumvents the above problem by keeping a depth bound.
- Nodes are only expanded if they have depth less than the bound. This algorithm is known as depth-limited search.

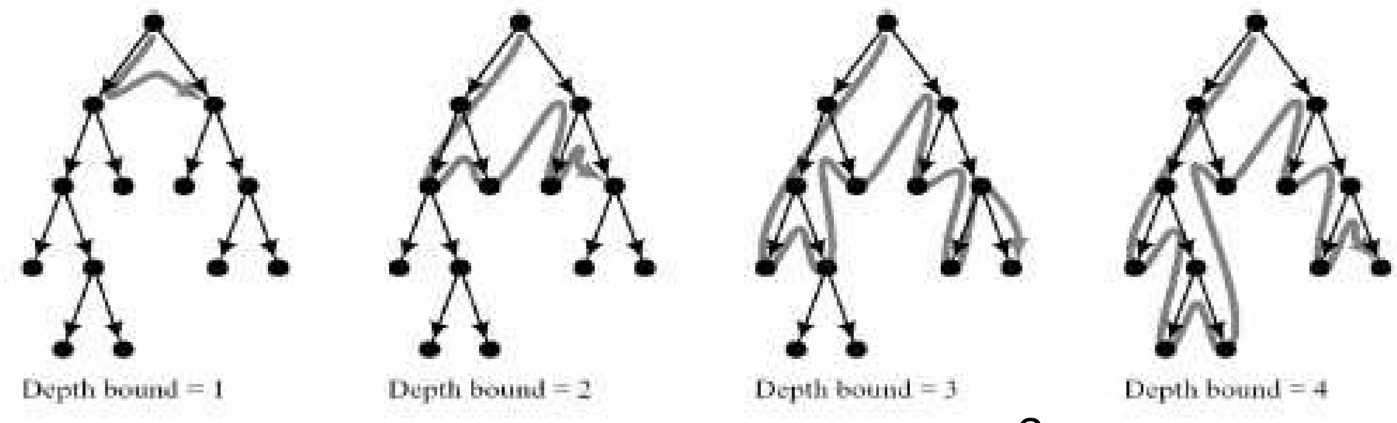


```
Depth limited search (limit)
Let fringe be a list containing the initial state
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
         if Node is a goal
            then return the path from initial state to Node
        else if depth of Node = limit return cutoff
        else add generated nodes to the front of fringe
End Loop
```

## Depth First Iterative deepening Search (DFID)

- First do DFS to depth 0 (i.e., treat start node as having no successors), then, if no solution found, do DFS to depth 1, etc.

```
DFID
until solution found do
DFS with depth cutoff c
c = c+1
```

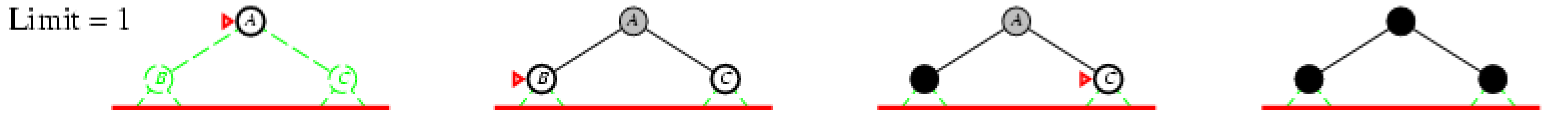


## Iterative deepening search

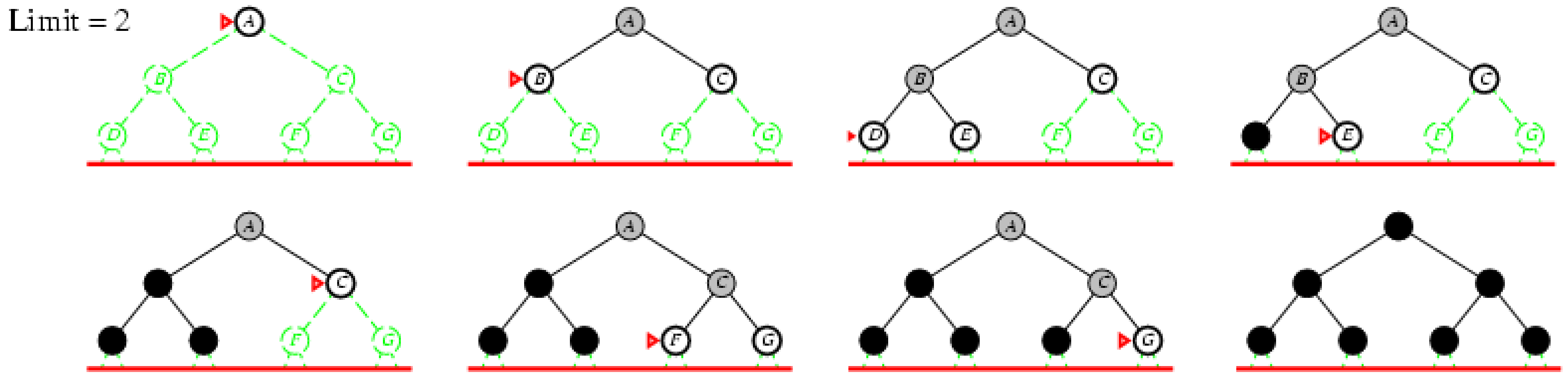
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

## Iterative deepening search *l* =0

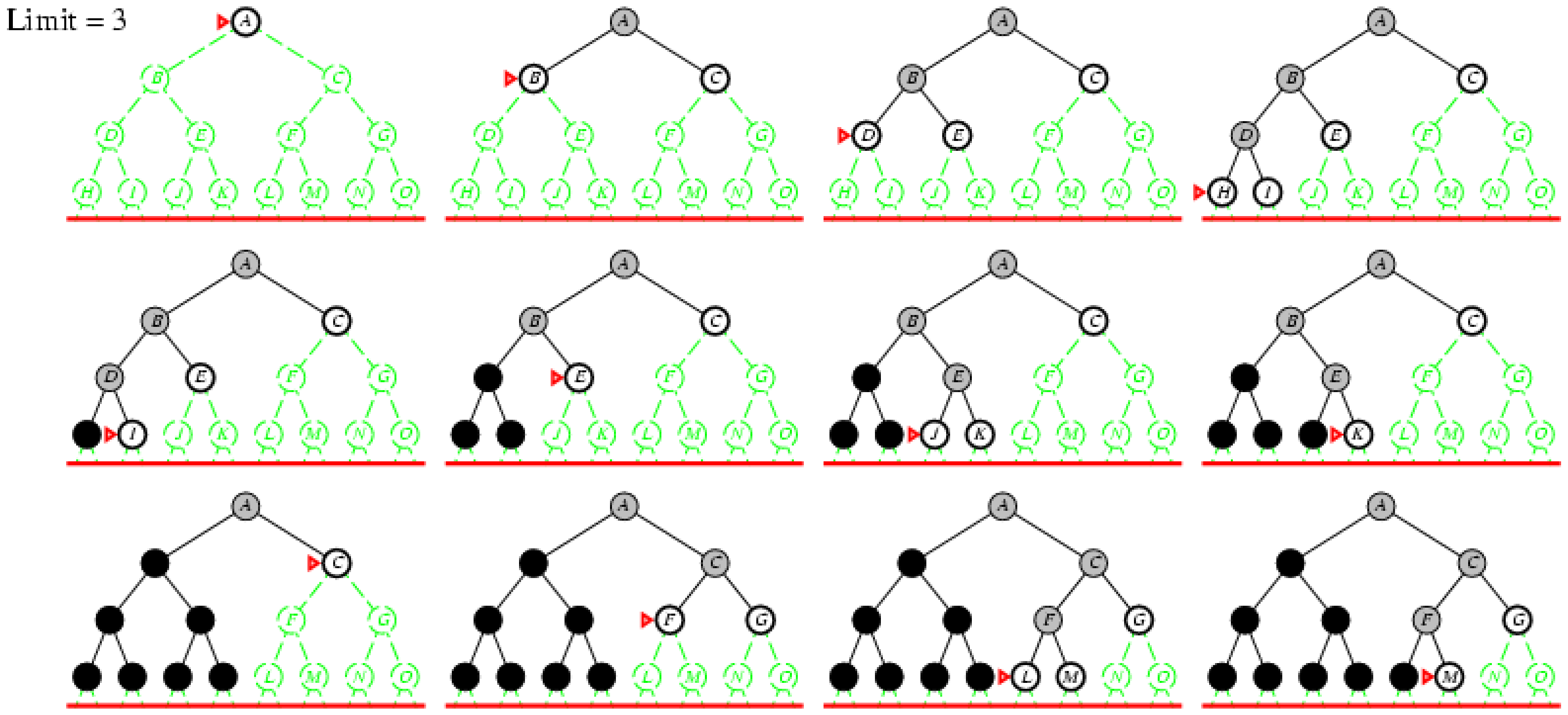
# Iterative deepening search *l* =1



# Iterative deepening search *l* =2



# Iterative deepening search *l* =3



# Iterative deepening search

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

  *NDLS = b0 + b1 + b2 + … + bd-2 + bd-1 + bd*

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

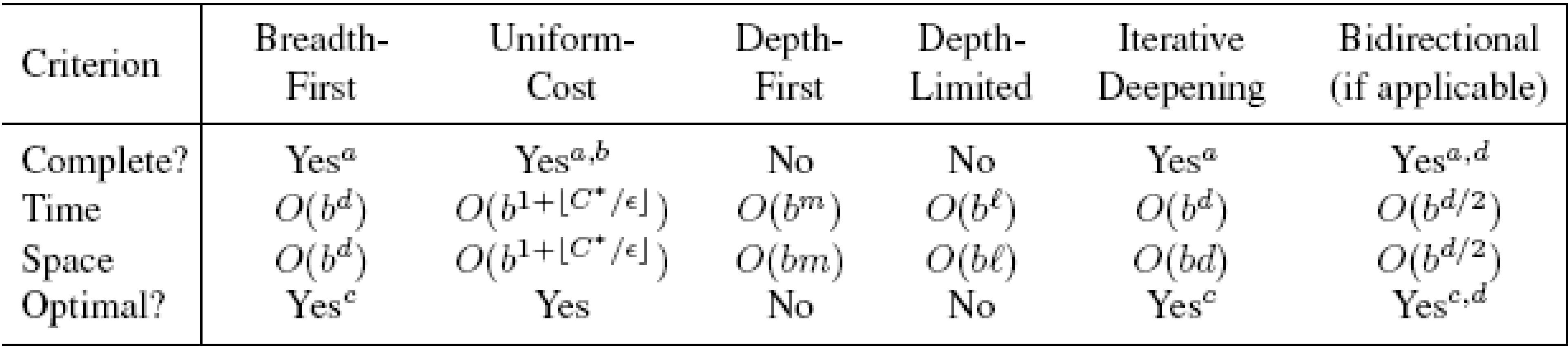
  NIDS = (d+1)b0 + d b1 + (d-1)b2 + … + 3bd-2 +2bd-1 + 1bd

- For *b = 10*, *d = 5*,
  - NDLS = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
  - NIDS = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456

## Properties of iterative deepening search

- Complete? Yes
- Time? *(d+1)b0 + d b1 + (d-1)b2 + … + bd = O(bd)*
- Space? *O(bd)*
- Optimal? Yes, if step cost = 1

# Comparing Uninformed Search Strategy



| 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] |

# Problem Solving by Searching

## Search Methods :

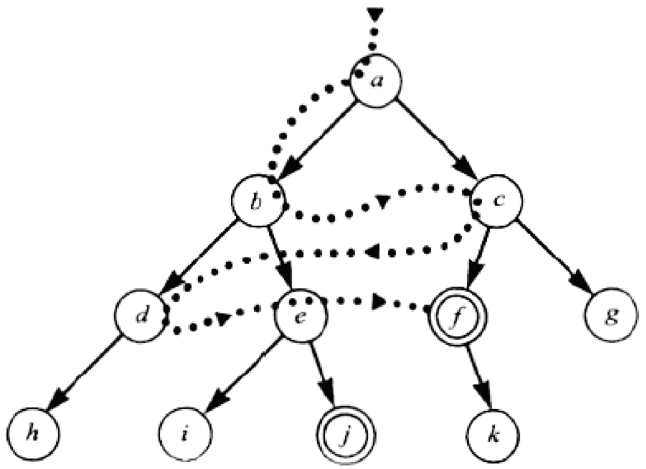
informed (Heuristic) search

## Using problem specific knowledge to aid searching

- Without incorporating knowledge into searching, one can have no *bias* (i.e. a preference) on the search space.

- Without a bias, one is forced to look everywhere to find th answer. Hence, the complexity of uninformed search is intractable.

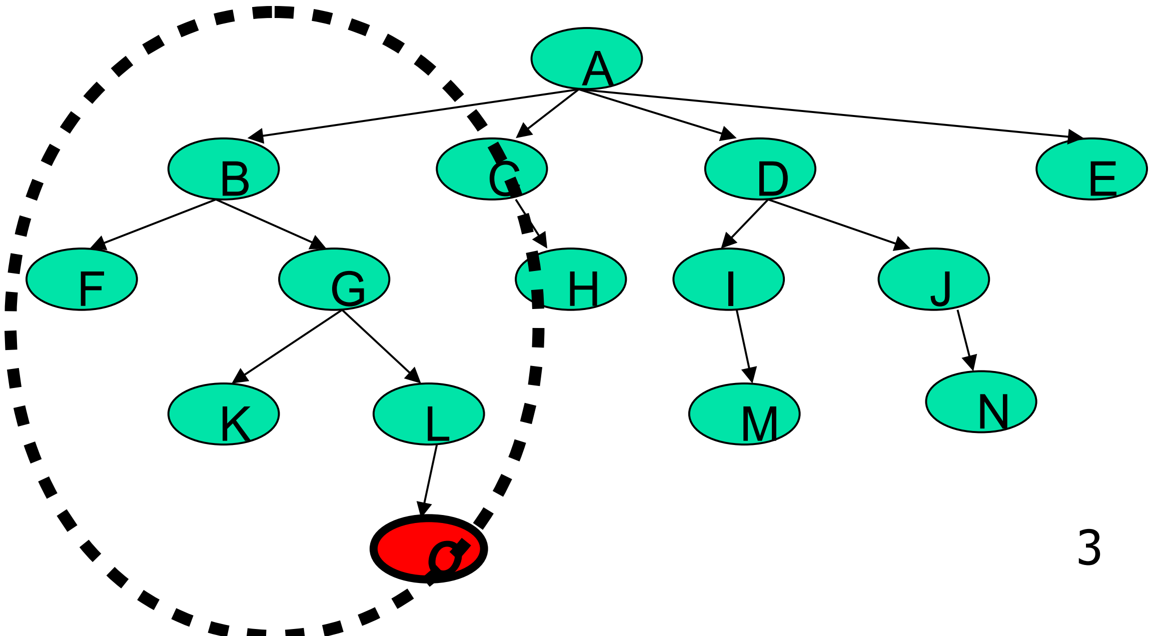
Search everywhere!!



# Using problem specific knowledge to aid searching

- With knowledge, one can search the state space as if he was given "hints" when exploring a maze.
  - Heuristic information in search = Hints
- Leads to dramatic speed up in efficiency.

Search only in this subtree!!



# More formally, why heuristic functions work?

- In any search problem where there are at most *b* choices at each node and a depth of *d* at the goal node, a naive search algorithm would have to, in the worst case, search around *O(bd)* nodes before finding a solution (Exponential Time Complexity).

- Heuristics improve the efficiency of search algorithms by reducing the <u>effective branching</u> factor from *b* to (ideally) a low constant b* such that
  - 1 =< b* << b

# Heuristic Functions

- A heuristic function is a function *f(n)* that gives an estimation on the "cost" of getting from node *n* to the goal state – so that the node with the least cost among all possible choices can be selected for expansion first.

- Three approaches to defining *f*:
  - *f* measures the value of the current state (its "goodness")
  - *f* measures the estimated cost of getting to the goal from the current state:
    - *f(n)* = *h(n)* where *h(n)* = an estimate of the cost to get from *n* to a goal
  - *f* measures the estimated cost of getting to the goal state from the *current state* and the cost of the existing path to it. Often, in this case, we decompose *f*:
    - *f(n)* = *g(n)* + *h(n)* where *g(n)* = the cost to get to *n* (from initial state)

## Approach 1: *f* Measures the Value of the Current State

- Usually the case when solving optimization problems
  - Finding a state such that the value of the metric *f* is optimized
- Often, in these cases, *f* could be a weighted sum of a set of component values:
  - N-Queens
    - Example: the number of queens under attack …
  - Data mining
    - Example: the "predictive-ness" (a.k.a. accuracy) of a rule discovered

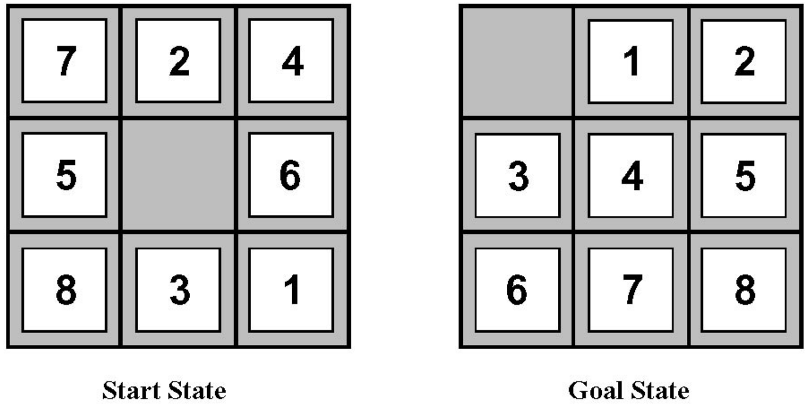
# Approach 2: *f* Measures the Cost to the Goal

A state *X* would be better than a state *Y* if the estimated cost of getting from *X* to the goal is lower than that of *Y* – because *X* would be closer to the goal than *Y*

- 8–Puzzle

**h1**: The number of misplaced tiles (squares with number).

**h2**: The sum of the distances of the tiles from their goal positions.



## Approach 3: *f* measures the total cost of the solution path (Admissible Heuristic Functions)

- A heuristic function $f(n) = g(n) + h(n)$ is admissible if $h(n)$ **never** overestimates the cost to reach the goal.
  - Admissible heuristics are "optimistic": "the cost is not that much …"
- However, $g(n)$ is the exact cost to reach node $n$ from the initial state.
- Therefore, $f(n)$ never over-estimate the true cost to reach the goal state through node $n$.
- Theorem: A search is optimal if $h(n)$ is admissible.
  - I.e. The search using $h(n)$ returns an optimal solution.
- Given $h2(n) > h1(n)$ for all $n$, it's always more efficient to use $h2(n)$.
  - *h2* is more realistic than *h1 (more informed)*, though both are optimistic.

## Traditional informed search strategies

- Greedy Best first search
  - "Always chooses the successor node with the best *f* value" where $f(n) = h(n)$
  - We choose the one that is nearest to the final state among all possible choices

- A* search
  - Best first search using an "admissible" heuristic function *f* that takes into account the current cost *g*
  - Always returns the optimal solution path

# Informed Search Strategies

Best First Search

## An implementation of Best First Search

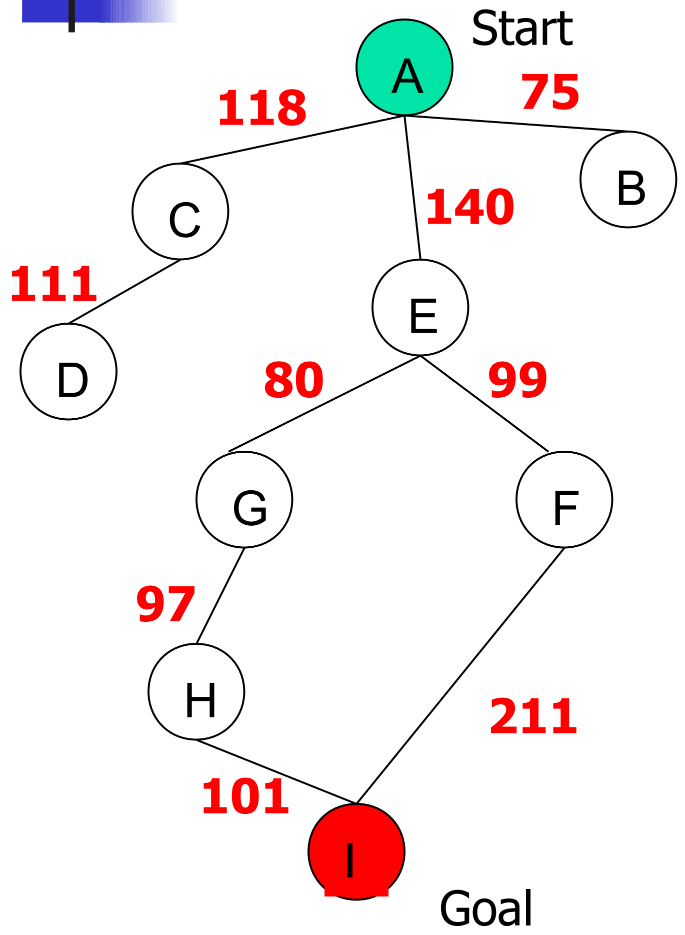
```
function BEST-FIRST-SEARCH (problem, eval-fn)
        returns a solution sequence, or failure

    queuing-fn = a function that sorts nodes by eval-fn

return GENERIC-SEARCH (problem,queuing-fn)
```

# Informed Search Strategies

## Greedy Search

*eval-fn*: f(n) = h(n)

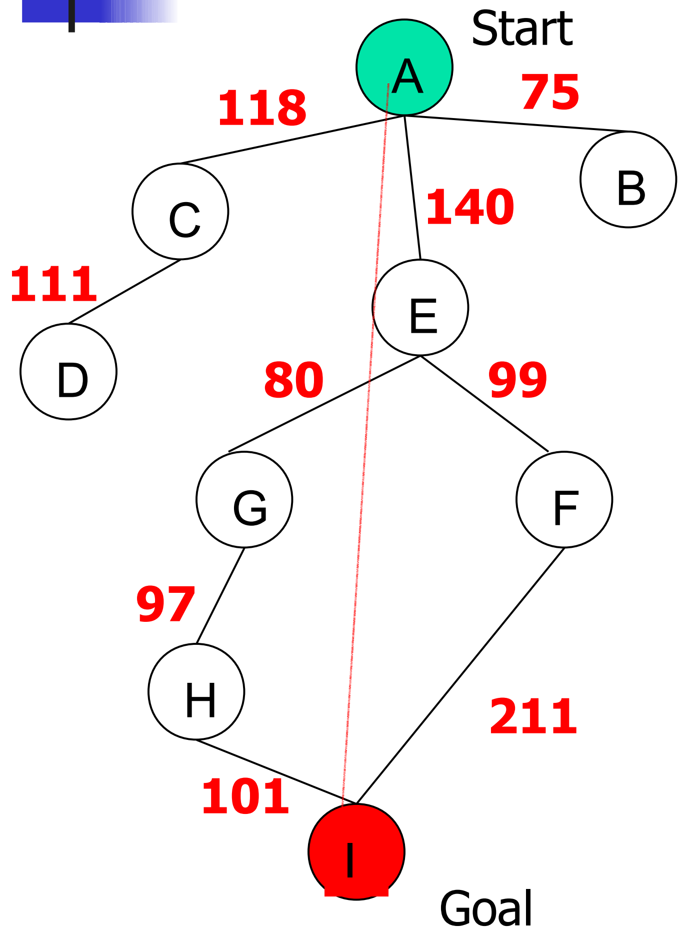
# Greedy Search



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

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

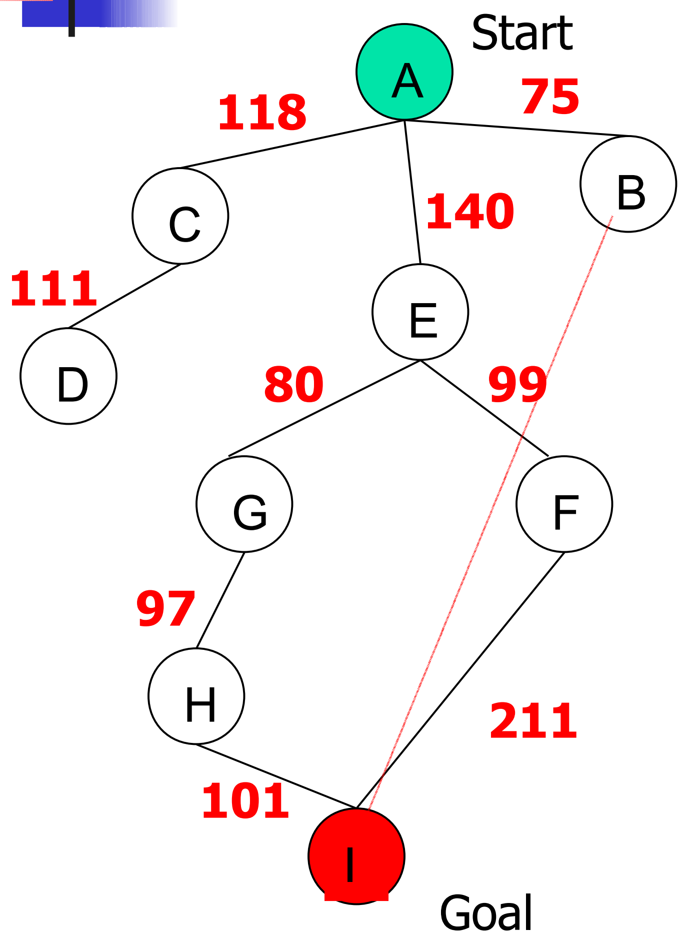
# Greedy Search



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

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

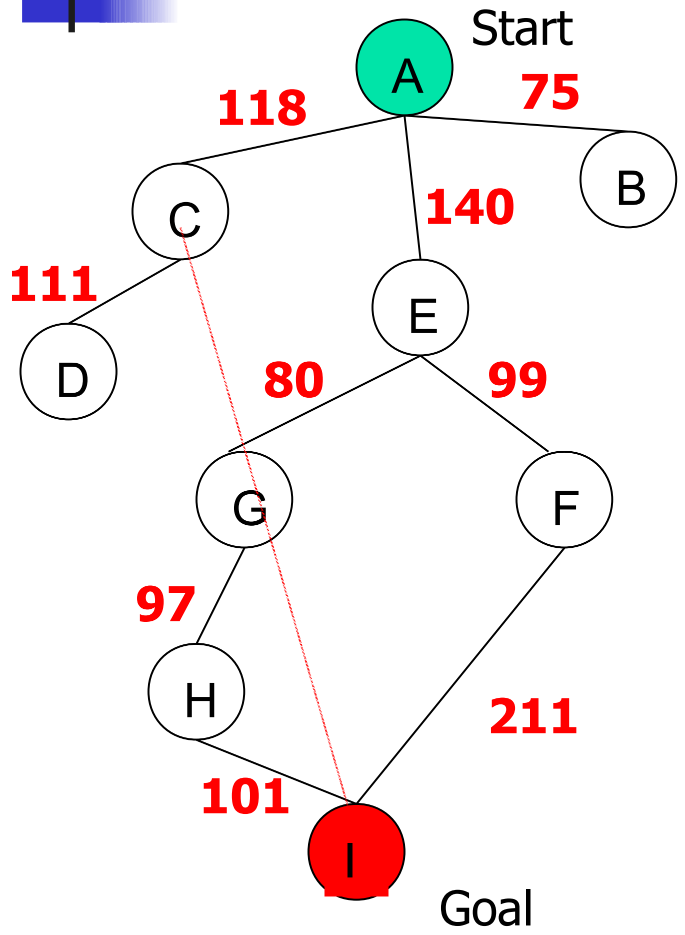
# Greedy Search



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

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

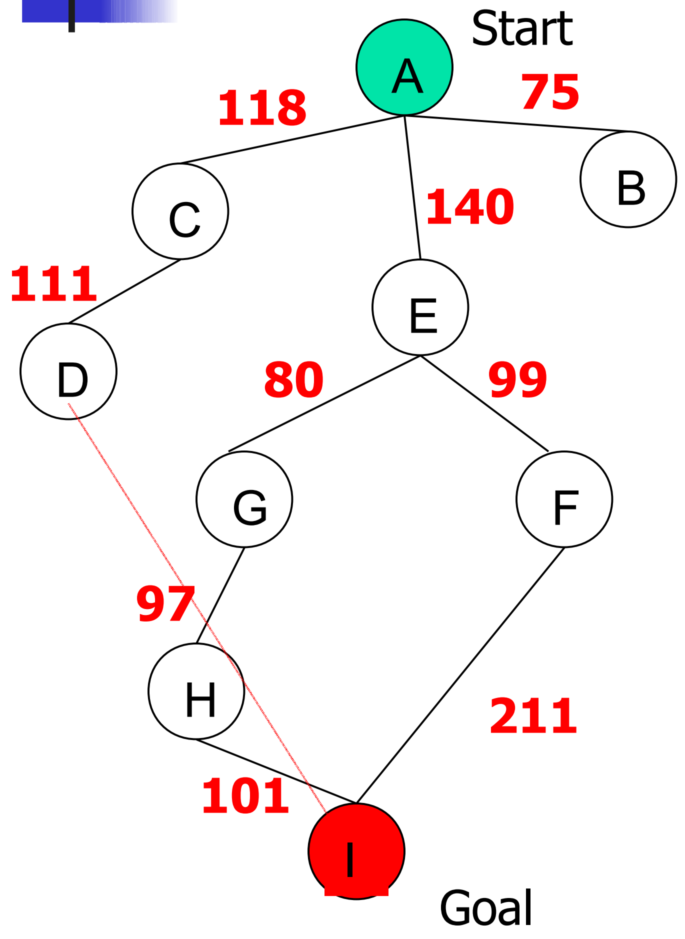
# Greedy Search



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

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

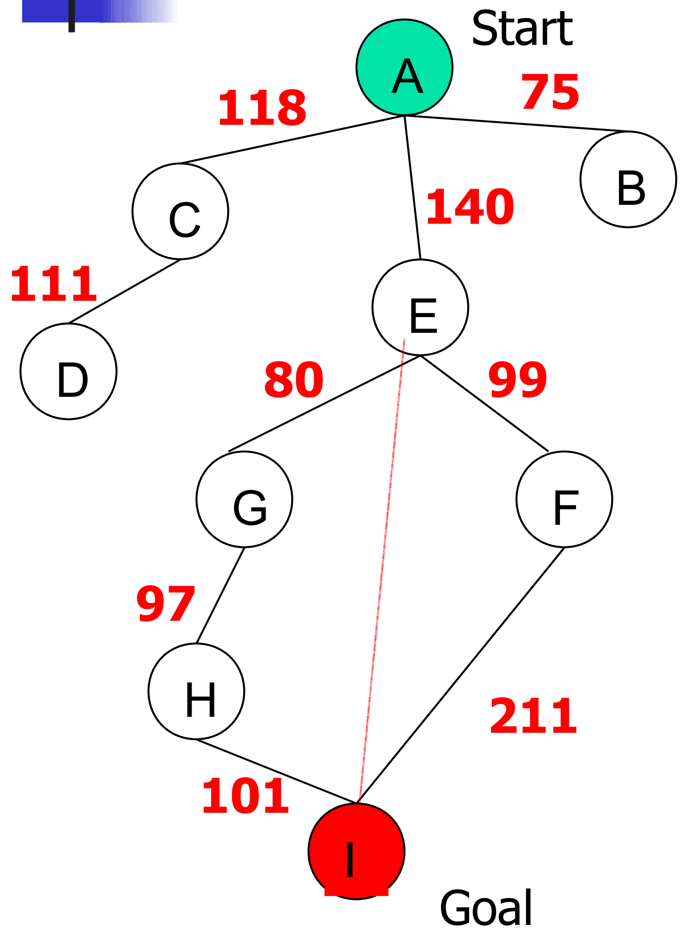
# Greedy Search



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

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

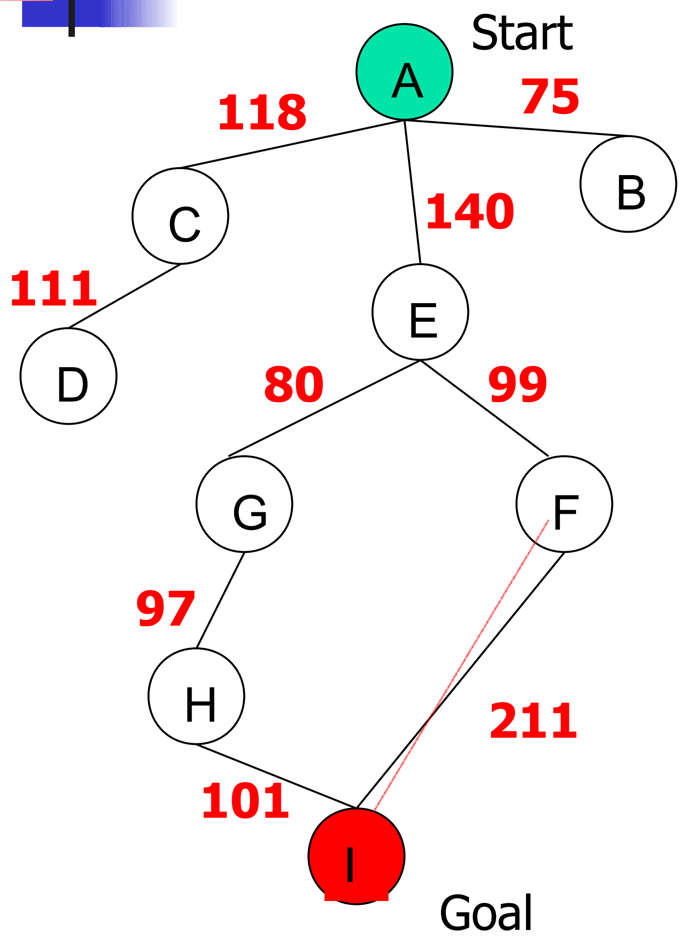
# Greedy Search



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

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

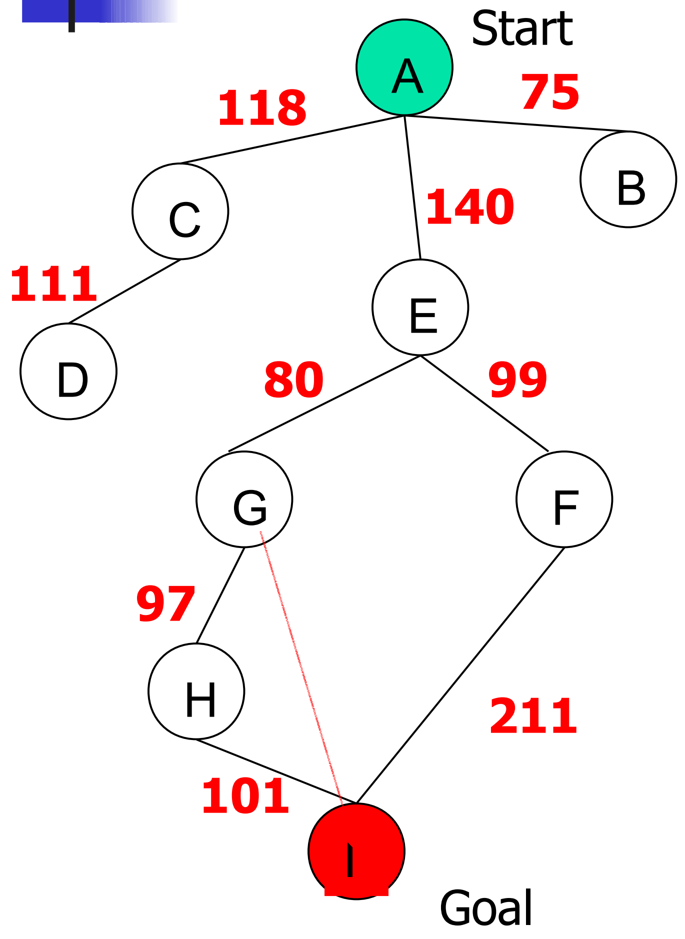
## Greedy Search



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

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

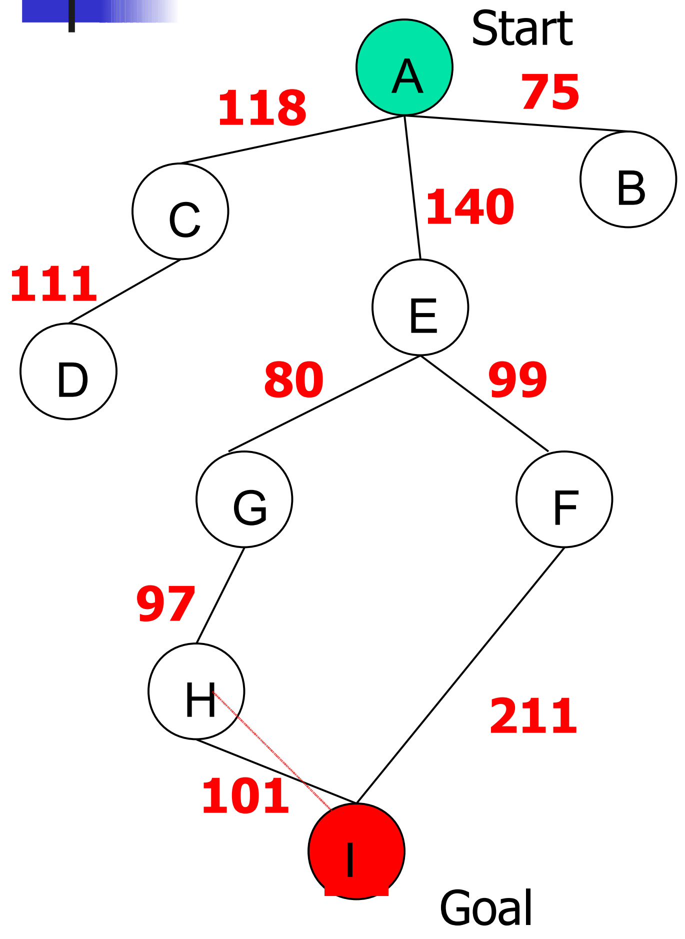
## Greedy Search



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

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

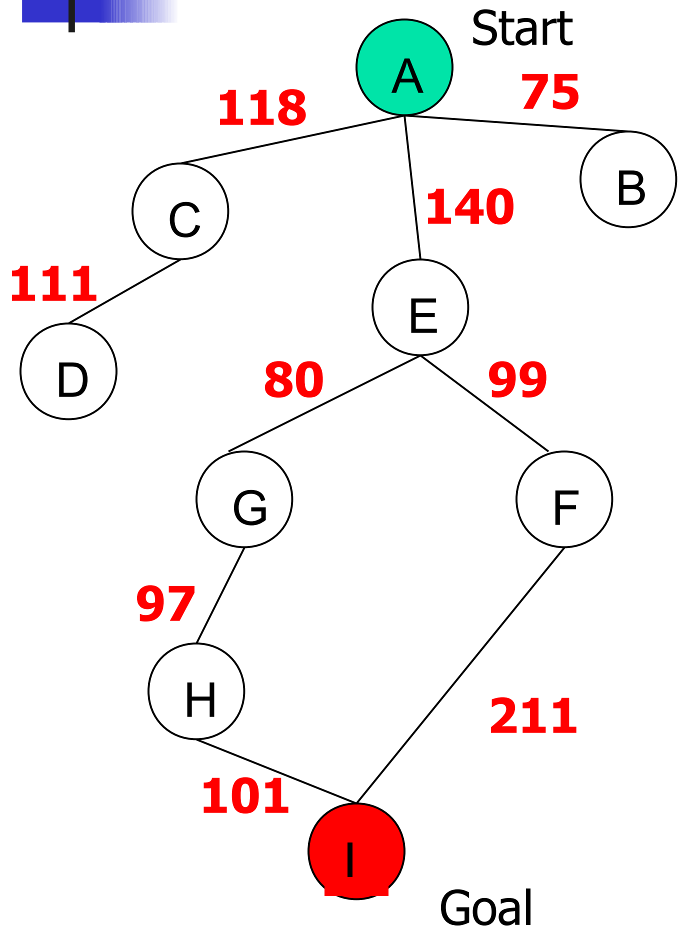
# Greedy Search



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

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

# Greedy Search

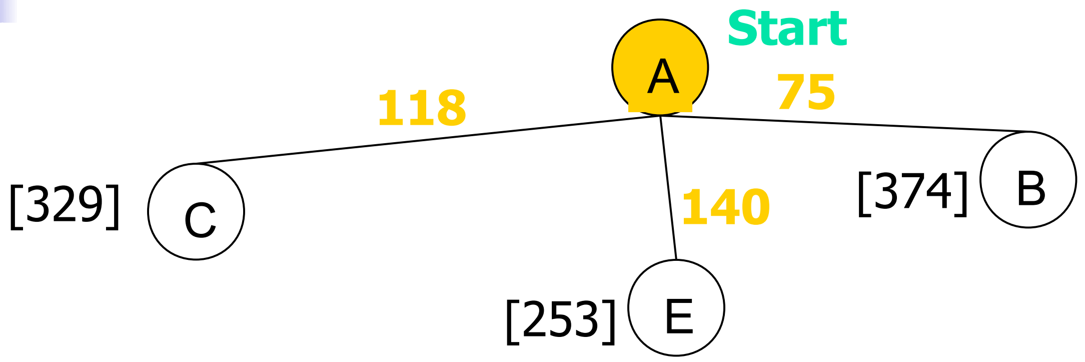


| State | Heuristic: h(n) |
|---|---|
| A | 366 |
| B | 374 |
| C | 329 |
| D | 244 |
| E | 253 |
| F | 178 |
| G | 193 |
| H | 98 |
| **I** | **0** |

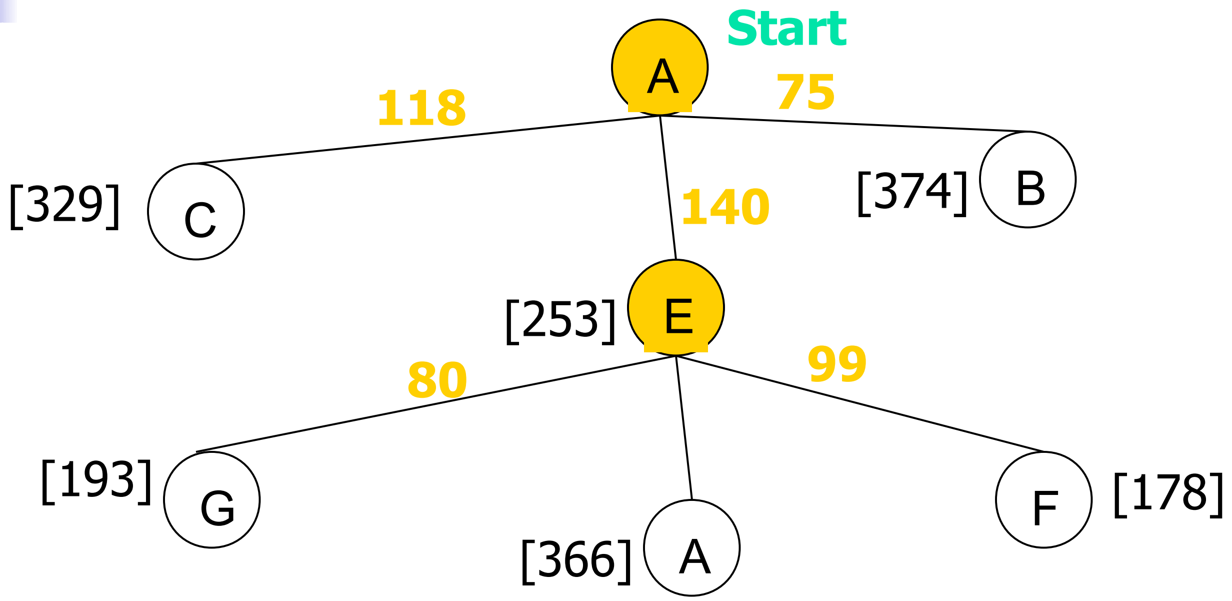
***f(n) = h (n)* = straight-line distance heuristic**

## Greedy Search: Tree Search

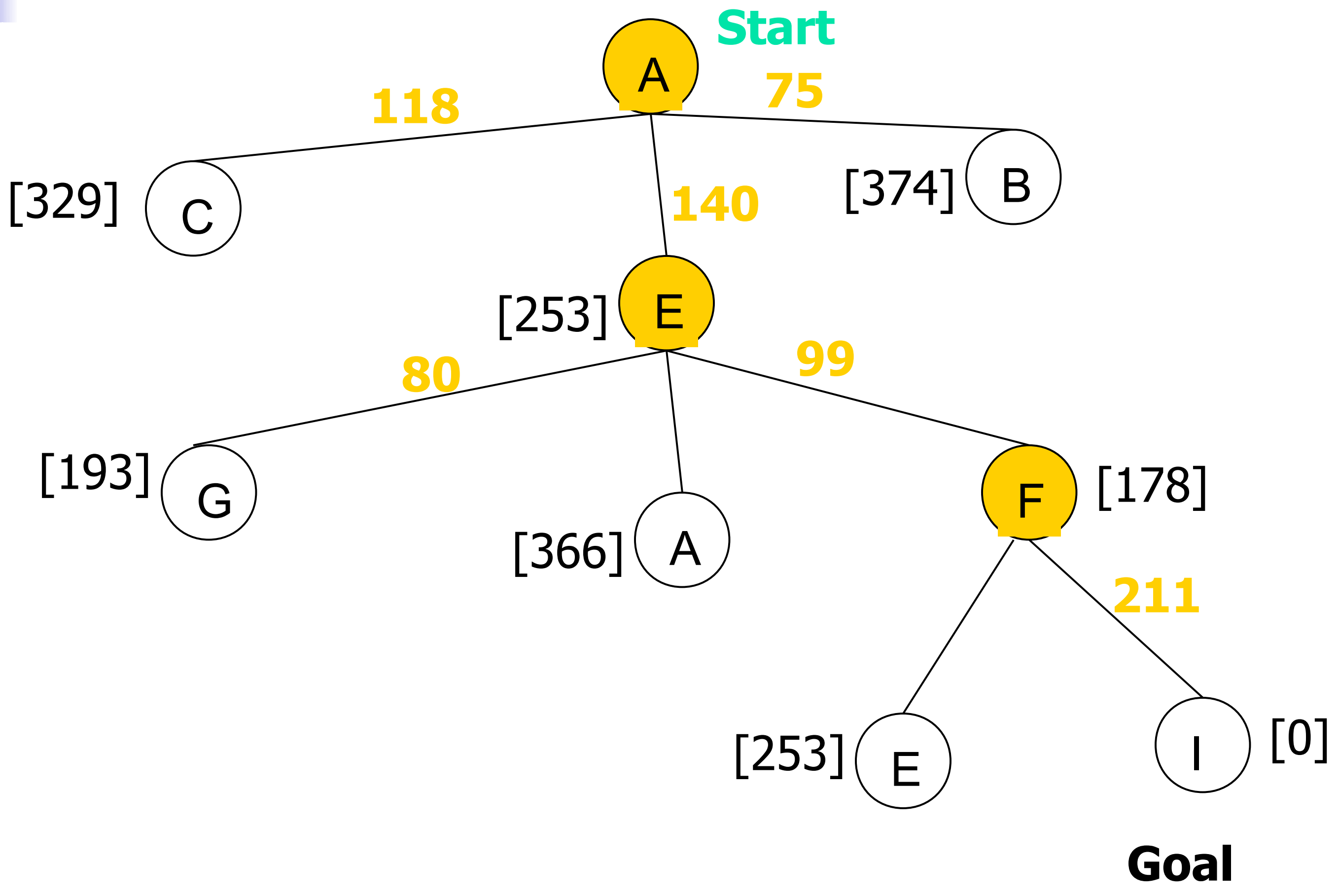
## Greedy Search: Tree Search



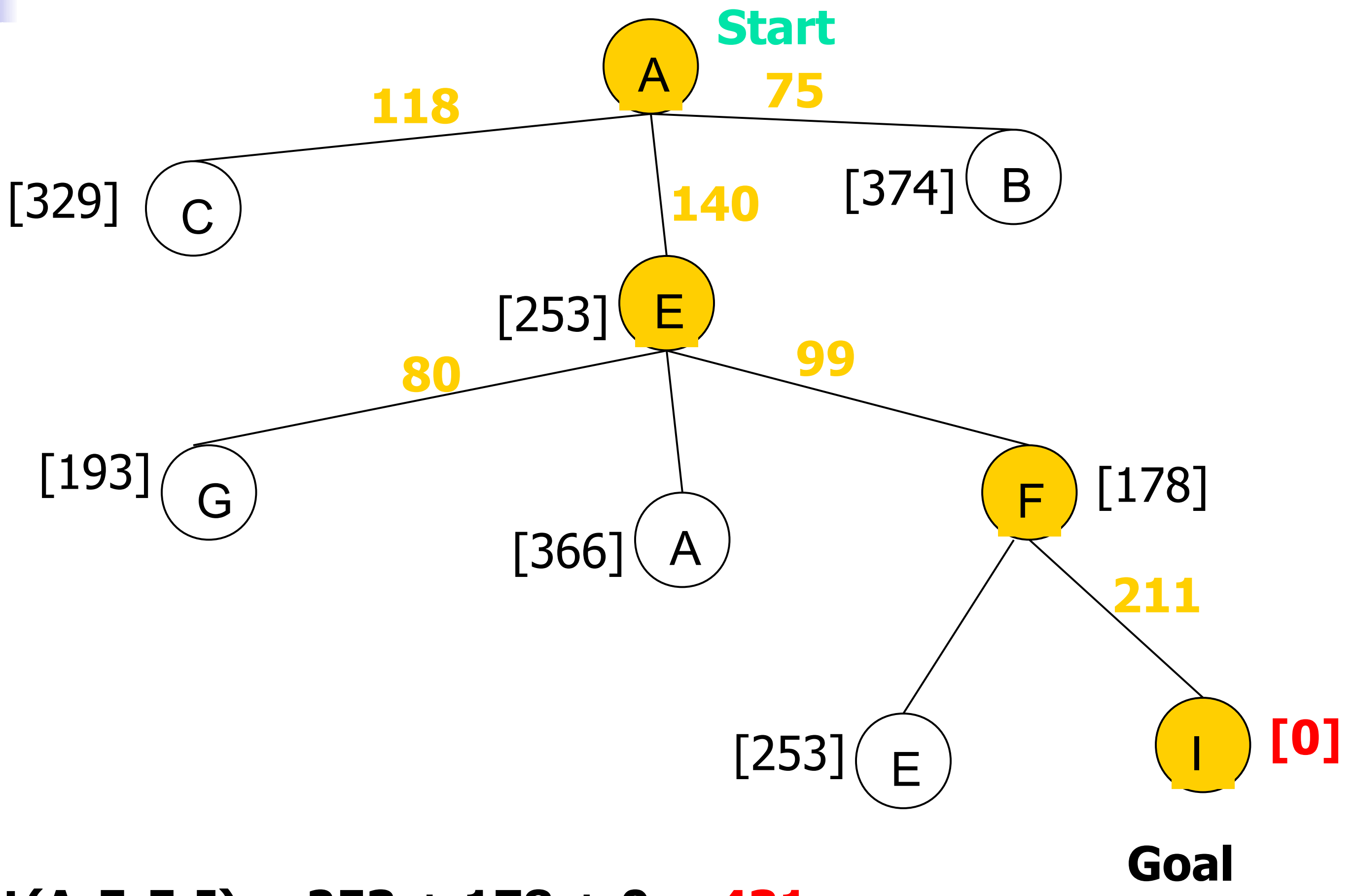
# Greedy Search: Tree Search



# Greedy Search: Tree Search



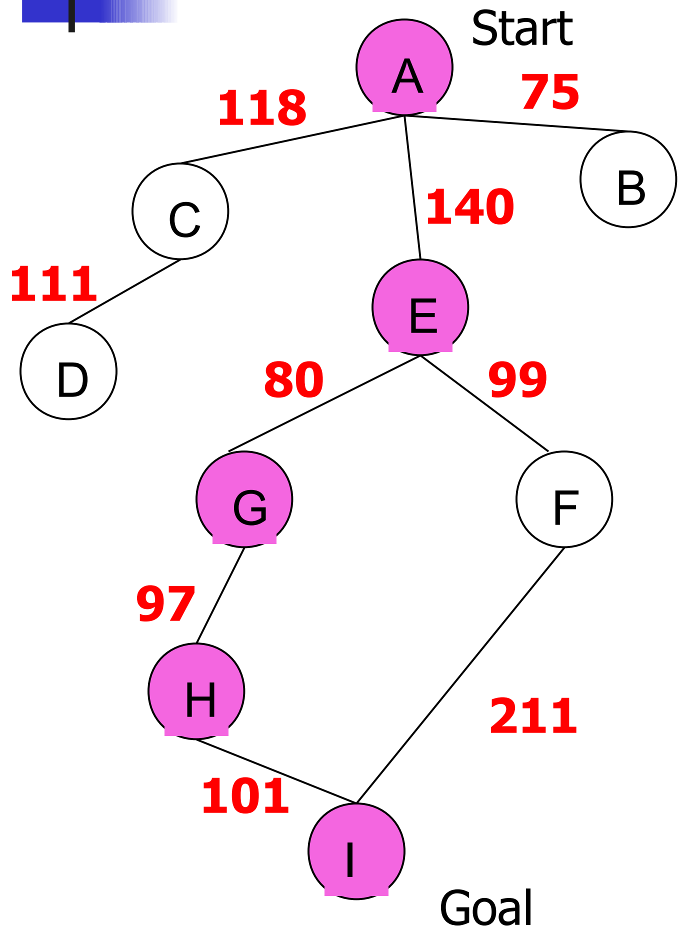
# Greedy Search: Tree Search



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

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

# Greedy Search: Optimal ?

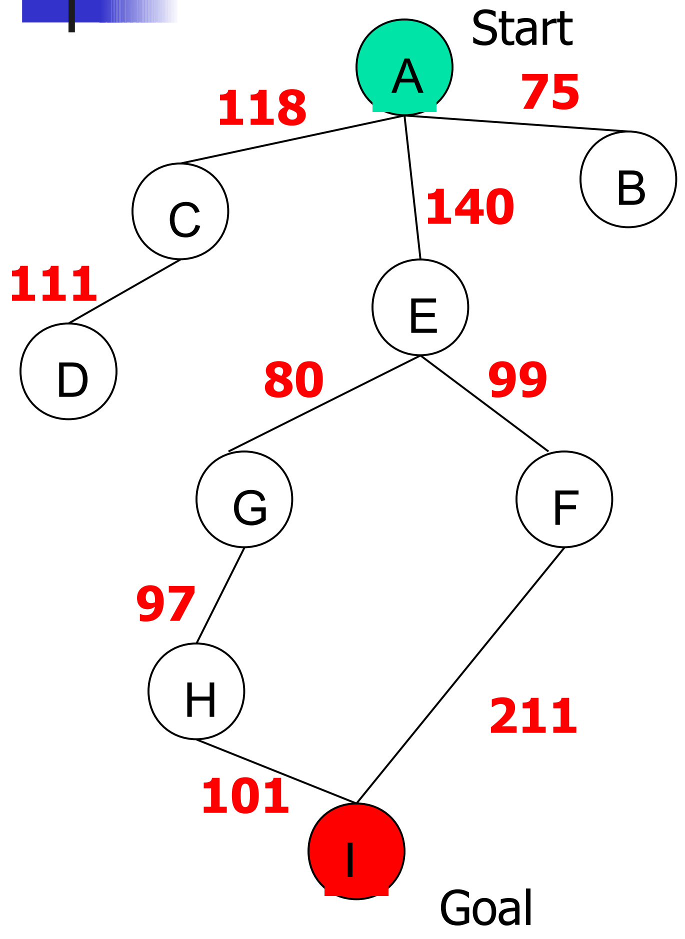


| State | Heuristic: h(n) |
|---|---|
| A | 366 |
| B | 374 |
| C | 329 |
| D | 244 |
| **E** | **253** |
| F | 178 |
| **G** | **193** |
| **H** | **98** |
| I | 0 |

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

**dist(A-E-G-H-I) =140+80+97+101=418**

# Greedy Search: Complete ?

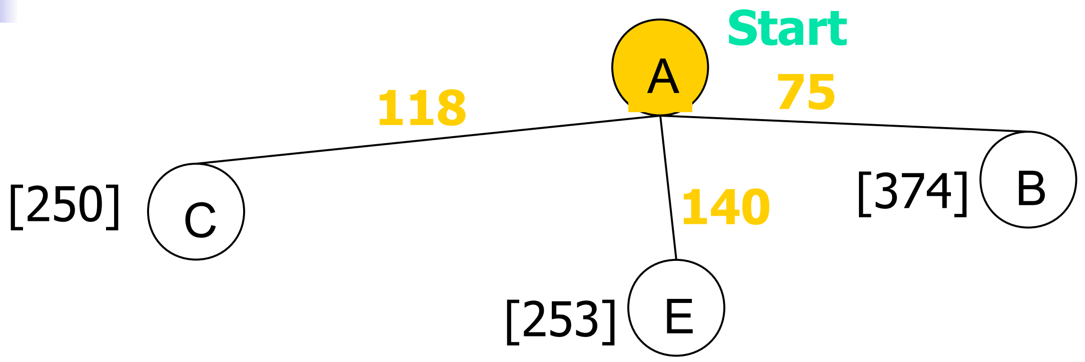


| State | Heuristic: h(n) |
|---|---|
| A | 366 |
| B | 374 |
| ** C | **250** |
| D | 244 |
| E | 253 |
| F | 178 |
| G | 193 |
| H | 98 |
| I | 0 |

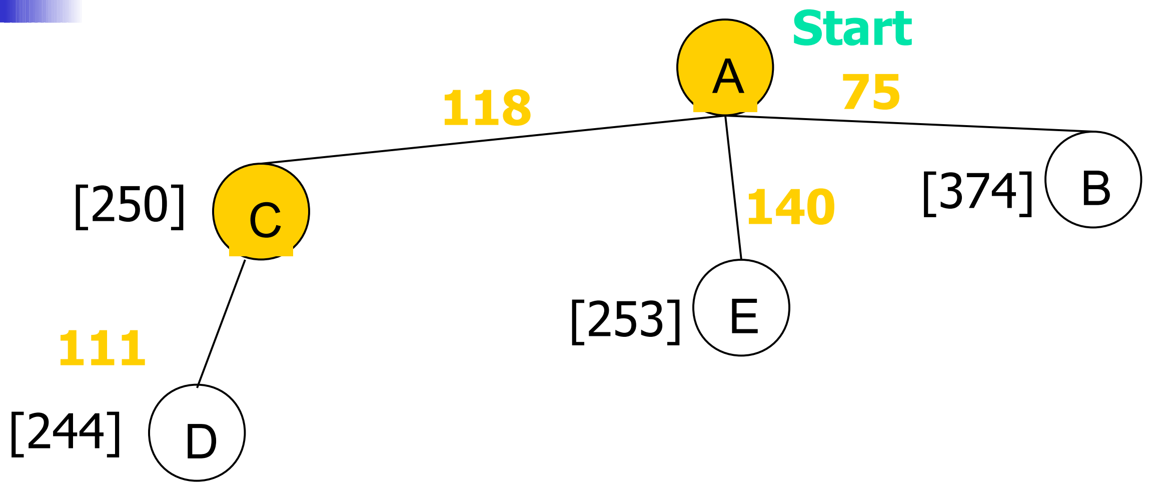
***f(n) = h*** **(*n*) = straight-line distance heuristic**

# Greedy Search: Tree Search

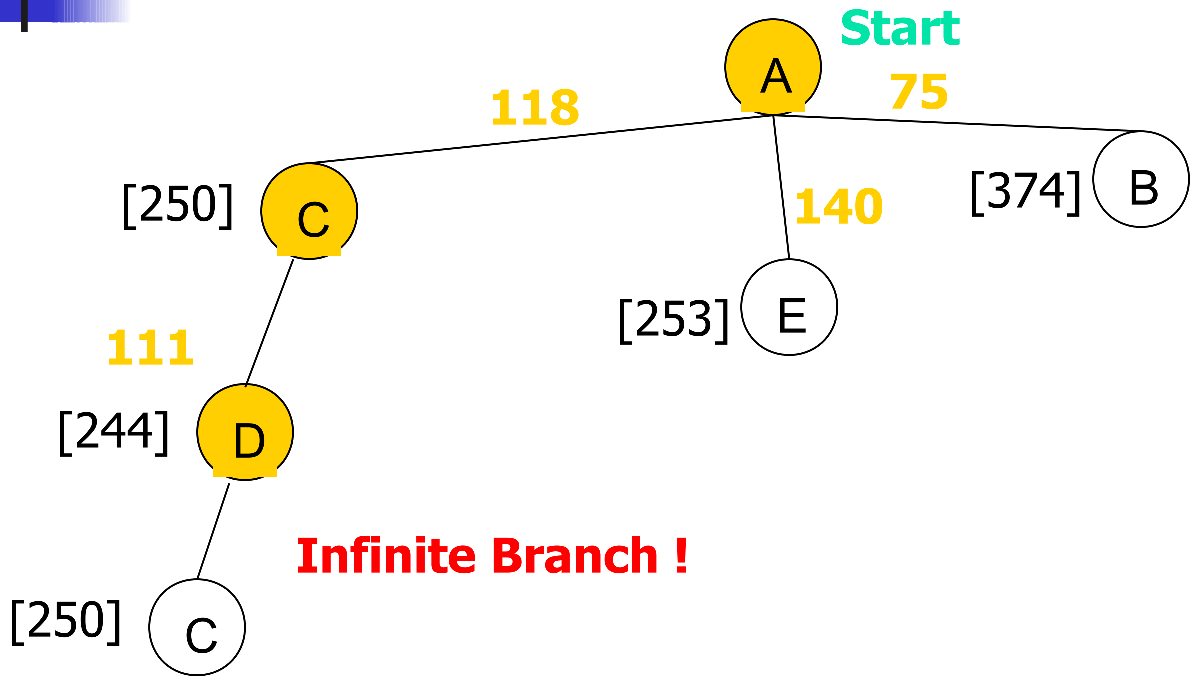
# Greedy Search: Tree Search



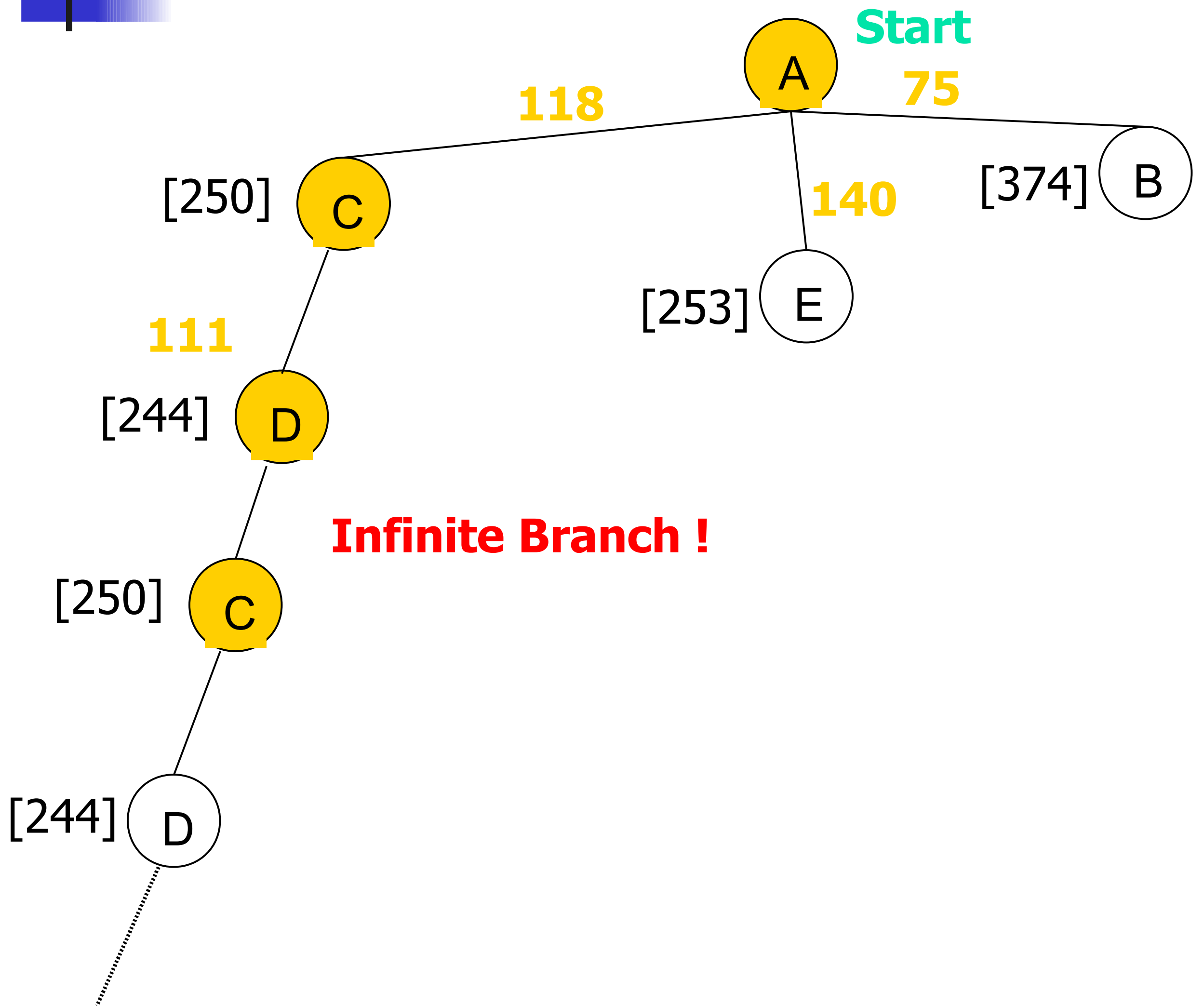
# Greedy Search: Tree Search



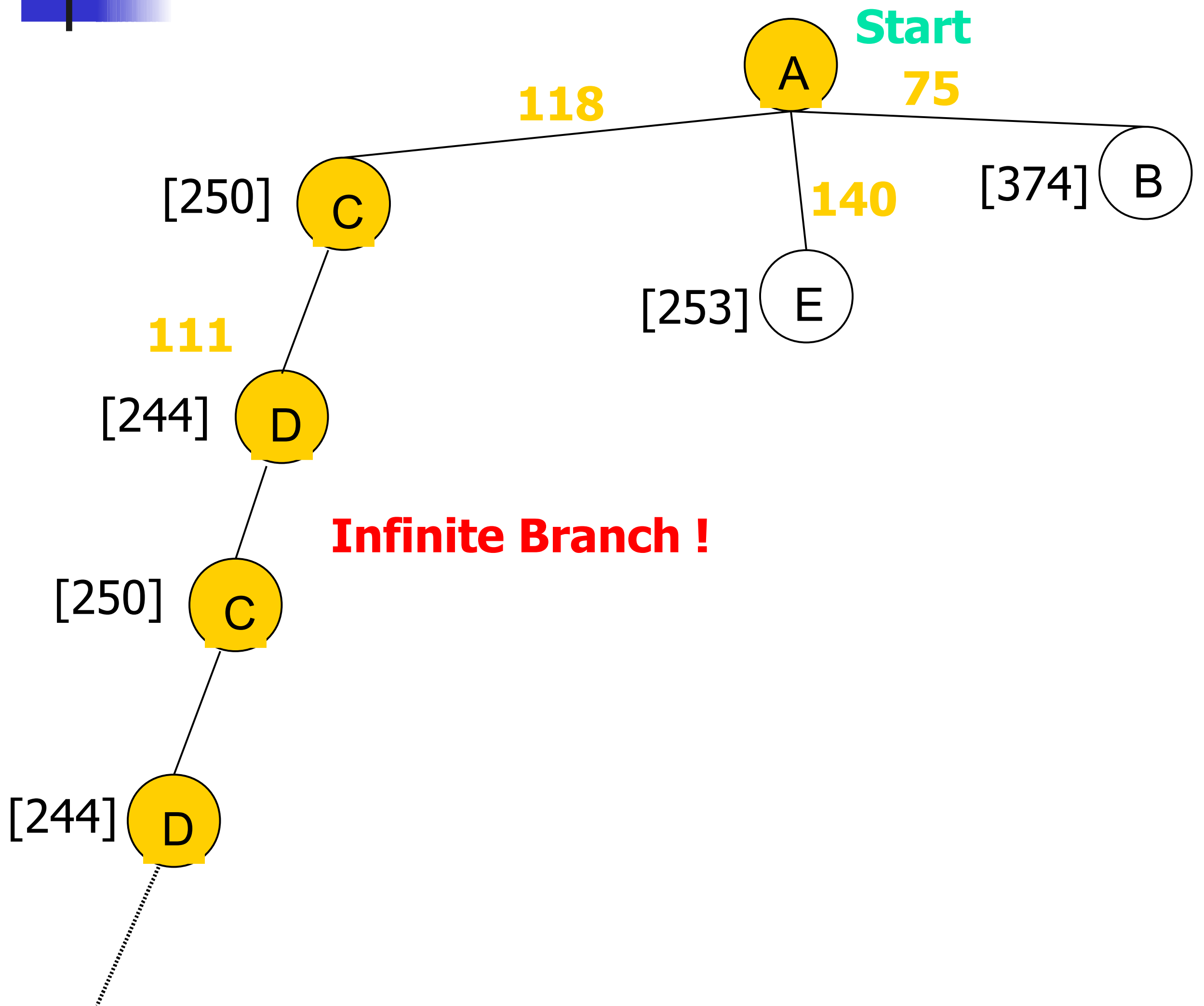
# Greedy Search: Tree Search



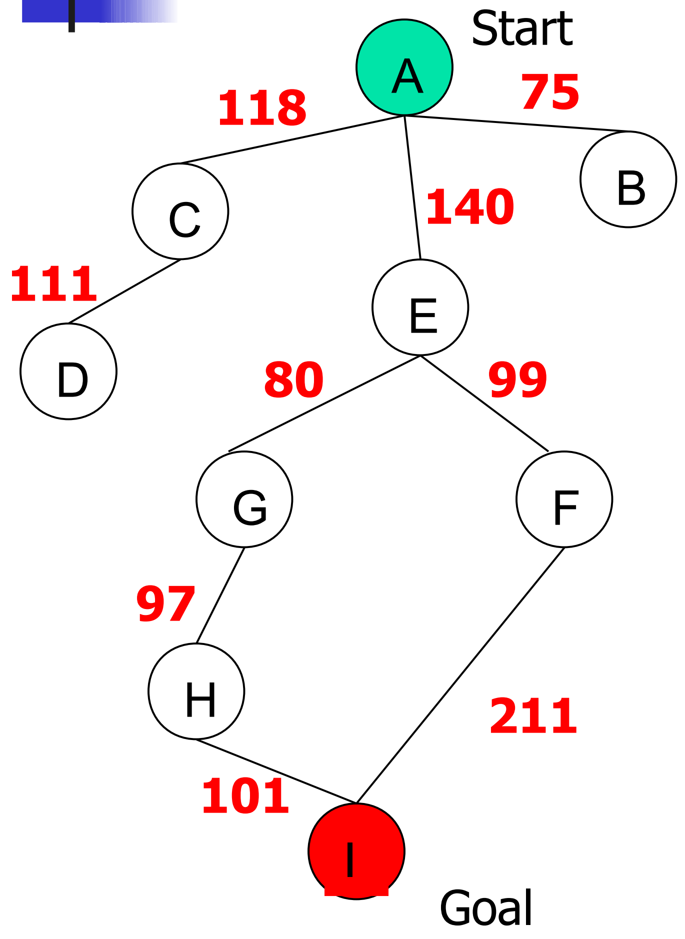
# Greedy Search: Tree Search



# Greedy Search: Tree Search



## Greedy Search: Time and Space Complexity ?



- Greedy search is not optimal.
- Greedy search is incomplete without systematic checking of repeated states.
- In the worst case, the Time and Space Complexity of Greedy Search are both O(bm)

Where b is the branching factor and m the maximum path length

# Informed Search Strategies

## A* Search

*eval-fn*: f(n)=g(n)+h(n)

# A* (A Star)

- Greedy Search minimizes a heuristic h(n) which is an estimated cost from a node n to the goal state. Greedy Search is efficient but it is not optimal nor complete.

- Uniform Cost Search minimizes the cost g(n) from the initial state to n. UCS is optimal and complete but not efficient.

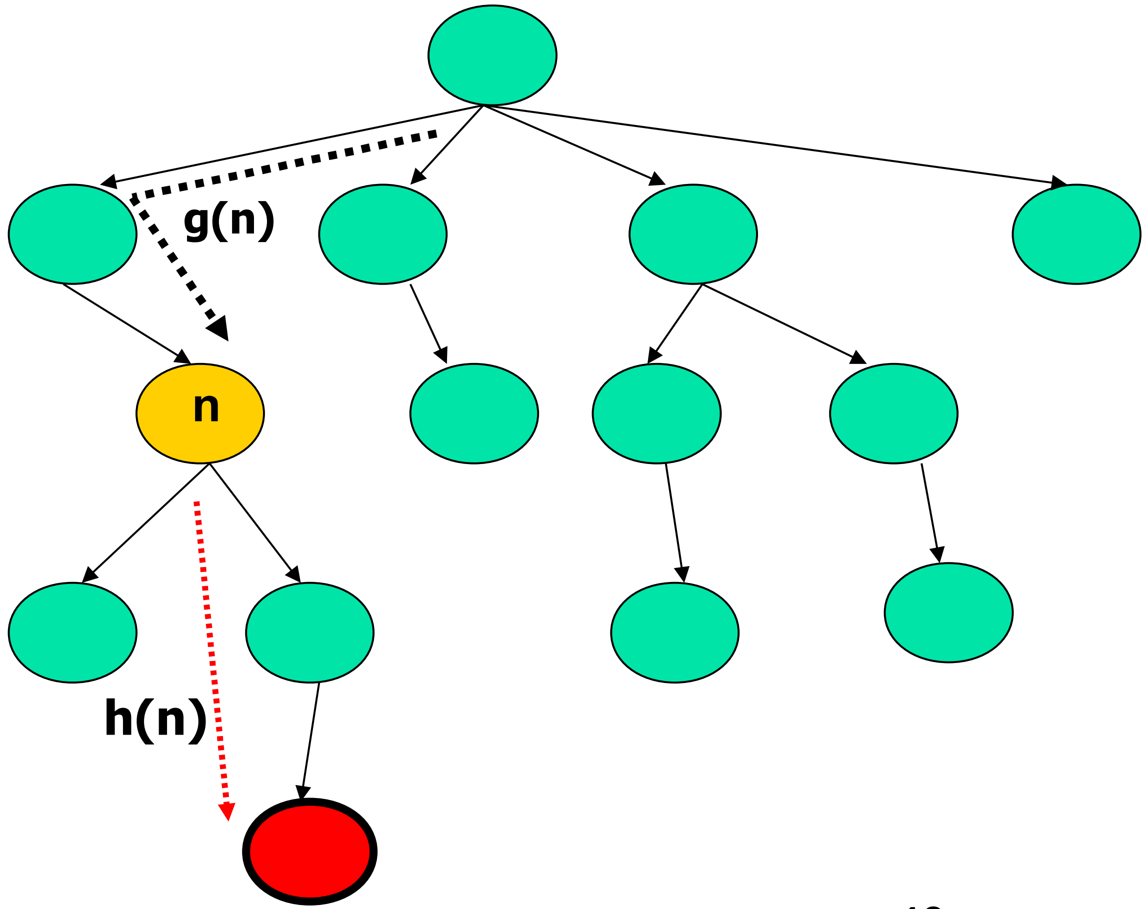
- **New Strategy**: Combine Greedy Search and UCS to get an efficient algorithm which is complete and optimal.

## A* (A Star)

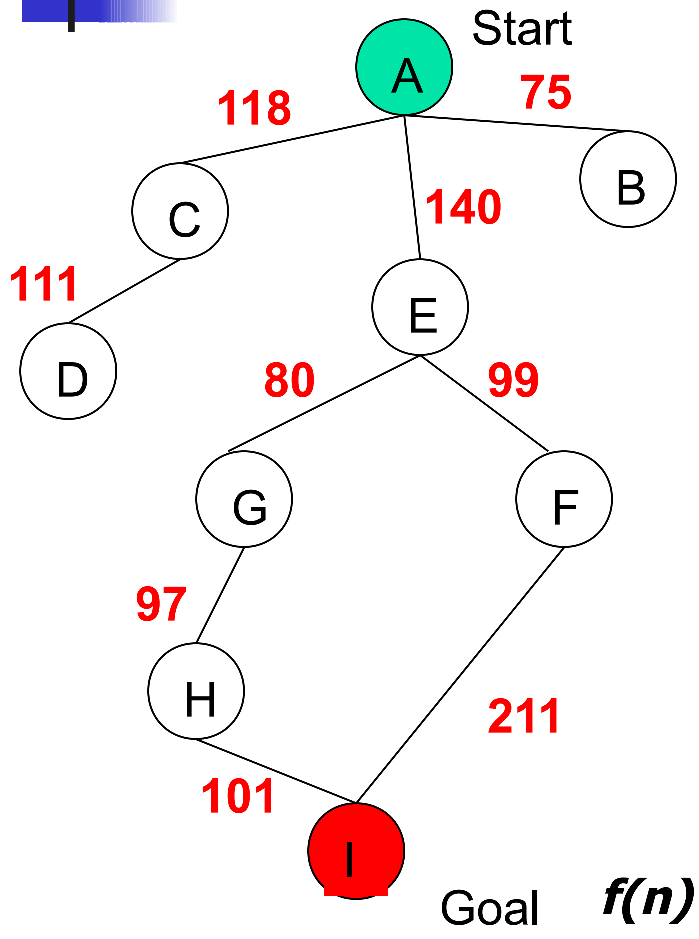
- A* uses a heuristic function which combines g(n) and h(n): f(n) = g(n) + h(n)
- **g(n)** is the exact cost to reach node *n* from the initial state.
- **h(n)** is an estimation of the remaining cost to reach the goal.

# A* (A Star)

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



# A* Search



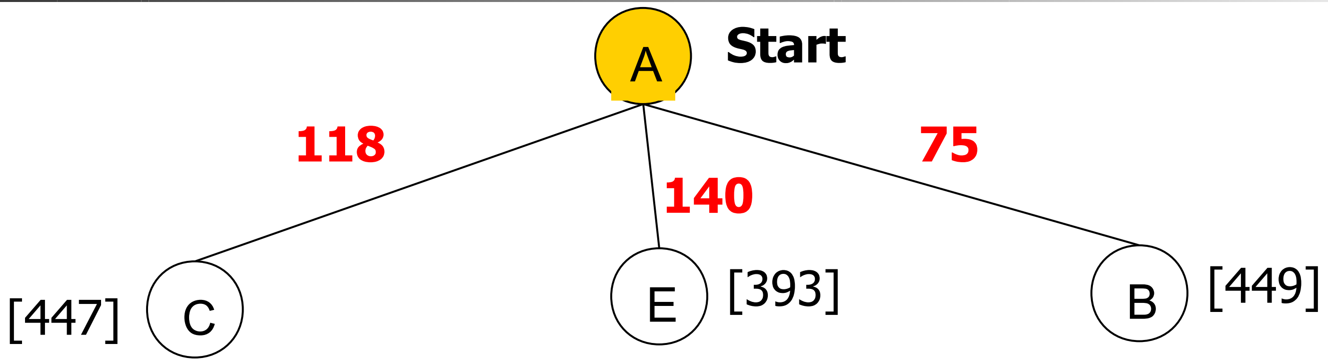
| State | Heuristic: h(n) |
|---|---|
| A | 366 |
| B | 374 |
| C | 329 |
| D | 244 |
| E | 253 |
| F | 178 |
| G | 193 |
| H | 98 |
| I | 0 |

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

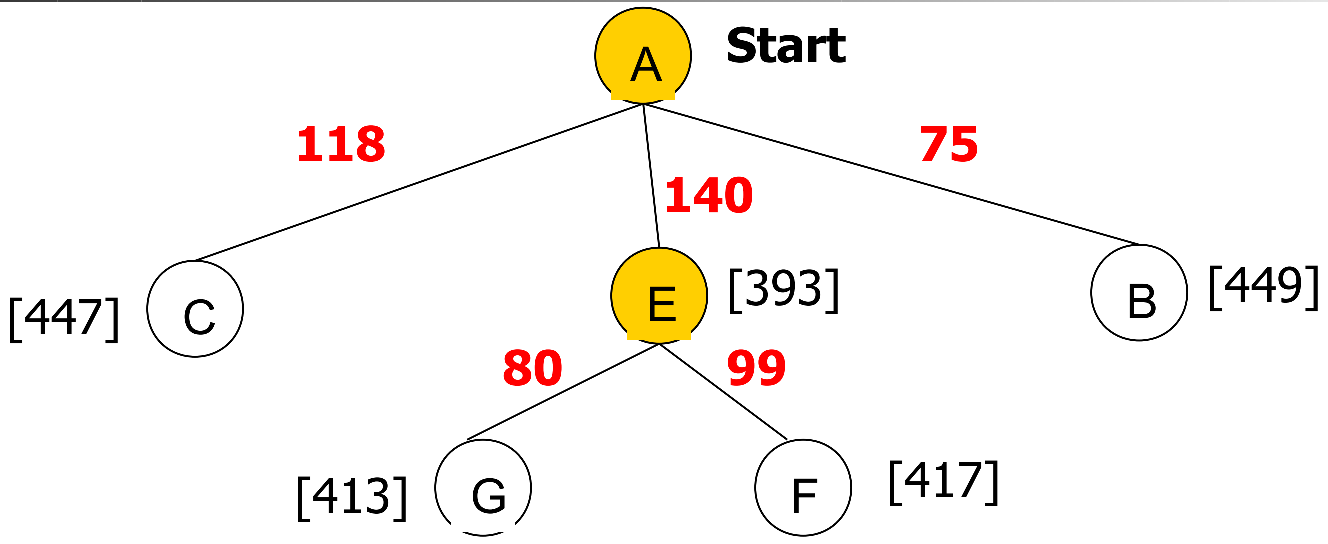
**g(n):** is the exact cost to reach node *n* from the initial state.

## A* Search: Tree Search

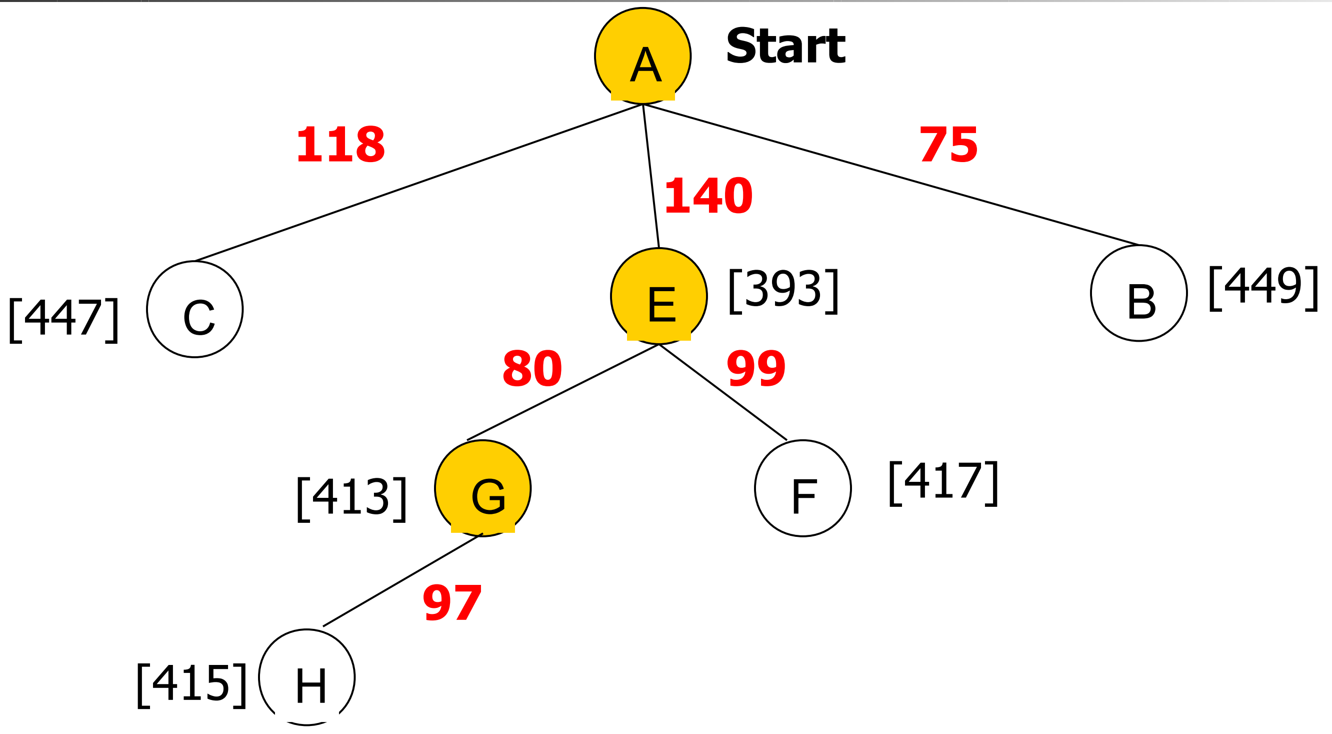
# A* Search: Tree Search



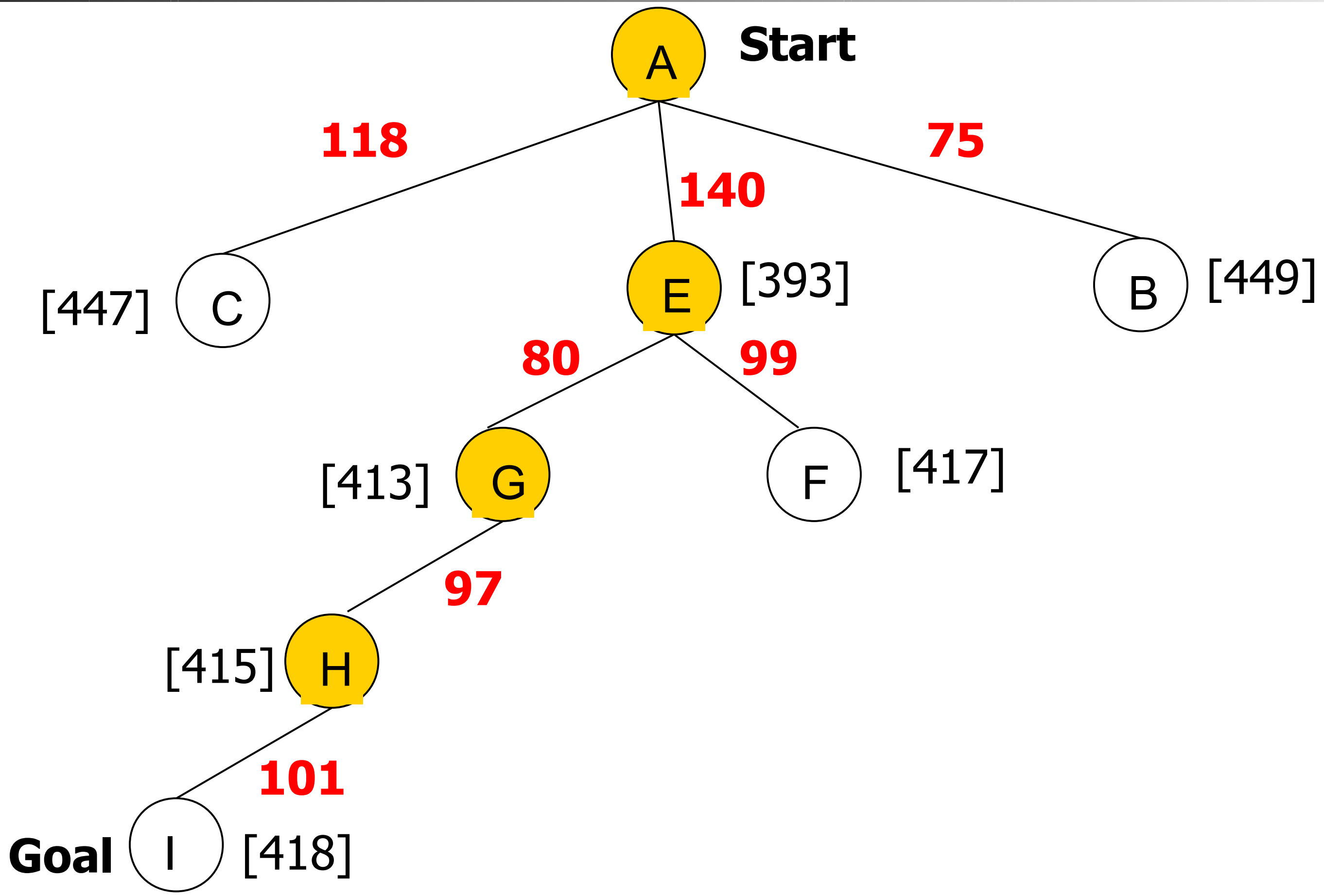
# A* Search: Tree Search



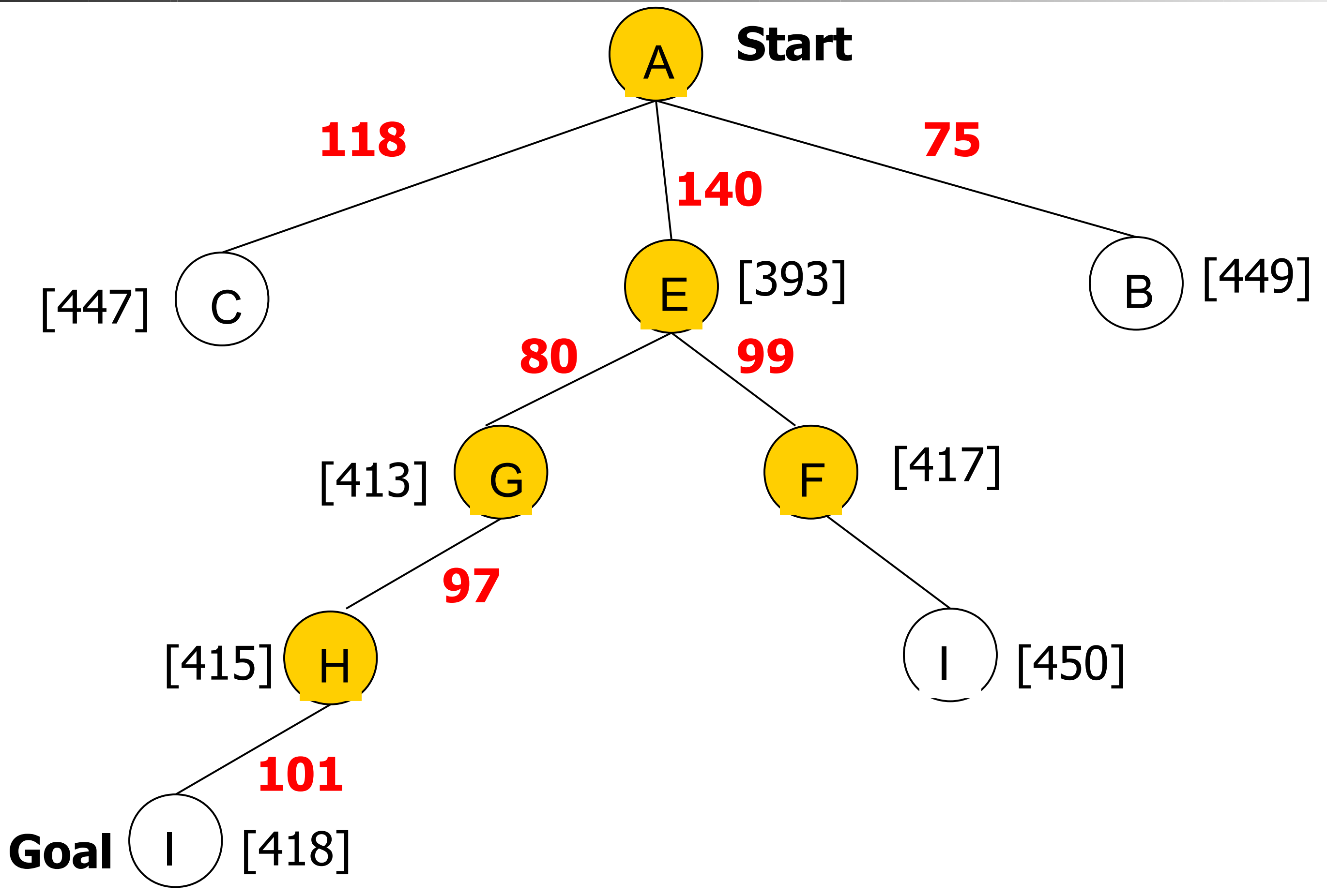
# A* Search: Tree Search



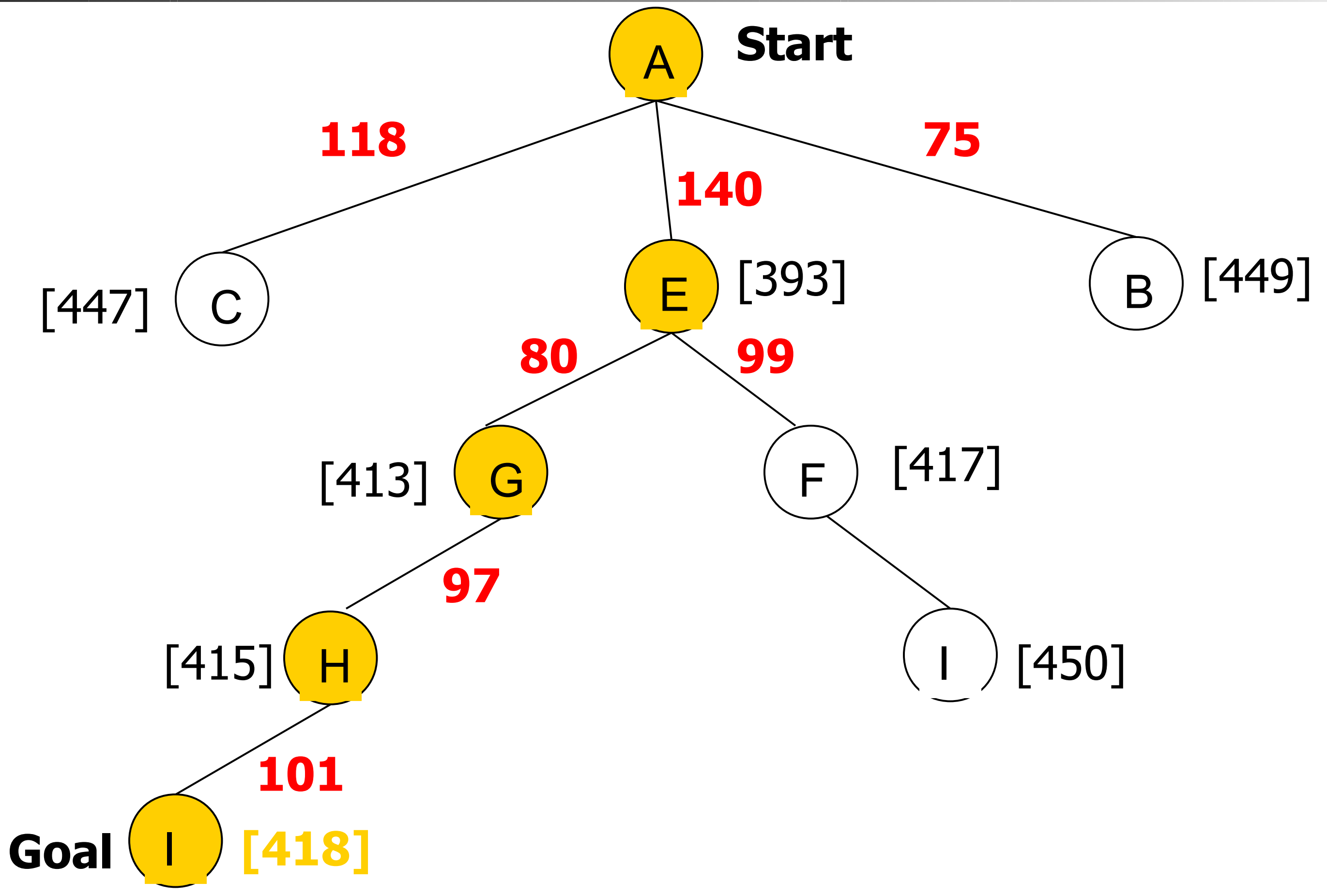
# A* Search: Tree Search



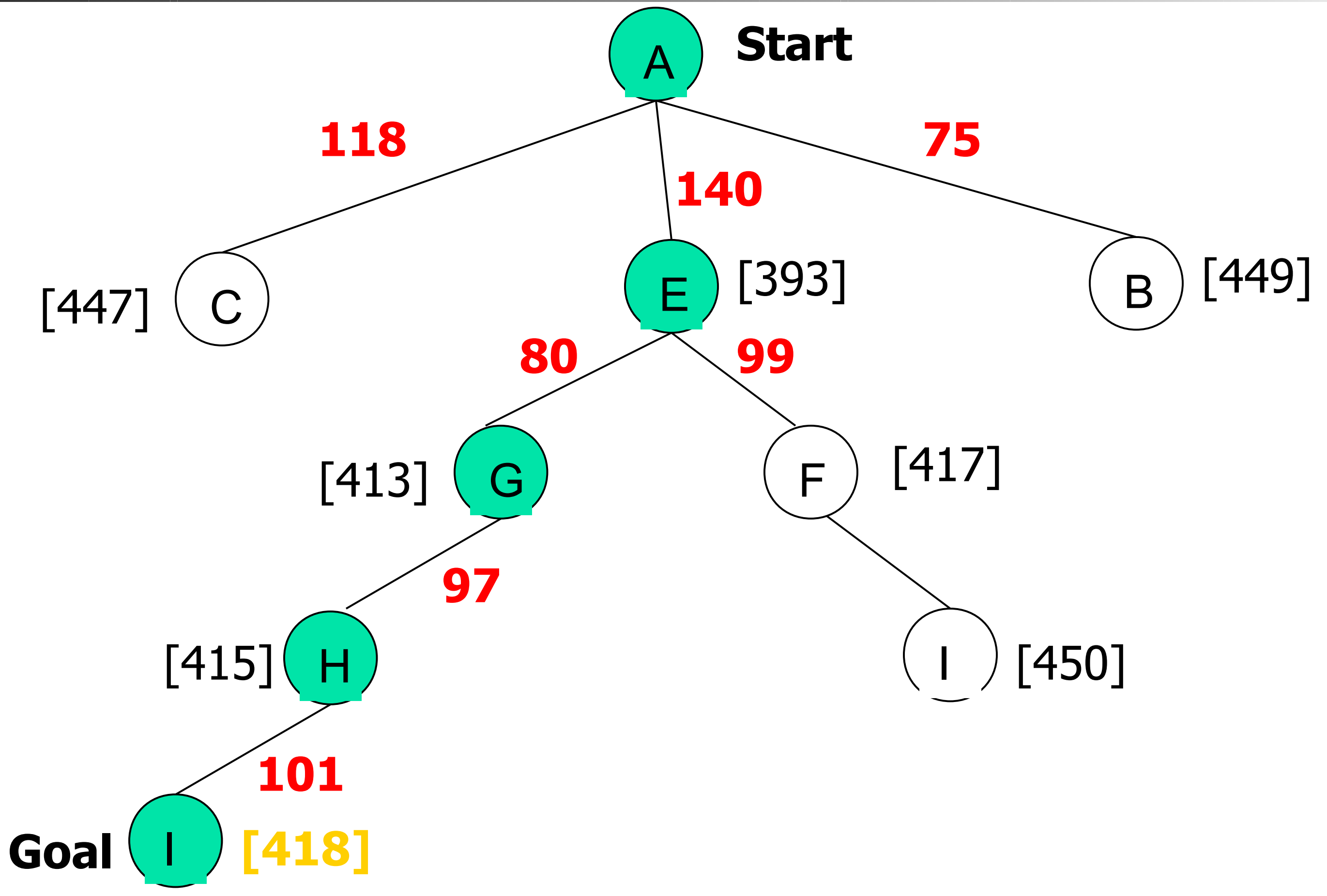
# A* Search: Tree Search



# A* Search: Tree Search



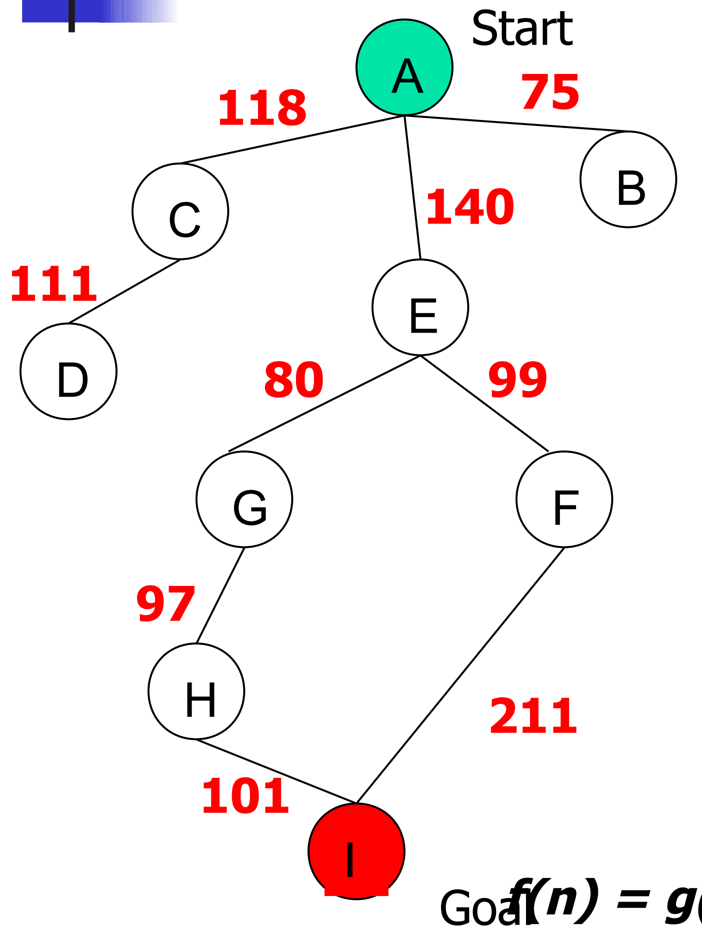
# A* Search: Tree Search



## A* with f() not Admissible

h() overestimates the cost to reach the goal state

# A* Search: *h* not admissible !



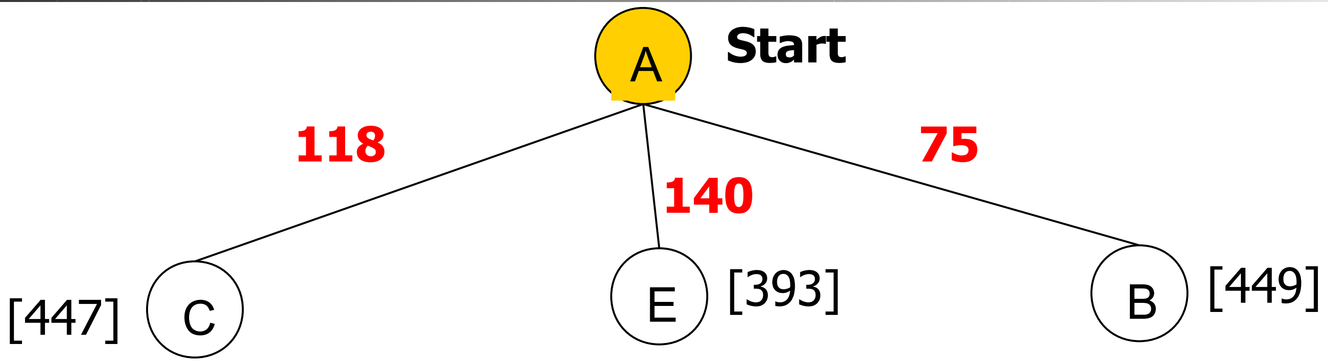
| State | Heuristic: h(n) |
|---|---|
| A | 366 |
| B | 374 |
| C | 329 |
| D | 244 |
| E | 253 |
| F | 178 |
| G | 193 |
| **H** | **138** |
| I | 0 |

***f(n) = g(n) + h* (*n*) – (H-I) Overestimated**

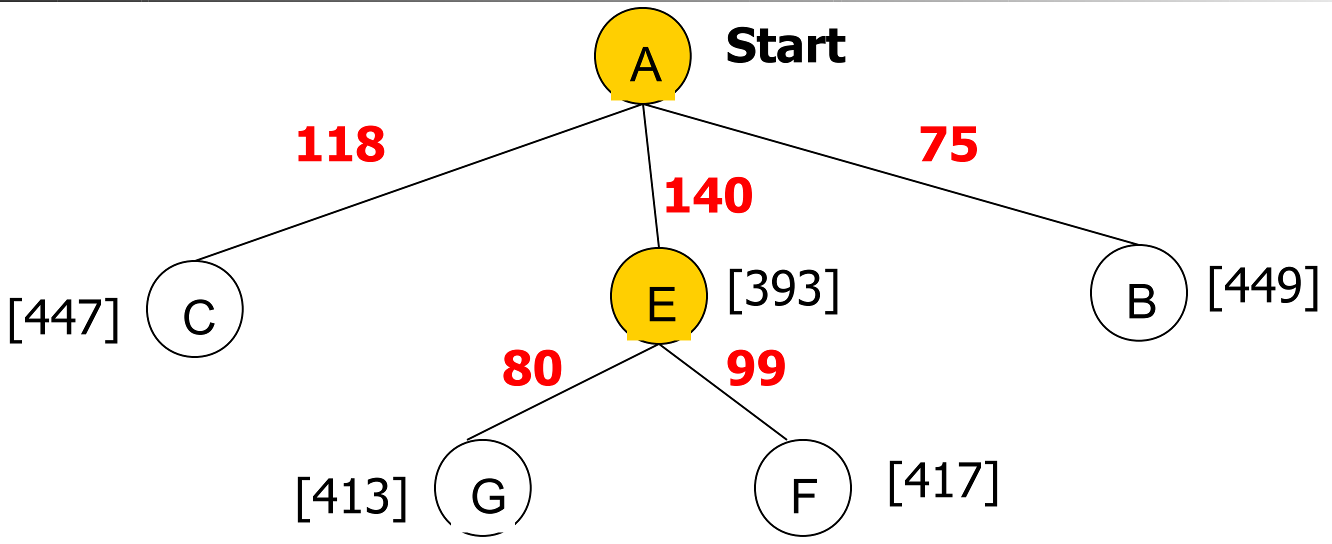
**g(n):** is the exact cost to reach node *n* from the initial state.

## A* Search: Tree Search

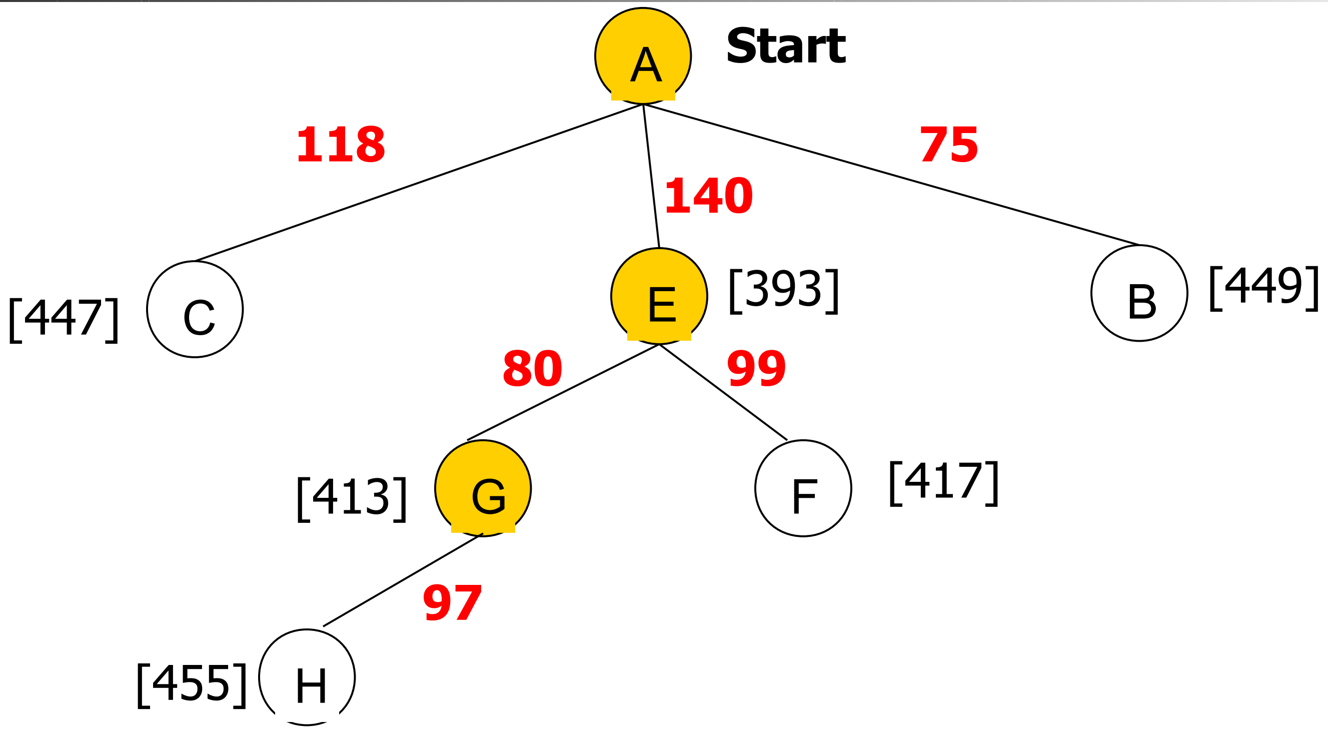
# A* Search: Tree Search



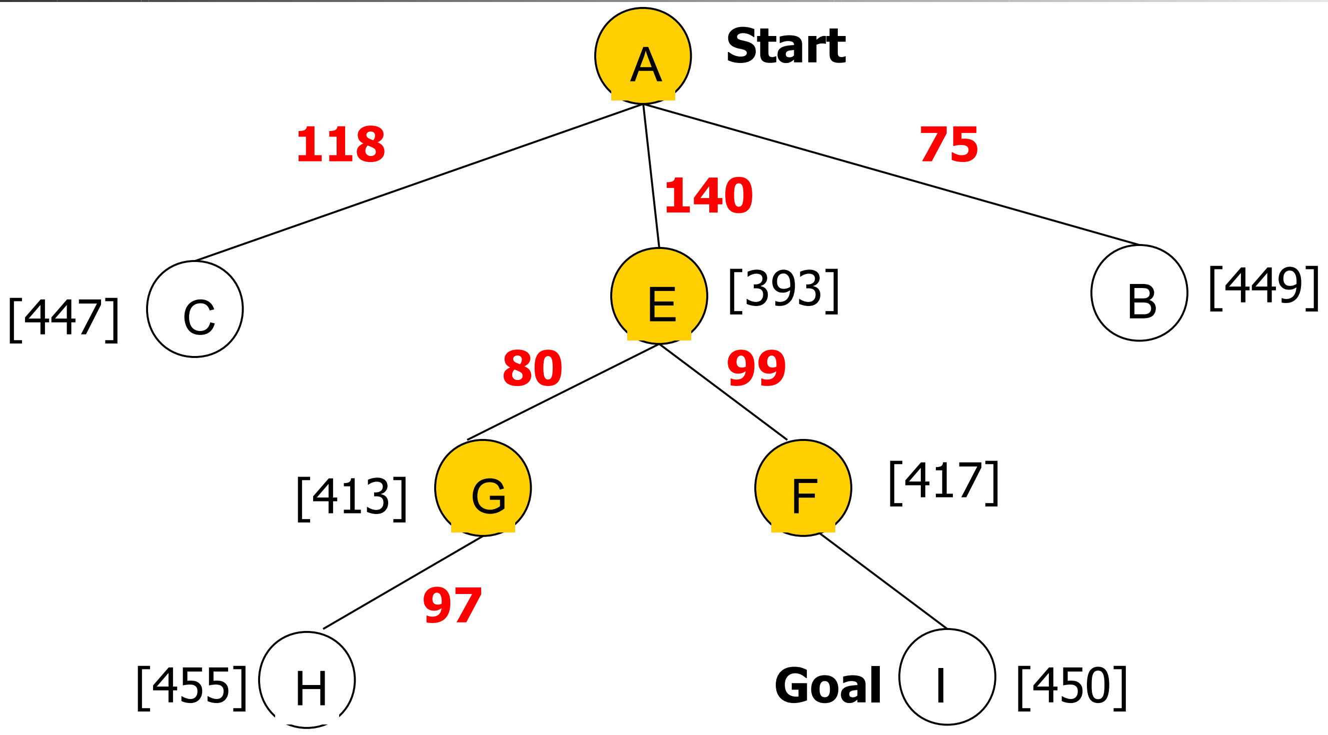
# A* Search: Tree Search



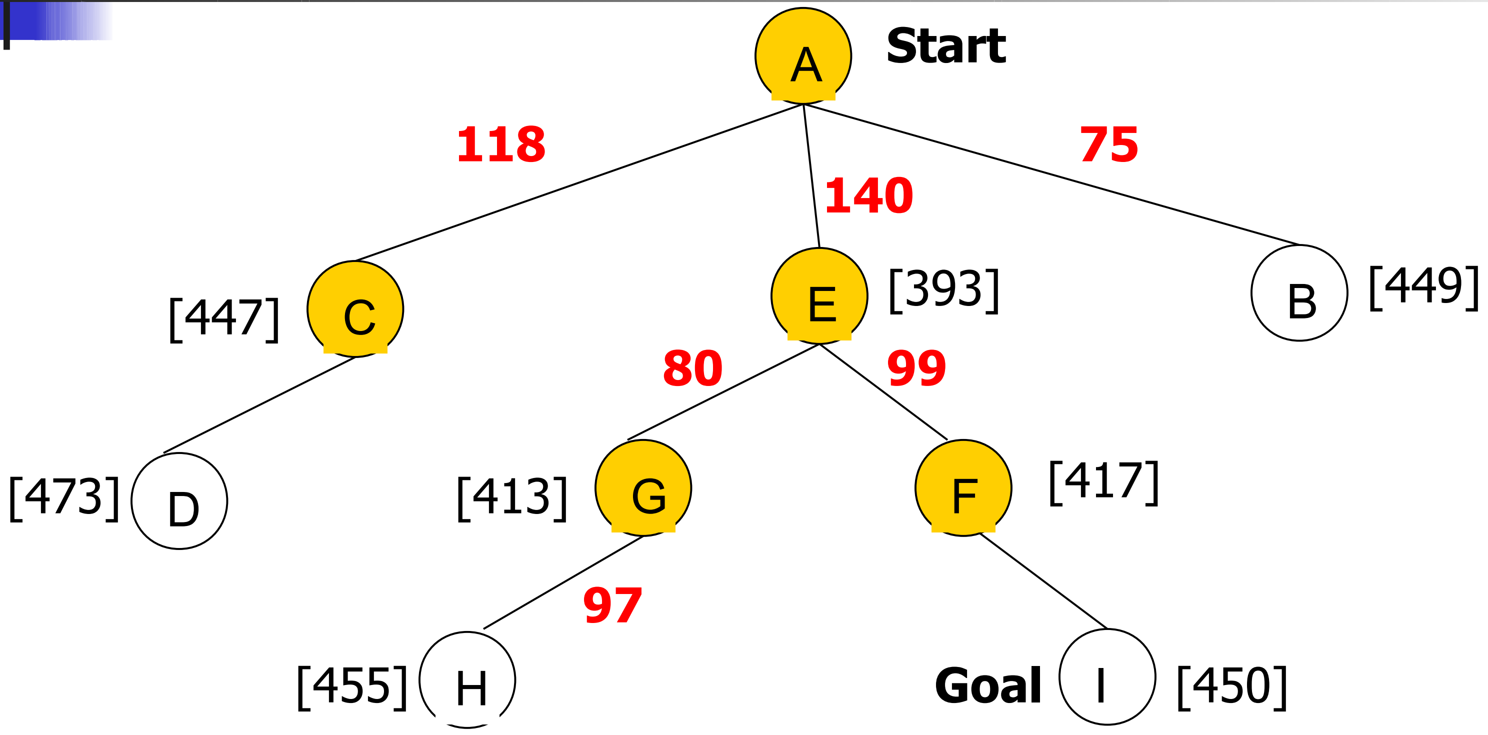
# A* Search: Tree Search



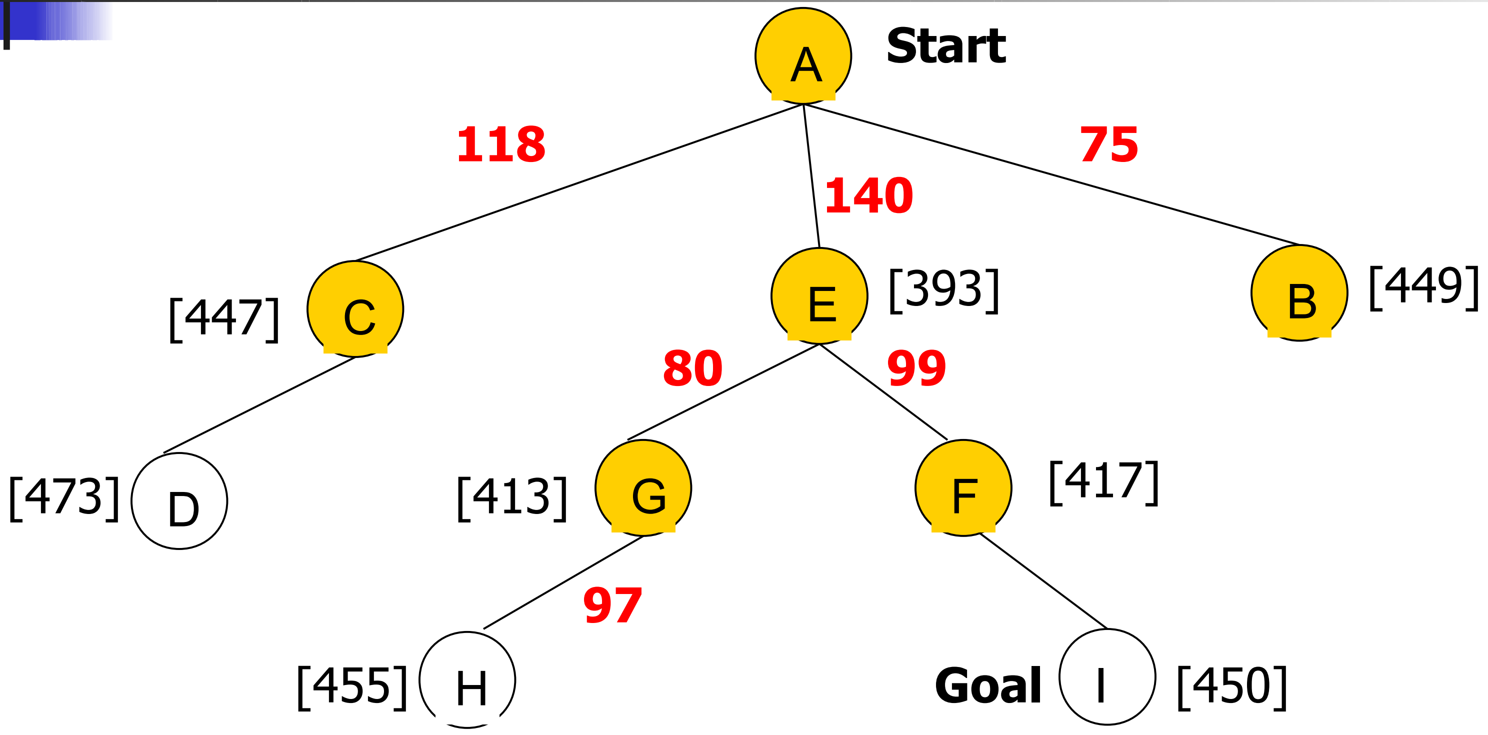
# A* Search: Tree Search



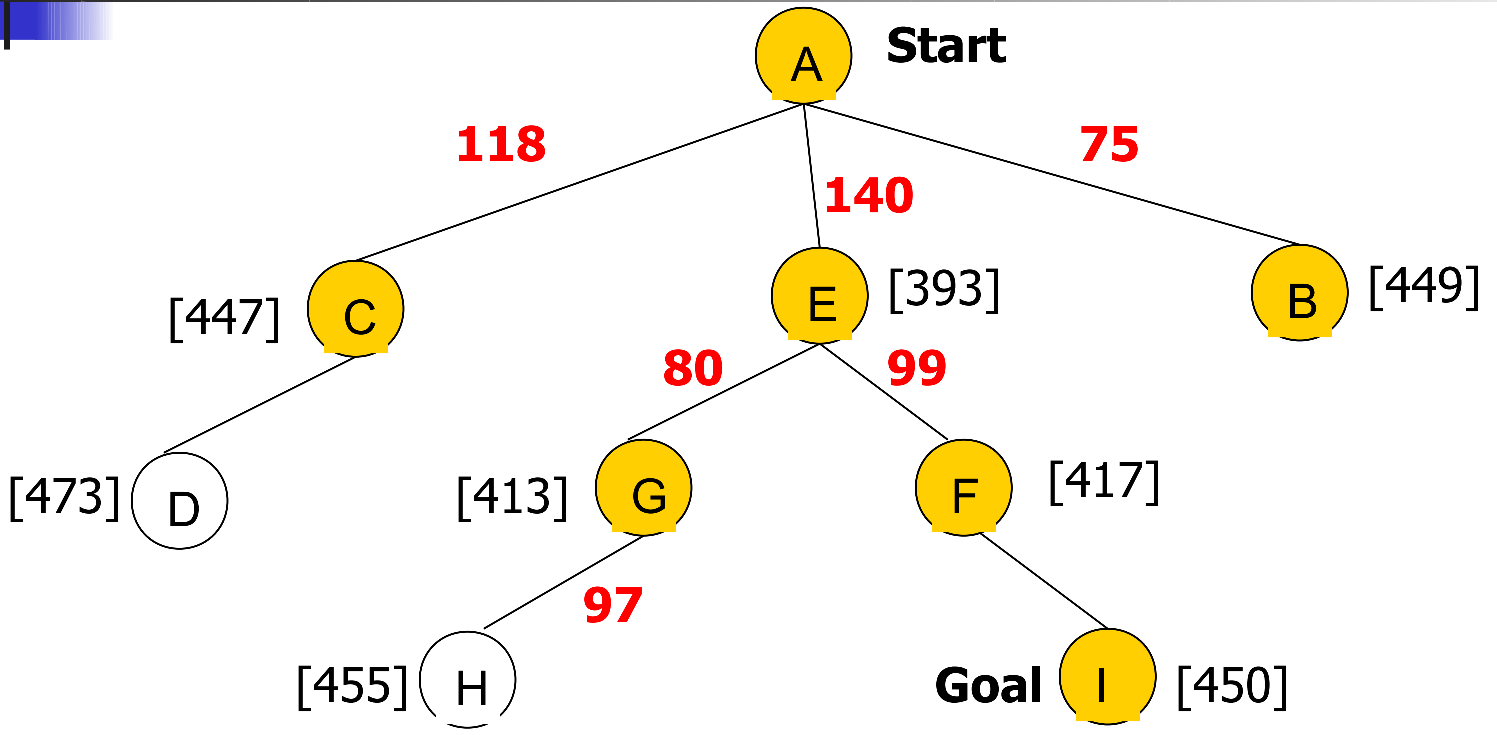
# A* Search: Tree Search



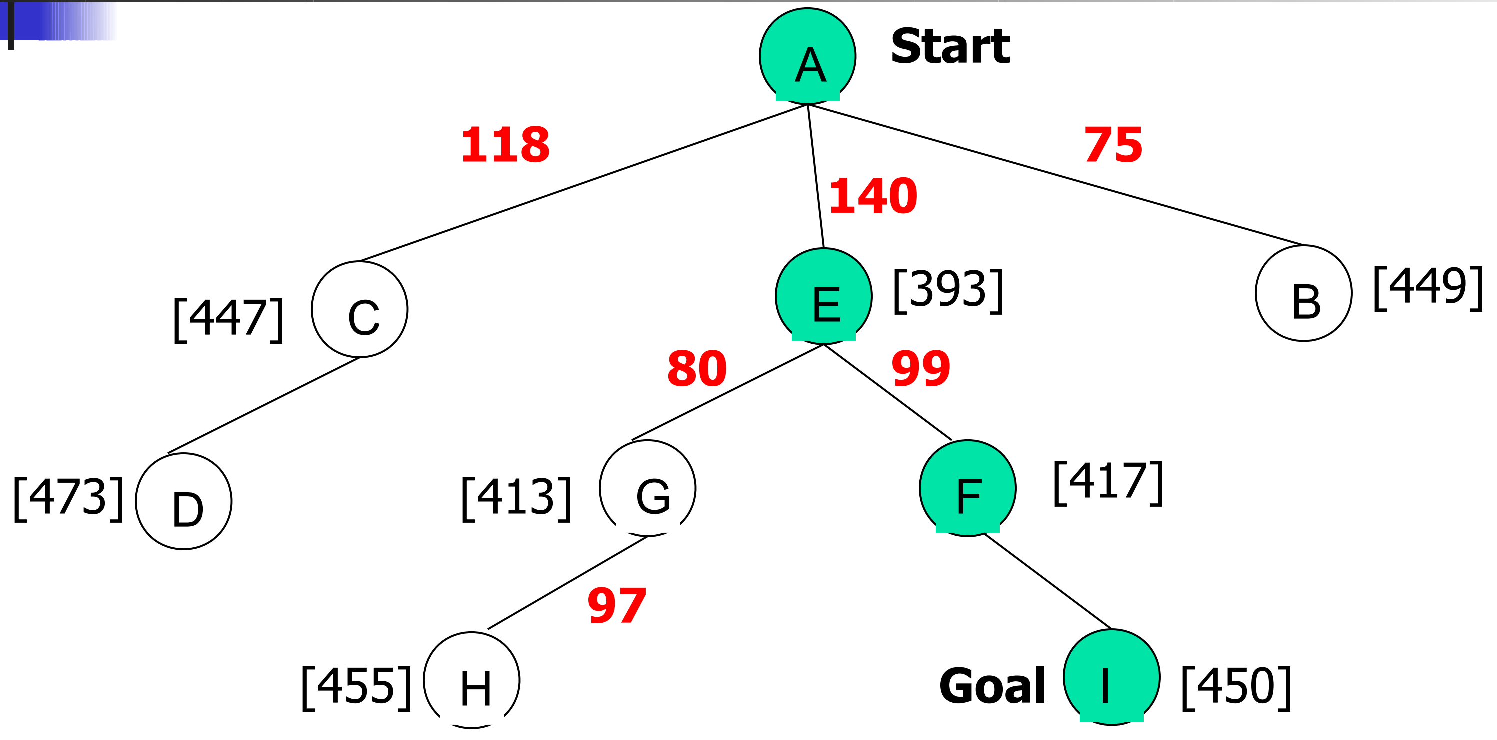
# A* Search: Tree Search



# A* Search: Tree Search



# A* Search: Tree Search



A* not optimal !!!

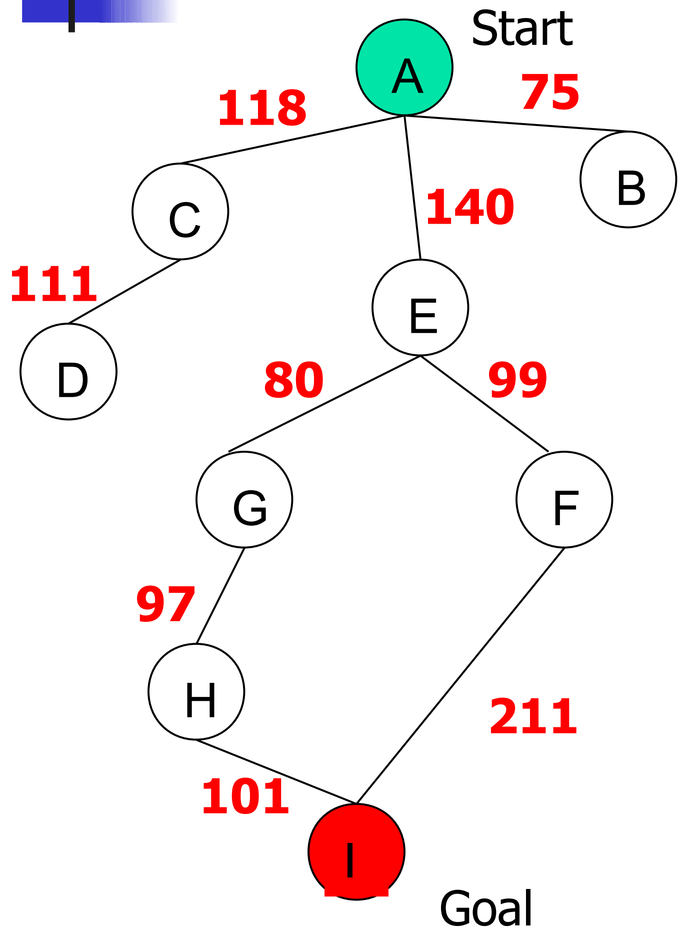
# A* Algorithm

A* with systematic checking for repeated states …

# A* Algorithm

```
1. Search queue Q is empty.
2. Place the start state s in Q with f value h(s).
3. If Q is empty, return failure.
4. Take node n from Q with lowest f value.
   (Keep Q sorted by f values and pick the first element).
5. If n is a goal node, stop and return solution.
6. Generate successors of node n.
7. For each successor n′ of n do:
      a) Compute f(n′) = g(n) + cost(n,n′) + h(n′).
      b) If n′ is new (never generated before), add n′ to Q.
      c) If node n′ is already in Q with a higher f value, replace it
         with current f(n′) and place it in sorted order in Q.
   End for
8. Go back to step 3.
```

# A* Search: Analysis



- A* is complete except if there is an infinity of nodes with f < f(G).
- A* is optimal if heuristic *h* is admissible.
- Time complexity depends on the quality of heuristic but is still exponential.
- For space complexity, A* keeps all nodes in memory. A* has worst case O(bd) space complexity, but an iterative deepening version is possible (IDA*).

# Informed Search Strategies

## Iterative Deepening A*

## Iterative Deepening A*:IDA*

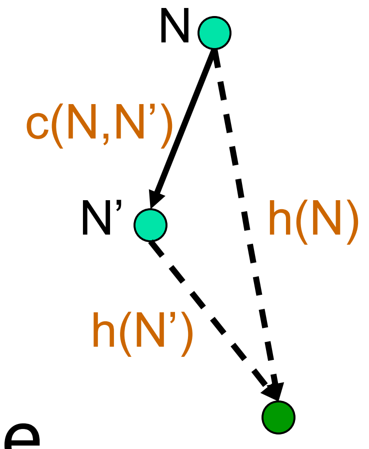
- Use $f(N) = g(N) + h(N)$ with admissible and consistent h
- Each iteration is depth-first with cutoff on the value of f of expanded nodes

# Consistent Heuristic

- The admissible heuristic h is consistent (or satisfies the monotone restriction) if for every node N and every successor N' of N:

  $h(N) \quad c(N,N') + h(N')$

  (triangular inequality)
- A consistent heuristic is admissible.



# IDA* Algorithm

- In the first iteration, we determine a **“f-cost limit” – cut-off value**

  f(n0) = g(n0) + h(n0) = h(n0), where n0 is the start node.

- We expand nodes using the **depth-first algorithm** and backtrack whenever f(n) for an expanded node n exceeds the cut-off value.

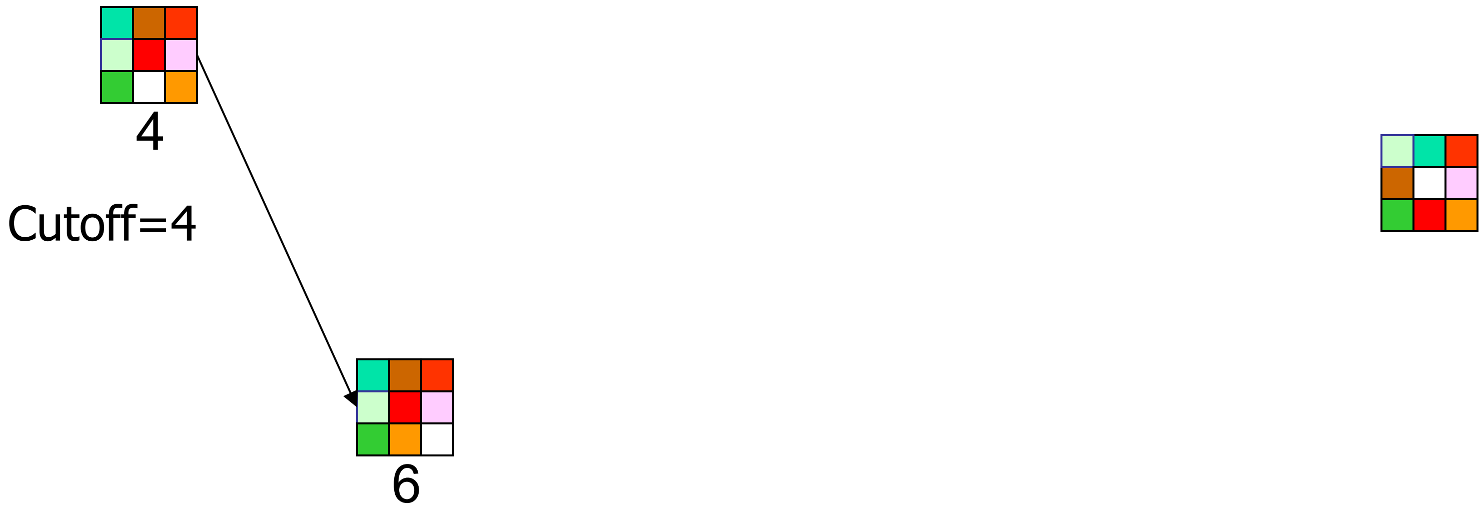
- If this search does not succeed, determine the **lowest f-value** among the nodes that were visited but not expanded.

- Use this f-value as the **new limit value – cut-off value** and do another depth-first search.

- Repeat this procedure until a goal node is found.

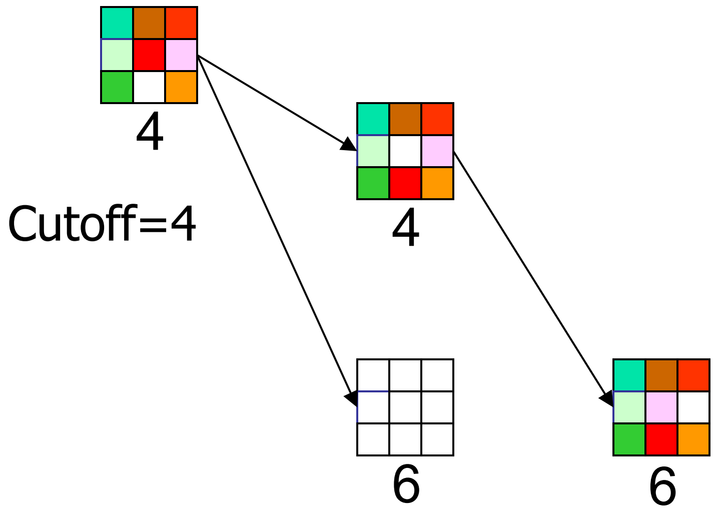
## 8-Puzzle

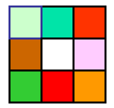
f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



# 8-Puzzle

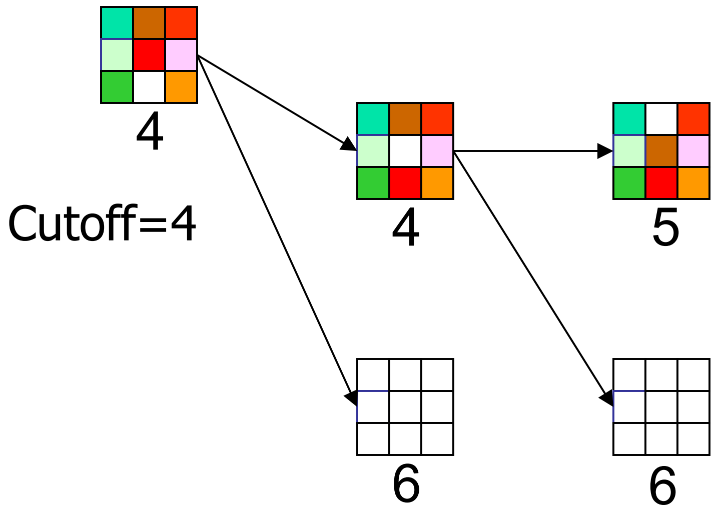
f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles

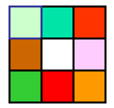




# 8-Puzzle

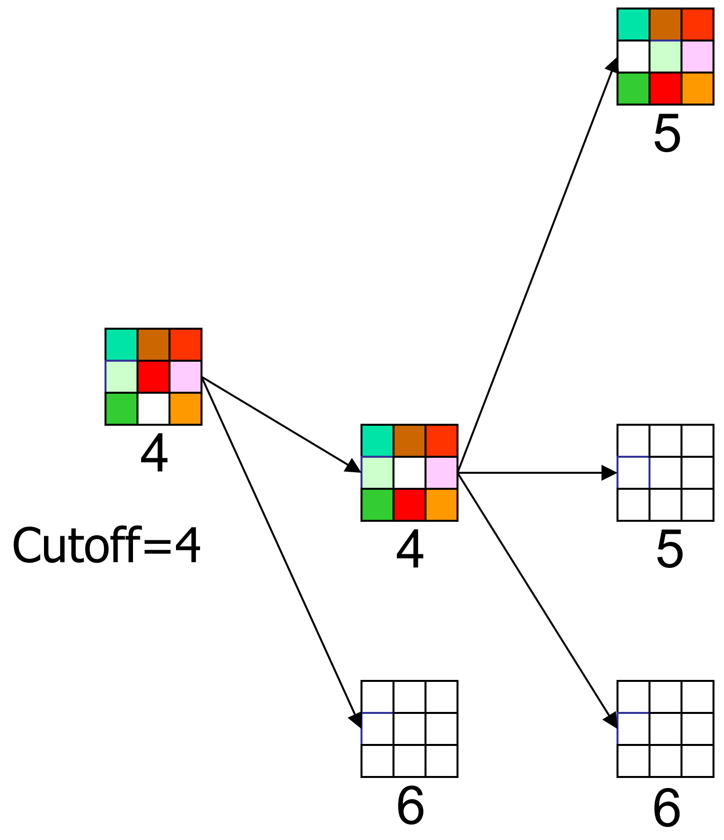
f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles

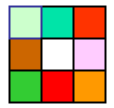




# 8-Puzzle

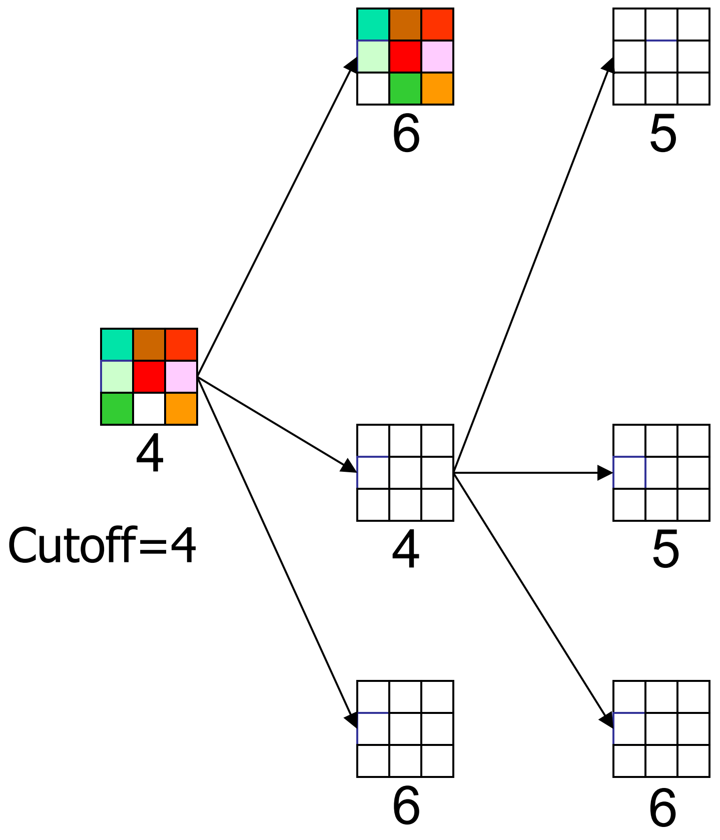
f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles

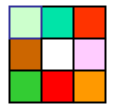




# 8-Puzzle

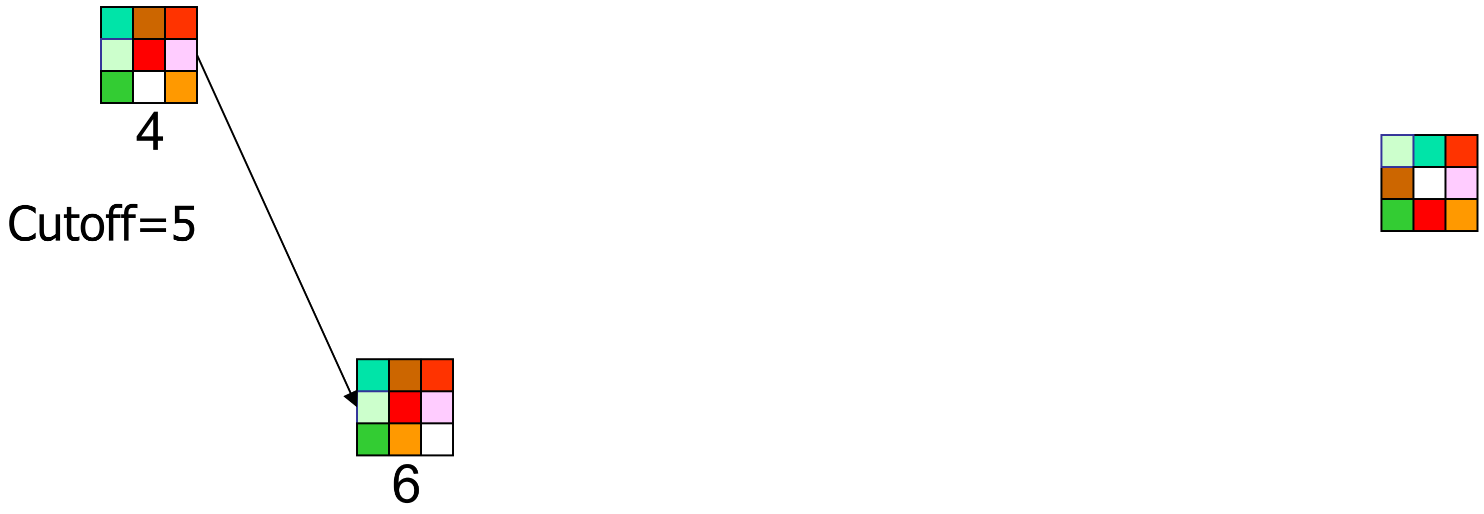
f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles





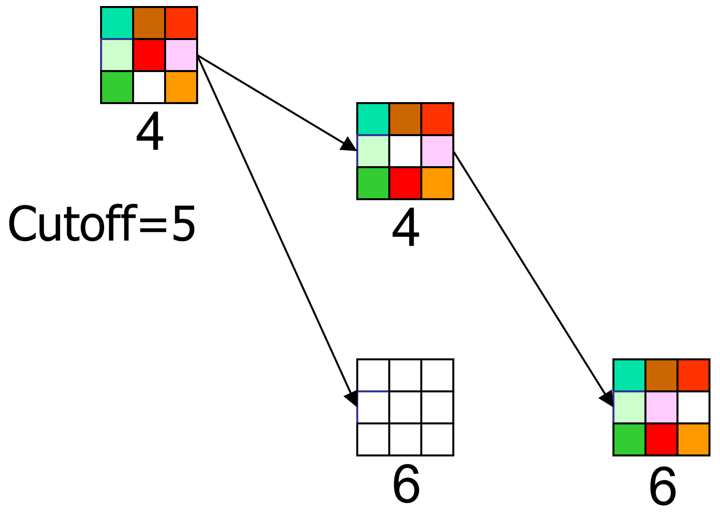
# 8-Puzzle

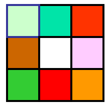
f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



# 8-Puzzle

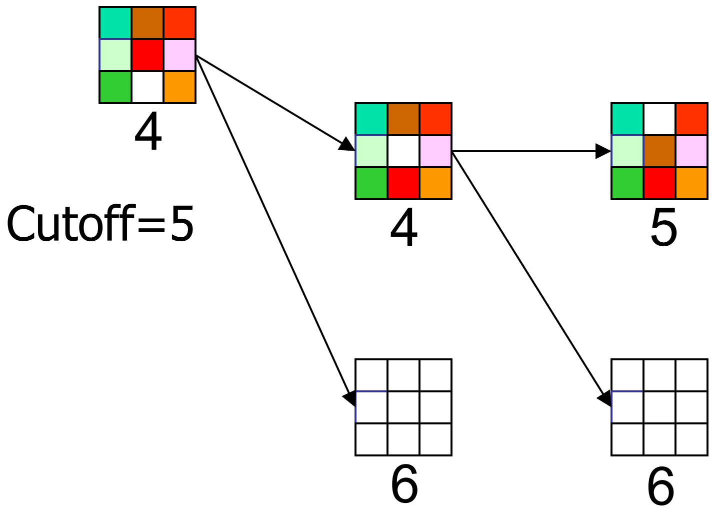
f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles

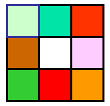




# 8-Puzzle

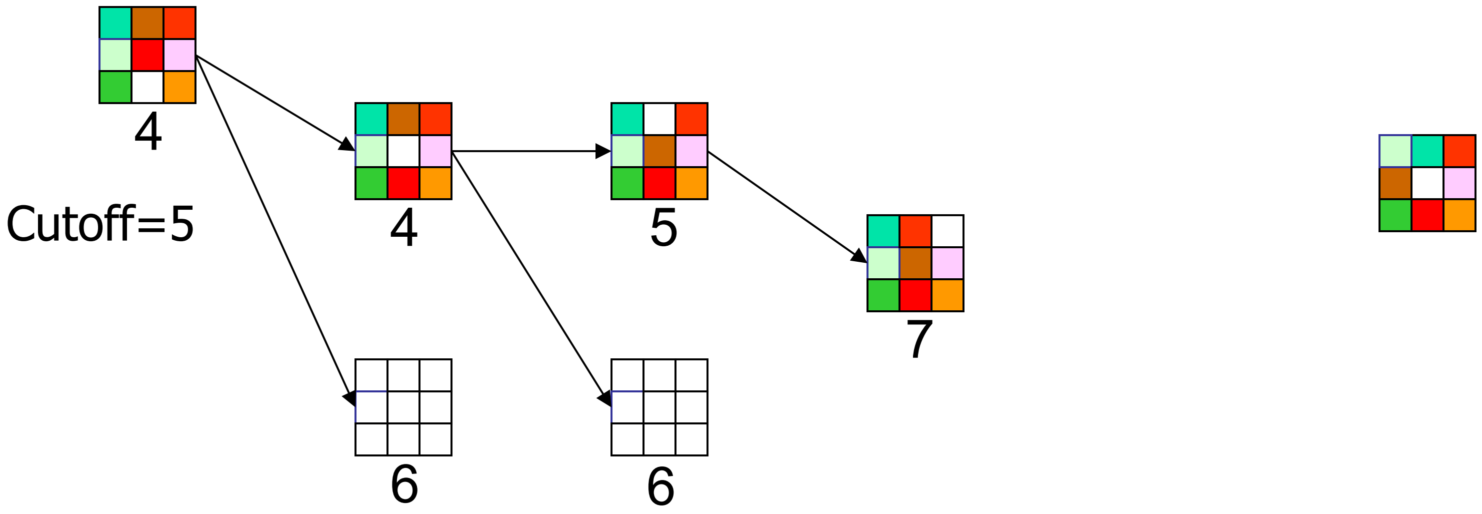
f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles





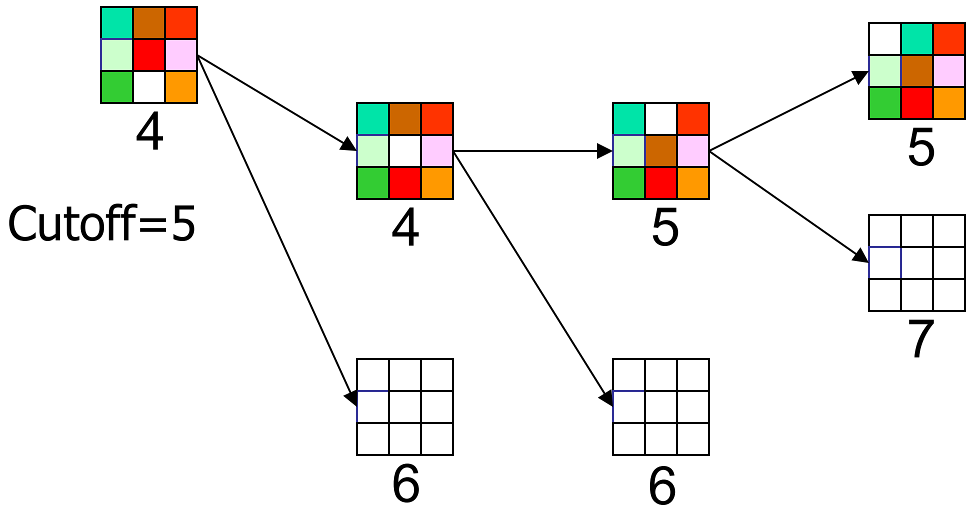
# 8-Puzzle

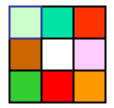
$f(N) = g(N) + h(N)$
with $h(N)$ = number of misplaced tiles



# 8-Puzzle

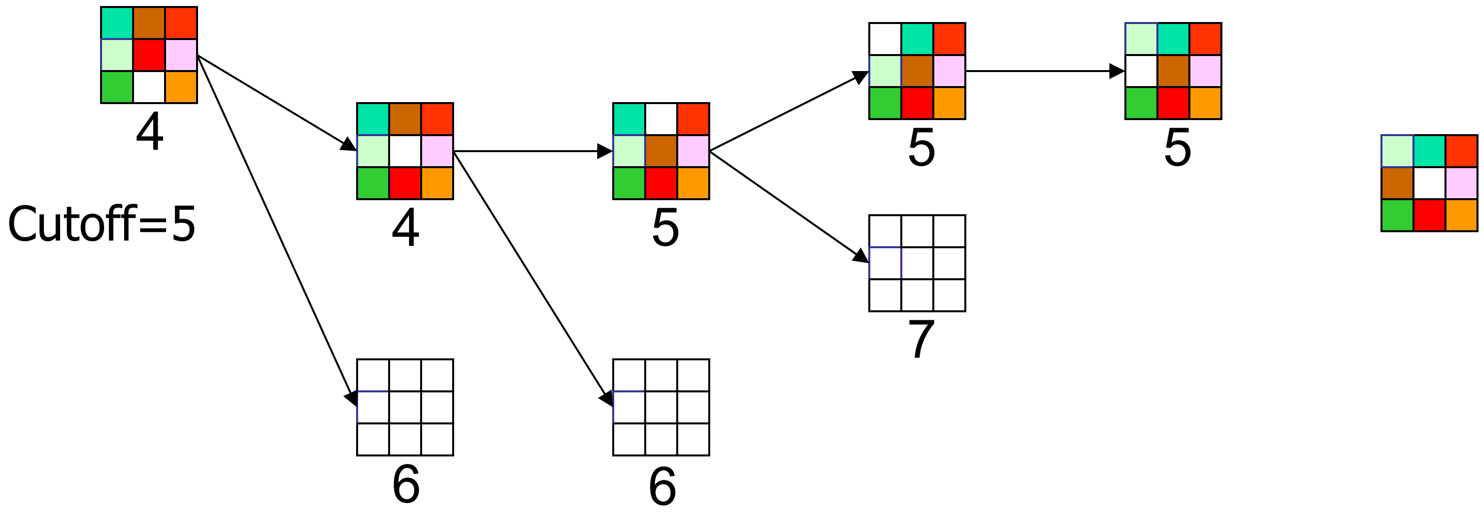
f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles





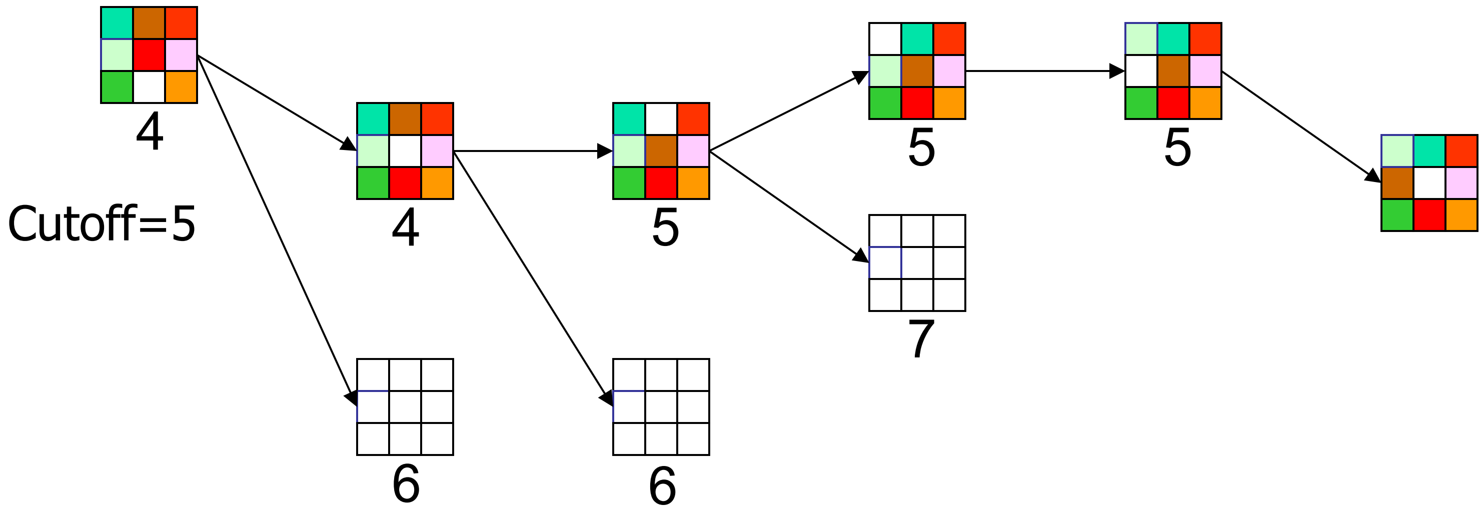
# 8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



# 8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



## When to Use Search Techniques

- The search space is small, and
  - There are no other available techniques, or
  - It is not worth the effort to develop a more efficient technique

- The search space is large, and
  - There is no other available techniques, and
  - There exist "good" heuristics

## Conclusions

- Frustration with *uninformed* search led to the idea of using domain specific knowledge in a search so that one can intelligently explore only the relevant part of the search space that has a good chance of containing the goal state. These new techniques are called informed (heuristic) search strategies.

- Even though heuristics improve the performance of informed search algorithms, they are still time consuming especially for large size instances.

## Admissible Heuristic

In computer science, specifically in algorithms related to pathfinding, a heuristic function is said to be admissible if it never overestimates the cost of reaching the goal, i.e. the cost it estimates to reach the goal is not higher than the lowest possible cost from the current point in the path.

An admissible heuristic is used to estimate the cost of reaching the goal state in an informed search algorithm. In order for a heuristic to be admissible to the search problem, the estimated cost must always be lower than or equal to the actual cost of reaching the goal state. The search algorithm uses the admissible heuristic to find an estimated optimal path to the goal state from the current node. For example, in **A*** search the evaluation function (where **n** is the current node) is:

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

where
$f(n)$ = the evaluation function.
$g(n)$ = the cost from the start node to the current node
$h(n)$ = estimated cost from current node to goal.

$h(n)$ is calculated using the heuristic function. With a non-admissible heuristic, the A* algorithm could overlook the optimal solution to a search problem due to an overestimation in $f(n)$.

Formulation

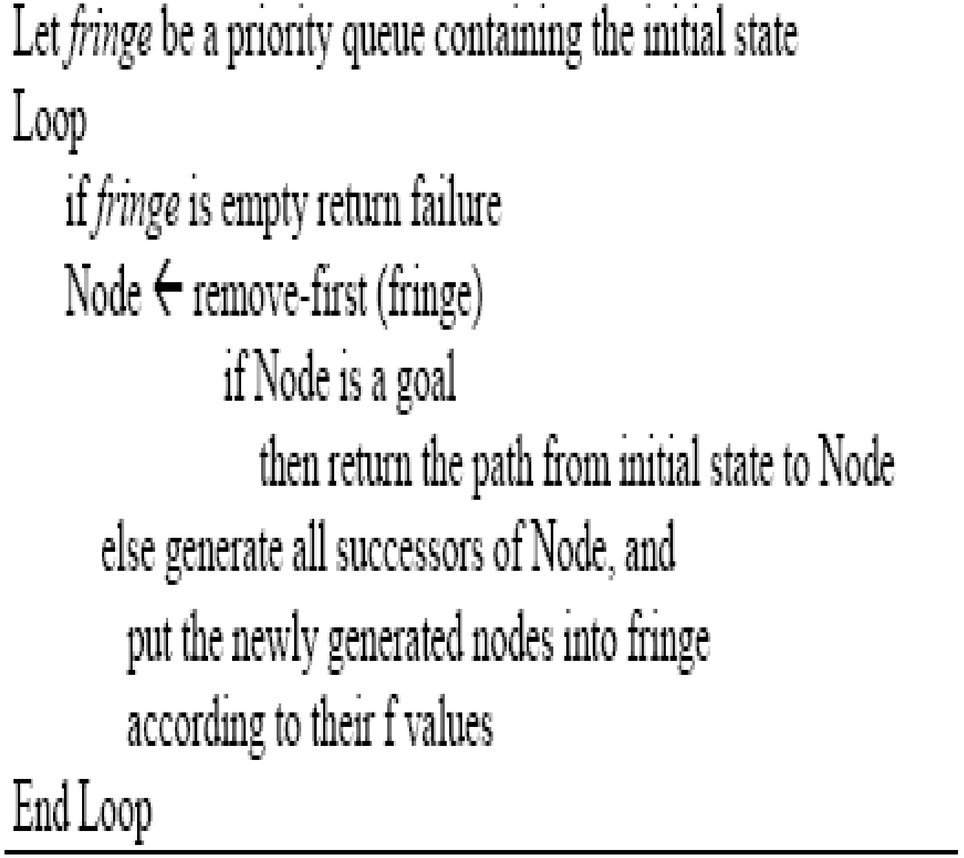
$n$ is a node
$h$ is a heuristic
$h(n)$ is cost indicated by h to reach a goal from
$h^*(n)$ is the actual cost to reach a goal from
$h(n)$ is admissible if, for all n

$$h(n) <= h^*(n)$$

# Heuristic Search Algorithm Best First and Branch and Bound Algorithms

## Best First Search

- Uniform Cost Search is a special case of the best first search algorithm. The algorithm maintains a priority queue of nodes to be explored. A cost function f(n) is applied to each node. The nodes are put in OPEN in the order of their f values. Nodes with smaller f(n) values are expanded earlier. The generic best first search algorithm is outlined below.



```
Let fringe be a priority queue containing the initial state
Loop
   if fringe is empty return failure
   Node ← remove-first (fringe)
            if Node is a goal
              then return the path from initial state to Node
     else generate all successors of Node, and
       put the newly generated nodes into fringe
       according to their f values
End Loop
```

## Concept

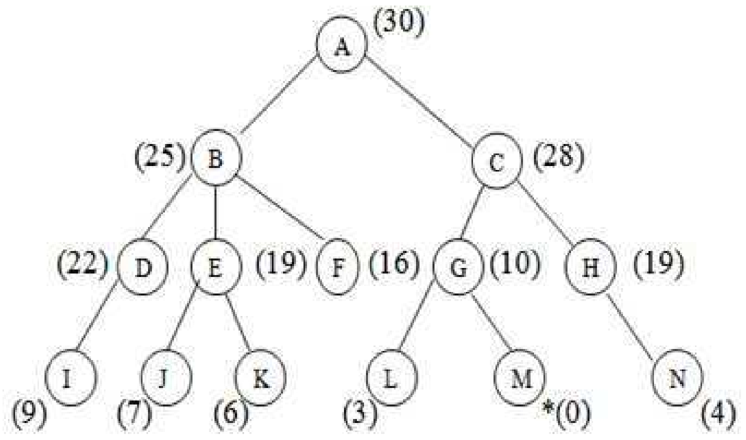
**Step 1:** Traverse the root node

**Step 2:** Traverse any neighbour of the root node, that is maintaining a least distance from the root node and insert them in ascending order into the queue.

**Step 3:** Traverse any neighbour of neighbour of the root node, that is maintaining a least distance from the root node and insert them in ascending order into the queue

**Step 4:** This process will continue until we are getting the goal node

# Example



**Step 1:**

Consider the node A as our root node. So the first element of the queue is A whish is not our goal node, so remove it from the queue and find its neighbour that are to inserted in ascending order.

A

**Step 2:**

The neighbours of A are B and C. They will be inserted into the queue in ascending order.

B C          A(Expanded Node)

**Step 3:**

Now B is on the FRONT end of the queue. So calculate the neighbours of B that are maintaining a least distance from the roof.

**Step 4:**

Now the node F is on the FRONT end of the queue. But as it has no further children, so remove it from the queue and proceed further.

E D C F (Expanded Node)

**Step 5:**

Now E is the FRONT end. So the children of E are J and K. Insert them into the queue in ascending order.

K J D C E(Expanded Node)

**Step 6:**

Now K is on the FRONT end and as it has no further children, so remove it and proceed further

J D C K(Expanded Node)

**Step7:**

Also, J has no corresponding children. So remove it and proceed further.

D C J(Expanded Node)

**Step 8:**

Now D is on the FRONT end and calculates the children of D and put it into the queue.

**Step 10:**

Now C is the FRONT node .So calculate the neighbours of C that are to be inserted in ascending order into the queue.

G H C(Expanded Node)

**Step 11:**

Now remove G from the queue and calculate its neighbour that is to insert in ascending order into the queue.

M L H G(Expanded Node)

**Step12:**

Now M is the FRONT node of the queue which is our goal node. So stop here and exit.

L H M(Expanded Node)

## Advantages

- It is more efficient than that of BFS and DFS.
- Time complexity of Best first search is much less than Breadth first search.
- The Best first search allows us to switch between paths by gaining the benefits of both breadth first and depth first search. Because, depth first is good because a solution can be found without computing all nodes and Breadth first search is good because it does not get trapped in dead ends.

## Disadvantages

- Sometimes, it covers more distance than our consideration.

# Branch and Bound

- Branch and Bound is an algorithmic technique which finds the optimal solution by keeping the best solution found so far.
- If partial solution can’t improve on the best it is abandoned, by this method the number of nodes which are explored can also be reduced.
- It also deals with the optimization problems over a search that can be presented as the leaves of the search tree.
- The usual technique for eliminating the sub trees from the search tree is called pruning.
- For Branch and Bound algorithm we will use stack data structure.

## Concept

**Step 1:** Traverse the root node.

**Step 2:** Traverse any neighbour of the root node that is maintaining least distance from the root node.

**Step 3:** Traverse any neighbour of the neighbour of the root node that is maintaining least distance from

the root node.

**Step 4:** This process will continue until we are getting the goal node.

# Algorithm

**Step 1:** PUSH the root node into the stack.

**Step 2:** If stack is empty, then stop and return failure.

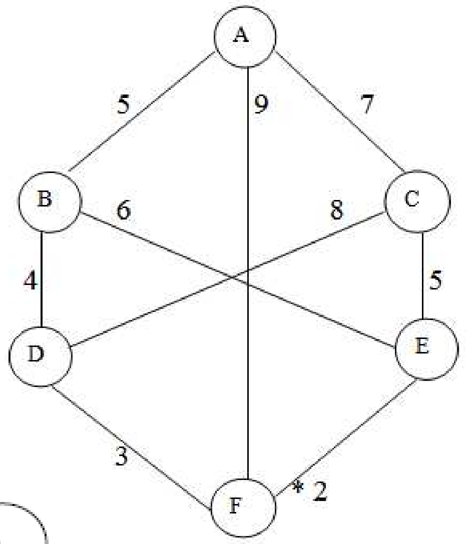
**Step 3:** If the top node of the stack is a goal node, then stop and return success.

**Step 4:** Else POP the node from the stack. Process it and find all its successors. Find out the path containing all its successors as well as predecessors and then PUSH the successors which are belonging to the minimum or shortest path.

**Step 5:** Go to step 2.

**Step 6:** Exit.

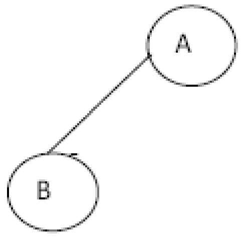
## Example



**Step 1:**

Consider the node A as our root node. Find its successors i.e. B, C, F. Calculate the distance from the root and PUSH them according to least distance.

A

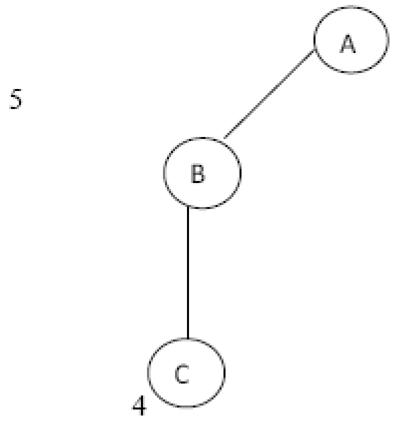


B: 0+5 = 5 (The cost of A is 0 as it is the starting node)

F: 0+9 = 9

C: 0+7 = 7

Here B (5) is the least distance.



**Step 2:**

Now the stack will be

C F B(Top)      A(Expanded)

As B is on the top of the stack so calculate the neighbours of B.

**Step 3:**

As the top of the stack is D. So calculate neighbours of D.

C F D(Top)            B(Expanded)

C: 0+5+4+8 = 17

F: 0+5+4+3 = 12

The least distance is F from D and it is our goal node. So stop and return success.

**Step 4:**

C F(Top)          D(Expanded)

Hence the searching path will be A-B -D-F

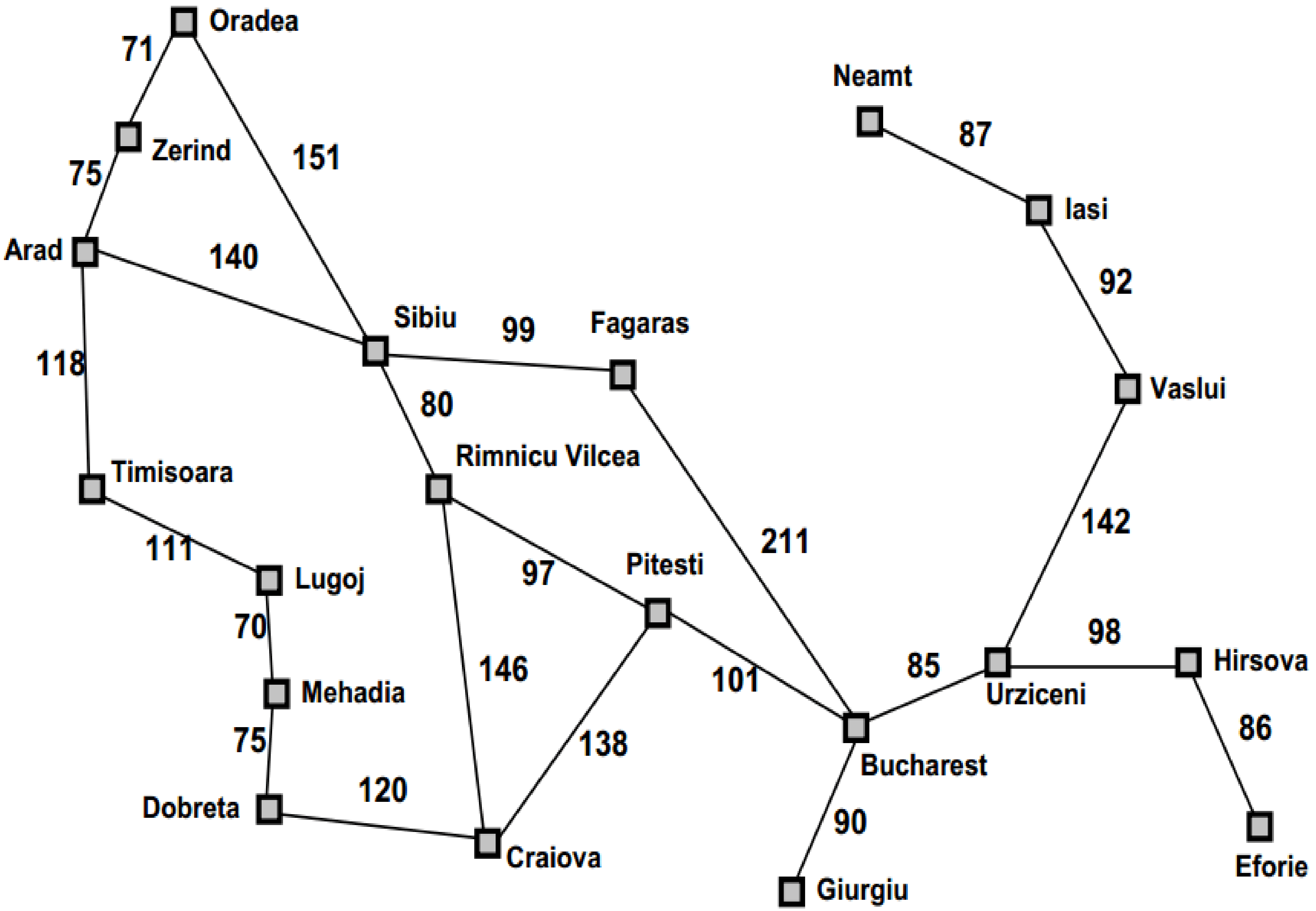
# Advantages/Disadvantages

**Advantages:**

- As it finds the minimum path instead of finding the minimum successor so there should not be any repetition.
- The time complexity is less compared to other algorithms.

**Disadvantages:**

- The load balancing aspects for Branch and Bound algorithm make it parallelization difficult.
- The Branch and Bound algorithm is limited to small size network. In the problem of large networks, where the solution search space grows exponentially with the scale of the network, the approach becomes relatively prohibitive.


Oradea
71
Zerind
151
75
Arad
140
Neamt
87
Iasi
92
Sibiu
99
Fagaras
118
80
Vaslui
Rimnicu Vilcea
Timisoara
211
142
111
97
Pitesti
Lugoj
70
98
Mehadia
146
85
Hirsova
101
Urziceni
75
138
86
Bucharest
120
Dobreta
90
Craiova
Eforie
Giurgiu

## Local Search and Optimization

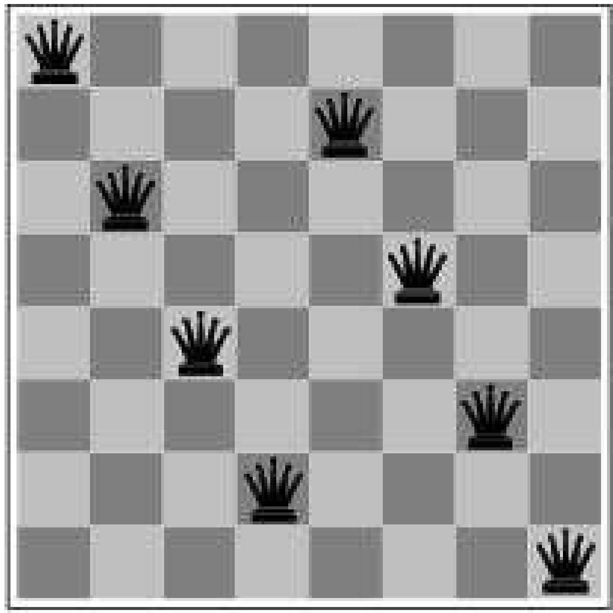
## Outline

- Local search techniques and optimization
  - Hill-climbing
  - Gradient methods
  - Simulated annealing
  - Issues with local search

## Local search and optimization

- Previously: systematic exploration of search space.
  - Backtrack search
  - Can solve n-queen problems for n = 200

- Different algorithms can be used
  - Local search
  - Can solve n-queen for n = 1,000,000



## Local search and optimization

- Local search
  - Keep track of single current state
  - Move only to neighboring states
  - Ignore paths

- Advantages:
  - Use very little memory
  - Can often find reasonable solutions in large or infinite (continuous) state spaces.

- “Pure optimization” problems
  - All states have an objective function
  - Goal is to find state with max (or min) objective value
  - Does not quite fit into CSP (satisfaction problem) formulation
  - Local search can do quite well on  these problems.

## Generated and Test

- Algorithm
  1. Generate a (potential goal) state:
     - Particular point in the problem space, or
     - A path from a start state
  2. Test if it is a goal state
     - Stop if positive
     - go to step 1 otherwise
- Systematic or Heuristic?
  - It depends on “Generate”

## Local search algorithms

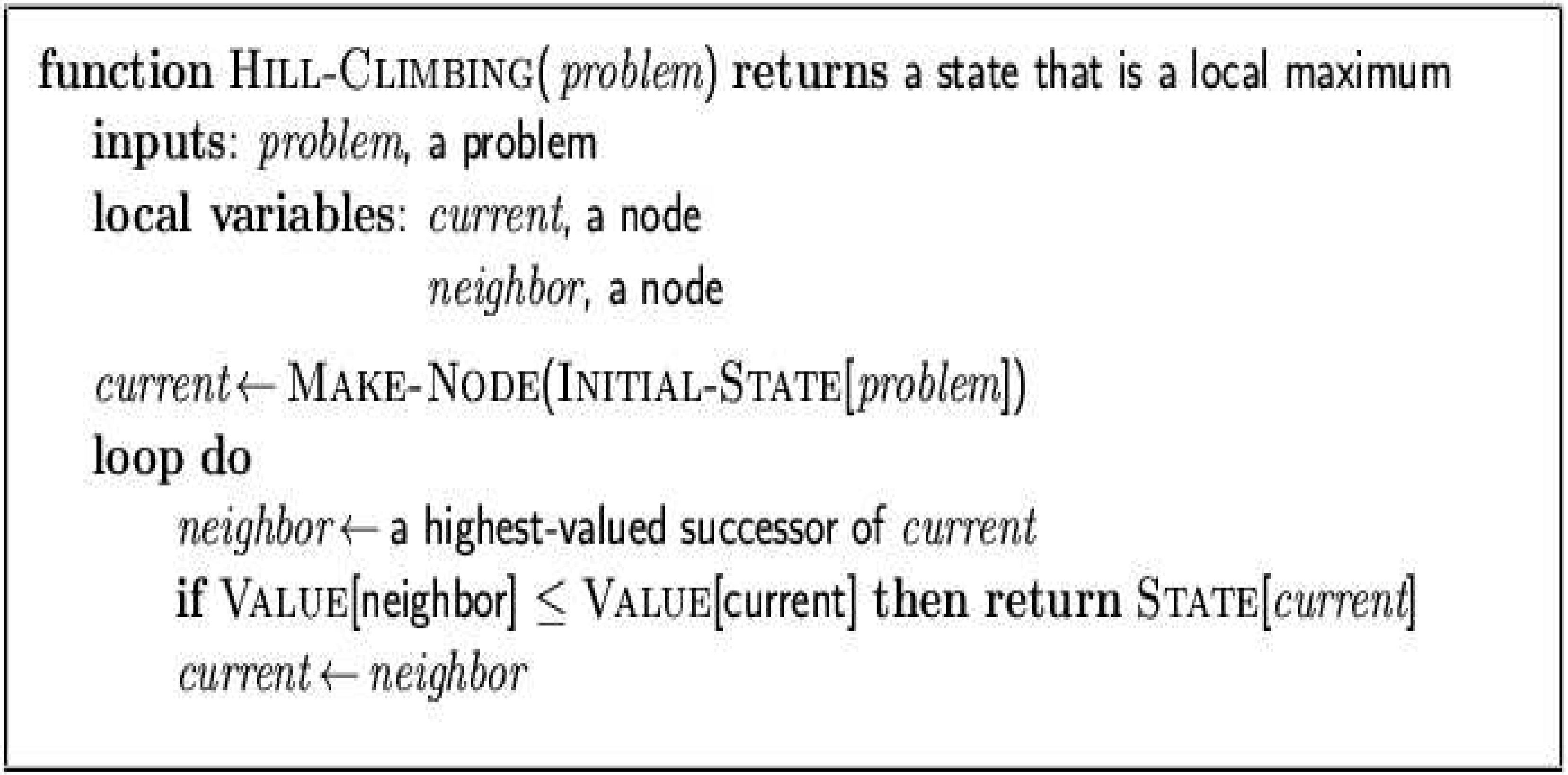
- In many problems, the path to the goal is irrelevant; the goal state itself is the solution
- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve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

## Hill Climbing

- Simple Hill Climbing
  - expand the current node
  - evaluate its children one by one (using the heuristic evaluation function)
  - choose the FIRST node with a better value
- Steepest Ascend Hill Climbing
  - expand the current node
  - Evaluate all its children (by the heuristic evaluation function)
  - choose the BEST node with the best value

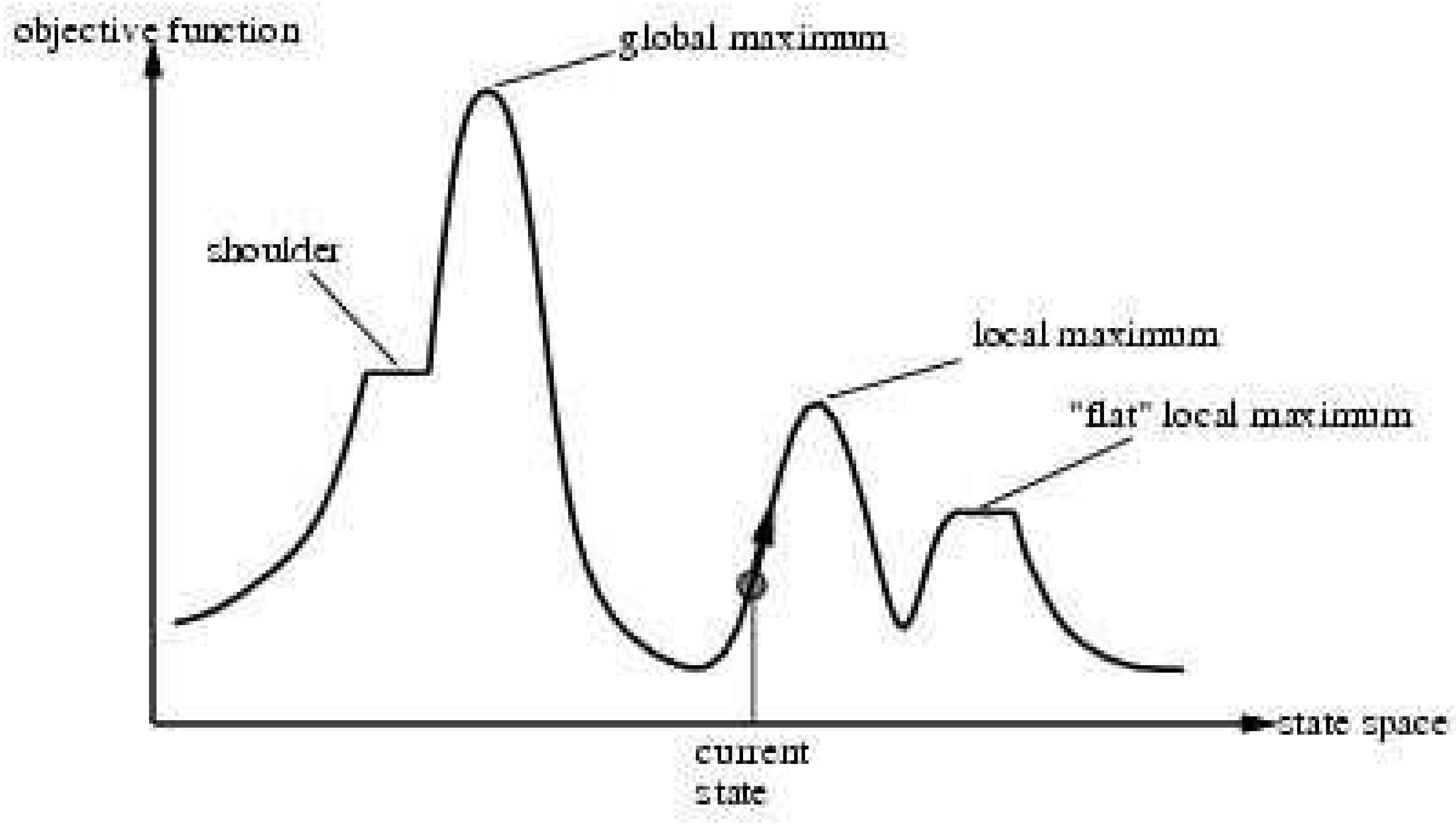
## Hill-climbing

- Steepest Ascend



```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

## “Landscape” of search for max value

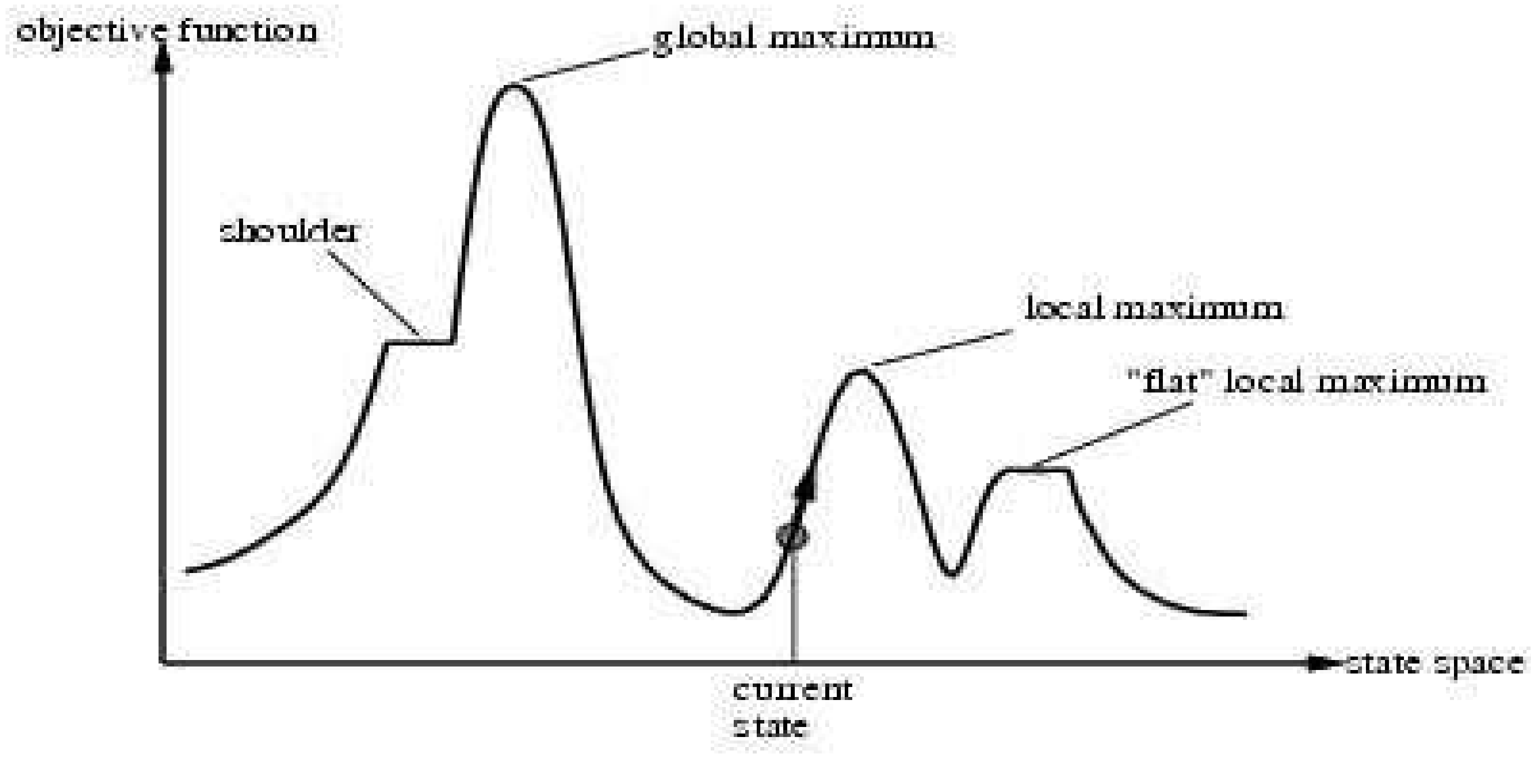


## Hill-climbing search

- “a loop that continuously moves in the direction of increasing value”
  - terminates when a peak is reached
  - Aka greedy local search
- Value can be either
  - Objective function value
  - Heuristic function value (minimized)
- Hill climbing does not look ahead of the immediate neighbors of the current state.
- Can randomly choose among the set of best successors, if multiple have the best value
- Characterized as “trying to find the top of Mount Everest while in a thick fog”

## Hill climbing and local maxima

- When local maxima exist, hill climbing is suboptimal
- Simple (often effective) solution
  - Multiple random restarts



## Hill-climbing example

- 8-queens problem, complete-state formulation
  - All 8 queens on the board in some configuration

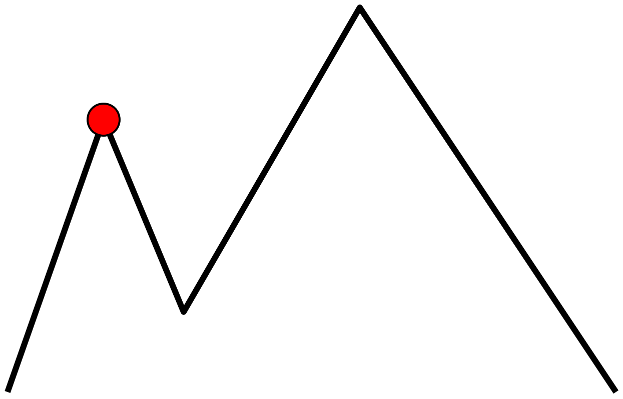
- Successor function:
  - move a single queen to another square in the same column.

- Example of a heuristic function *h(n)*:
  - the number of pairs of queens that are attacking each other (directly or indirectly)
  - (so we want to minimize this)

## Hill Climbing: Disadvantages

Local maximum

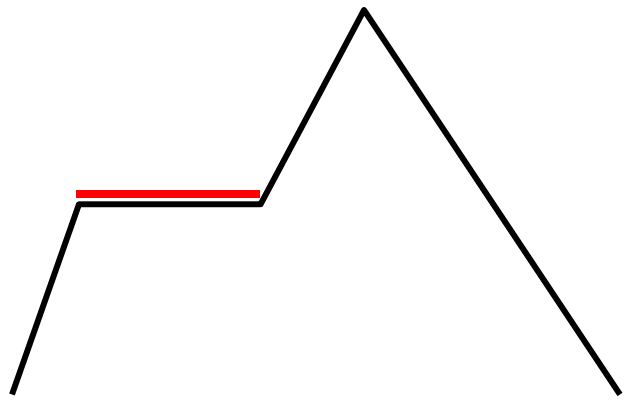
A state that is better than all of its neighbours, but not better than some other states far away.



## Hill Climbing: Disadvantages

Plateau

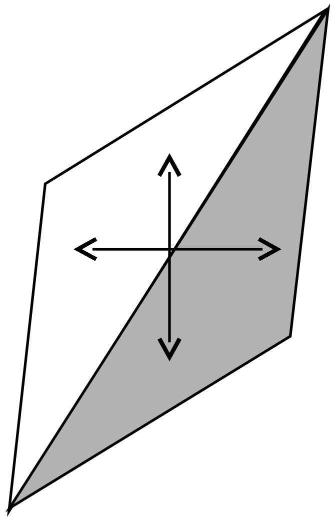
A flat area of the search space in which all neighbouring states have the same value.



## Hill Climbing: Disadvantages

Ridge

The orientation of the high region, compared to the set of available moves, makes it impossible to climb up. However, two moves executed serially may increase the height.



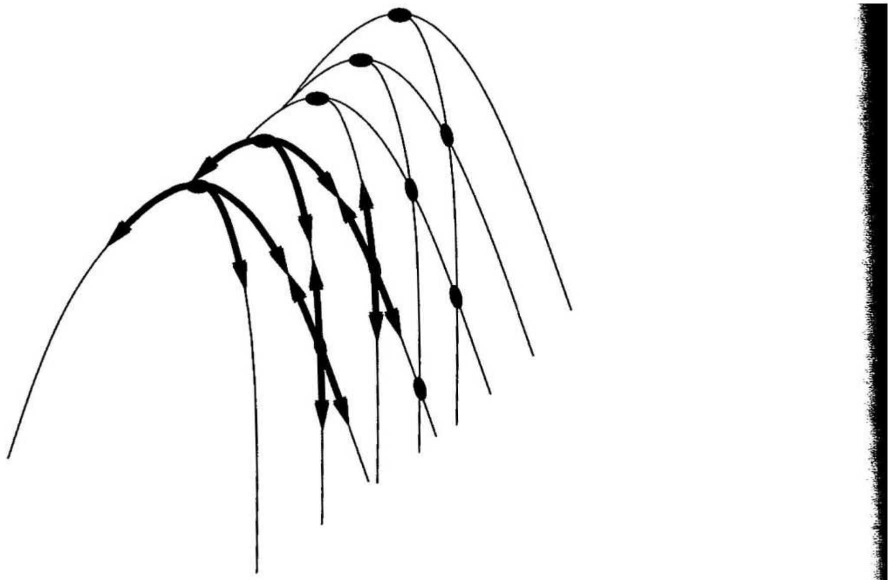


**Figure 4.4** Illustration of why ridges cause difficulties for hill climbing. The grid of sta (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of lo maxima that are not directly connected to each other. From each local maximum, all t available actions point downhill.

## Hill Climbing: Disadvantages

Ways Out

- Backtrack to some earlier node and try going in a different direction.
- Make a big jump to try to get in a new section.
- Moving in several directions at once.

## Performance of hill-climbing on 8-queens

- Randomly generated 8-queens starting states…
- 14% the time it solves the problem
- 86% of the time it get stuck at a local minimum
- However…
  - Takes only 4 steps on average when it succeeds
  - And 3 on average when it gets stuck
  - (for a state space with ~17 million states)

## Possible solution…sideways moves

- If no downhill (uphill) moves, allow sideways moves in hope that algorithm can escape
  - Need to place a limit on the possible number of sideways moves to avoid infinite loops

- For 8-queens
  - Now allow sideways moves with a limit of 100
  - Raises percentage of problem instances solved from 14 to 94%

  - However….
    - 21 steps for every successful solution
    - 64 for each failure

## Hill-climbing variations

- Stochastic hill-climbing
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- First-choice hill-climbing
  - stochastic hill climbing by generating successors randomly until a better one is found
  - Useful when there are a very large number of successors
- Random-restart hill-climbing
  - Tries to avoid getting stuck in local maxima.

## Hill-climbing with random restarts

- Different variations
  - For each restart: run until termination v. run for a fixed time
  - Run a fixed number of restarts or run indefinitely

- Analysis
  - Say each search has probability p of success
    - E.g., for 8-queens, p = 0.14 with no sideways moves

  - Expected number of restarts?
  - Expected number of steps taken?

## Local beam search

- Keep track of *k* states instead of one
  - Initially: *k* randomly selected states
  - Next: determine all  successors of *k* states
  - If any of successors is goal     finished
  - Else select *k* best  from successors and repeat.

- Major difference with random-restart search
  - Information is shared among *k* search threads.

- Can suffer from lack of diversity.
  - Stochastic beam search
    - choose k successors proportional to state quality.

## Search using Simulated Annealing

- Simulated Annealing = hill-climbing with non-deterministic search

- Basic ideas:
  - like hill-climbing identify the quality of the local improvements
  - instead of picking the best move, pick one randomly
  - say the change in objective function is $\delta$
  - if $\delta$ is positive, then move to that state
  - otherwise:
    - move to this state with probability proportional to $\delta$
    - thus: worse moves (very large negative $\delta$) are executed less often
  - however, there is always a chance of escaping from local maxima
  - over time, make it less likely to accept locally bad moves
  - (Can also make the size of the move random as well, i.e., allow "large" steps in state space)

## Physical Interpretation of Simulated Annealing

- Annealing = physical process of cooling a liquid or metal until particles achieve a certain frozen crystal state
    - simulated annealing:
        - free variables are like particles
        - seek "low energy" (high quality) configuration
        - get this by slowly reducing temperature T, which particles move around randomly

## Simulated annealing

```
function SIMULATED-ANNEALING( problem, schedule) return a solution state
    input: problem, a problem
        schedule, a mapping from time to temperature
    local variables: current, a node.
            next, a node.
            T, a "temperature" controlling the probability of downward steps

    current    MAKE-NODE(INITIAL-STATE[problem])
    for t    1 to ∞ do
        T    schedule[t]
        if T = 0 then return current
        next    a randomly selected successor of current
        ΔE    VALUE[next] - VALUE[current]
        if ΔE > 0 then current    next
        else current    next only with probability eΔE /T
```

# More Details on Simulated Annealing

- Lets say there are 3 moves available, with changes in the objective function of d1 = -0.1, d2 = 0.5, d3 = -5. (Let T = 1).
- pick a move randomly:
  - if d2 is picked, move there.
  - if d1 or d3 are picked, probability of move = exp(d/T)
  - move 1: prob1 = exp(-0.1) = 0.9,
    - i.e., 90% of the time we will accept this move
  - move 3: prob3 = exp(-5) = 0.05
    - i.e., 5% of the time we will accept this move

- T = "temperature" parameter
  - high T => probability of "locally bad" move is higher
  - low T => probability of "locally bad" move is lower
  - typically, T is decreased as the algorithm runs longer
    - i.e., there is a "temperature schedule"

## Simulated Annealing in Practice

- method proposed in 1983 by IBM researchers for solving VLSI layout problems (Kirkpatrick et al, *Science*, 220:671-680, 1983).
  - theoretically will always find the global optimum (the best solution)

- useful for some problems, but can be very slow
    - slowness comes about because T must be decreased very gradually to retain optimality
  - In practice how do we decide the rate at which to decrease T? (this is a practical problem with this method)

## Genetic algorithms

- Different approach to other search algorithms
- A successor state is generated by combining two parent states
  -

- A state is represented as a string over a finite alphabet (e.g. binary)
  - 8-queens
    - State = position of 8 queens each in a column

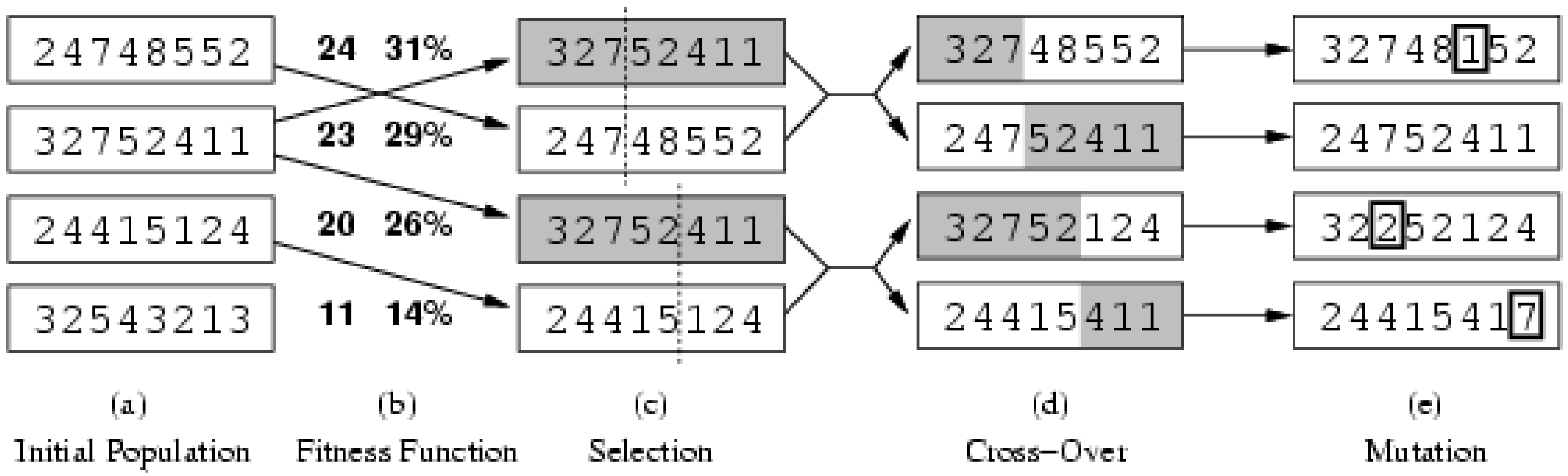
=> 8 x log(8) bits = 24 bits (for binary representation)

- Start with *k* randomly generated states (population)

- Evaluation function (fitness function).
- Higher values for better states.
  -
  - Opposite to heuristic function, e.g., # non-attacking pairs in 8-queens

- Produce the next generation of states by "simulated evolution"
  - Random selection
  - Crossover
  - Random mutation
  -

# Genetic algorithms



- 23/(24+23+20+11) = 29% etc
-

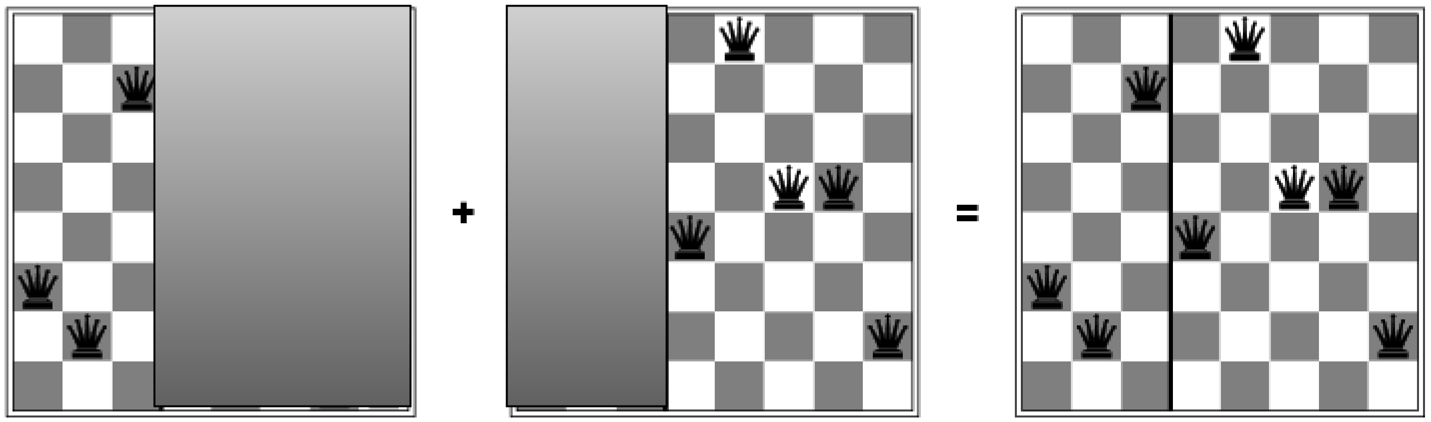
4 states for 8-queens problem

2 pairs of 2 states randomly selected based on fitness. Random crossover points selected

New states after crossover

Random mutation applied

## Genetic algorithms



Has the effect of “jumping” to a completely different new part of the search space (quite non-local)

# Genetic algorithm pseudocode

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

## Comments on genetic algorithms

- Positive points
  - Random exploration can find solutions that local search can’t
    - (via crossover primarily)
  - Appealing connection to human evolution
    - E.g., see related area of genetic programming

- Negative points
  - Large number of “tunable” parameters
    - Difficult to replicate performance from one problem to another

  - Lack of good empirical studies comparing to simpler methods

  - Useful on some (small?) set of problems but no convincing evidence that GAs are better than hill-climbing w/random restarts in general

## Adversarial Search and Game-Playing

## Typical assumptions

- Two agents whose actions alternate
- Utility values for each agent are the opposite of the other
  - creates the adversarial situation
- Fully observable environments
- In game theory terms:
  - "Deterministic, turn-taking, zero-sum games of perfect information"
- Can generalize to stochastic games, multiple players, non zero-sum, etc

## Search versus Games

- Search – no adversary
  - Solution is (heuristic) method for finding goal
  - Heuristics and CSP (Constraints Satisfaction Problem) techniques can find *optimal* solution
  - Evaluation function: estimate of cost from start to goal through given node
  - Examples: path planning, scheduling activities

- Games – adversary
  - Solution is strategy (strategy specifies move for every possible opponent reply).
  - Time limits force an *approximate* solution
  - Evaluation function: evaluate "goodness" of game position
  - Examples: chess, checkers, Othello, backgammon

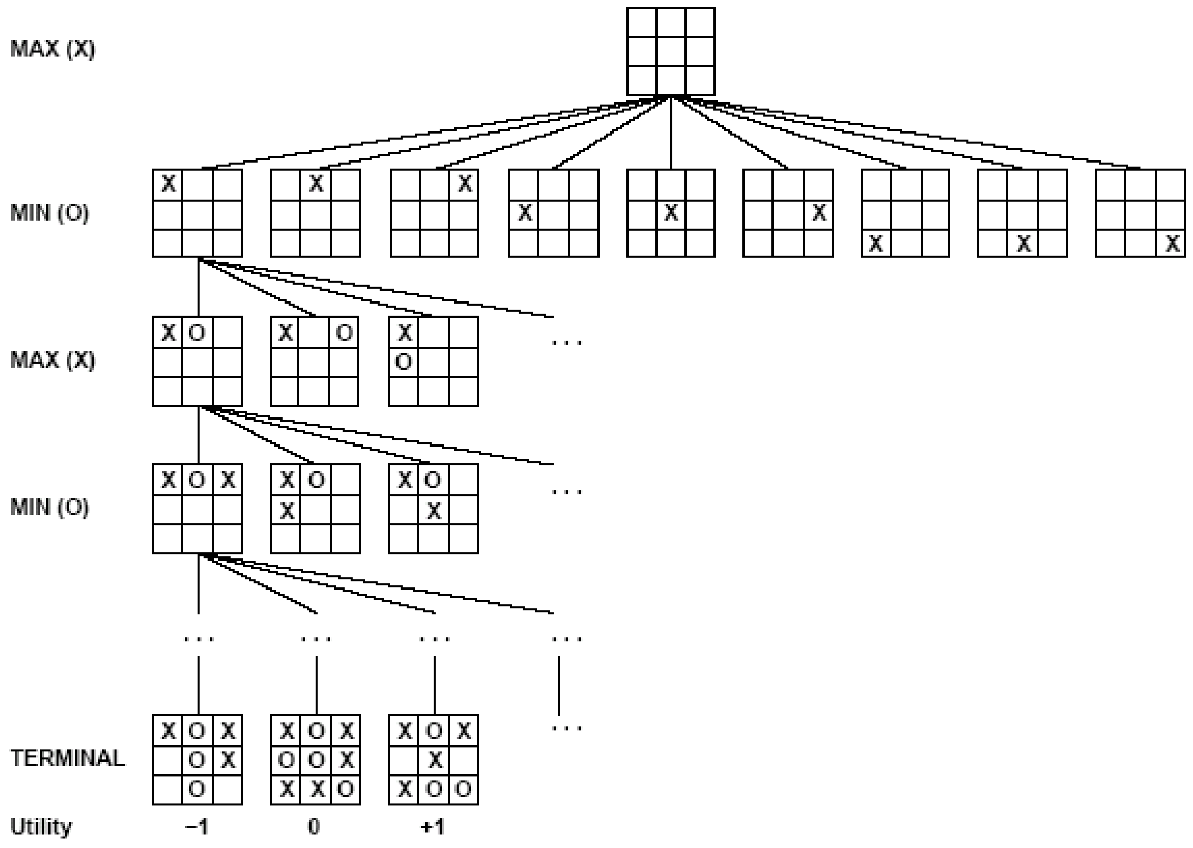
## Game Setup

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over
  - Winner gets award, loser gets penalty.
- Games as search:
  - Initial state: e.g. board configuration of chess
  - Successor function: list of (move,state) pairs specifying legal moves.
  - Terminal test: Is the game finished?
  - Utility function: Gives numerical value of terminal states. E.g. win (+1), lose (-1) and draw (0) in tic-tac-toe  or chess
- MAX uses  search tree to determine next move.

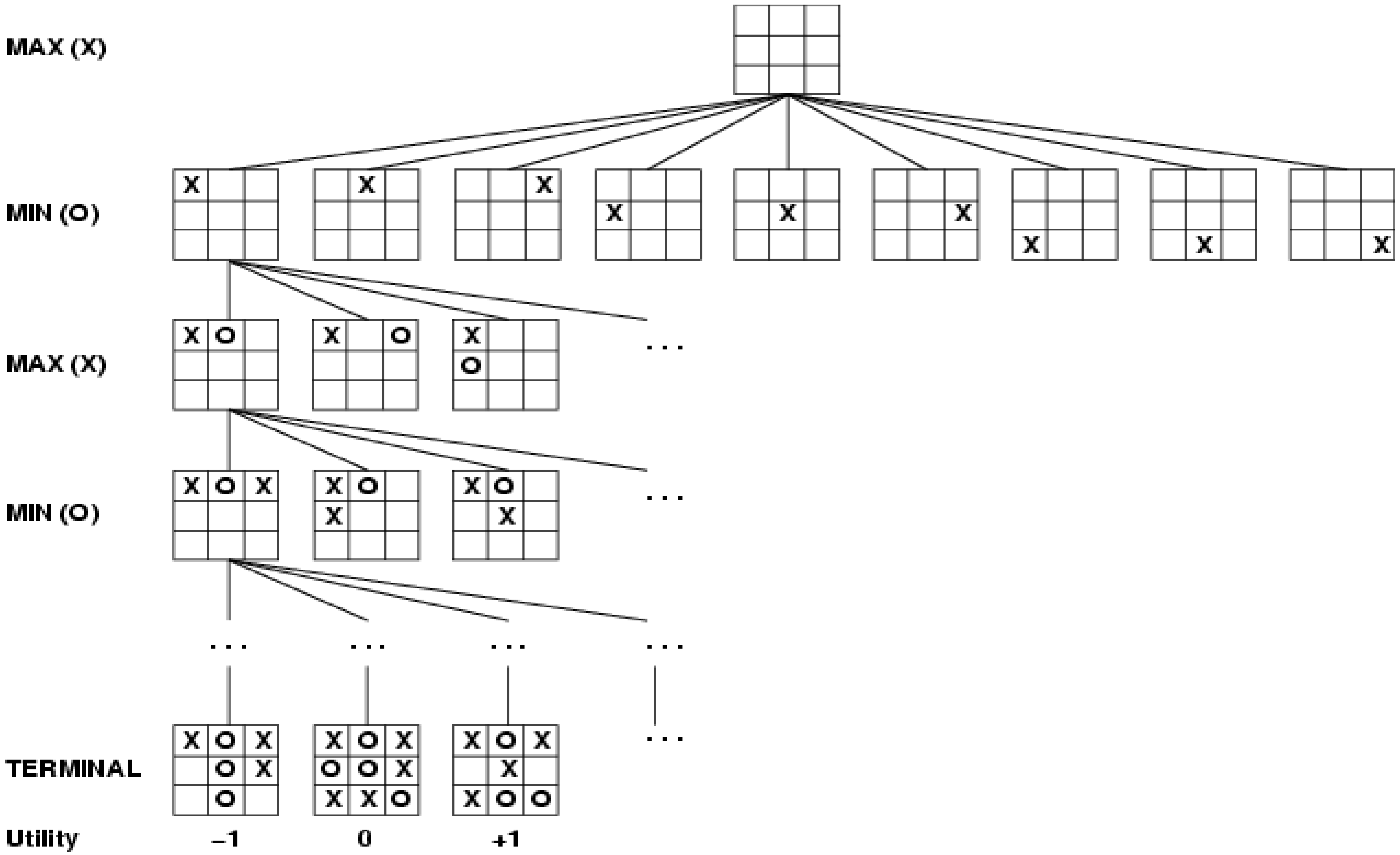
## Size of search trees

- b = branching factor
- d = number of moves by both players
- Search tree is O(bd)
- Chess
  - b ~ 35
  - D ~100
  - search tree is ~ 10 154 (!!)
  - completely impractical to search this
- Game-playing emphasizes being able to make optimal decisions in a finite amount of time
  - Somewhat realistic as a model of a real-world agent
  - Even if games themselves are artificial

## Partial Game Tree for Tic-Tac-Toe



## Game tree (2-player, deterministic, turns)



How do we search this tree to find the optimal move?

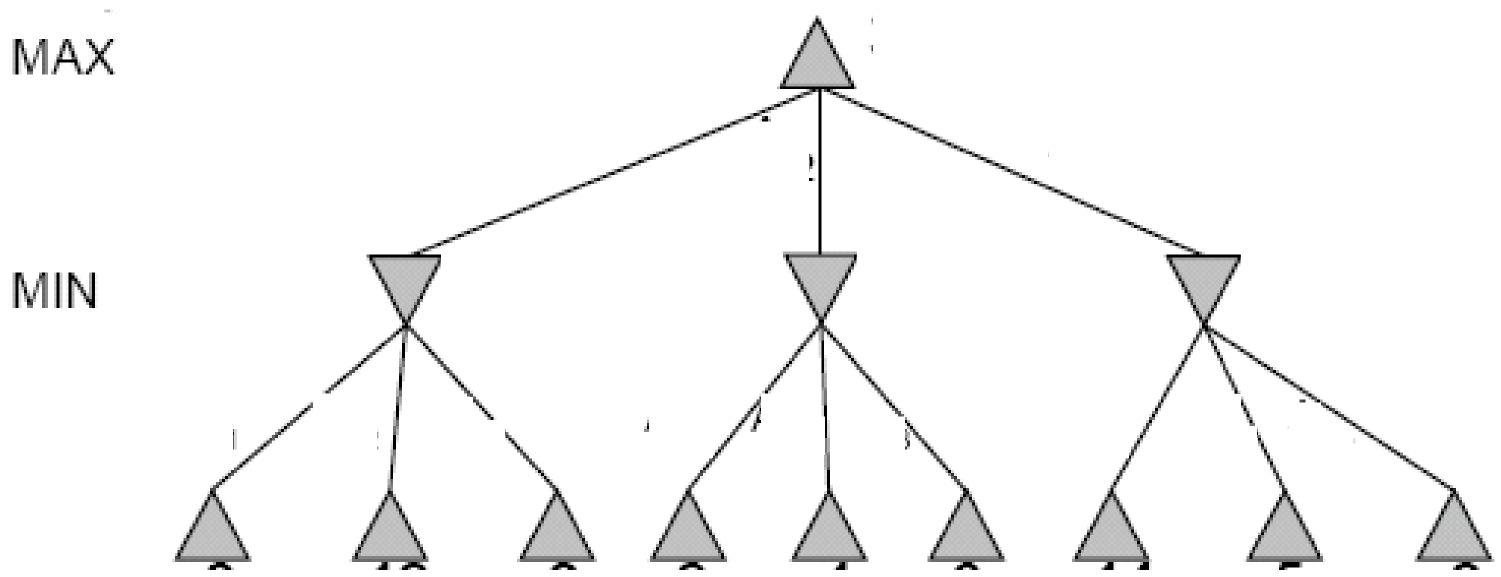
## Minimax strategy

- Find the optimal *strategy* for MAX assuming an infallible MIN opponent
  - Need to compute this all the down the tree
- Assumption: Both players play optimally!
- Given a game tree, the optimal strategy can be determined by using the minimax value of each node:

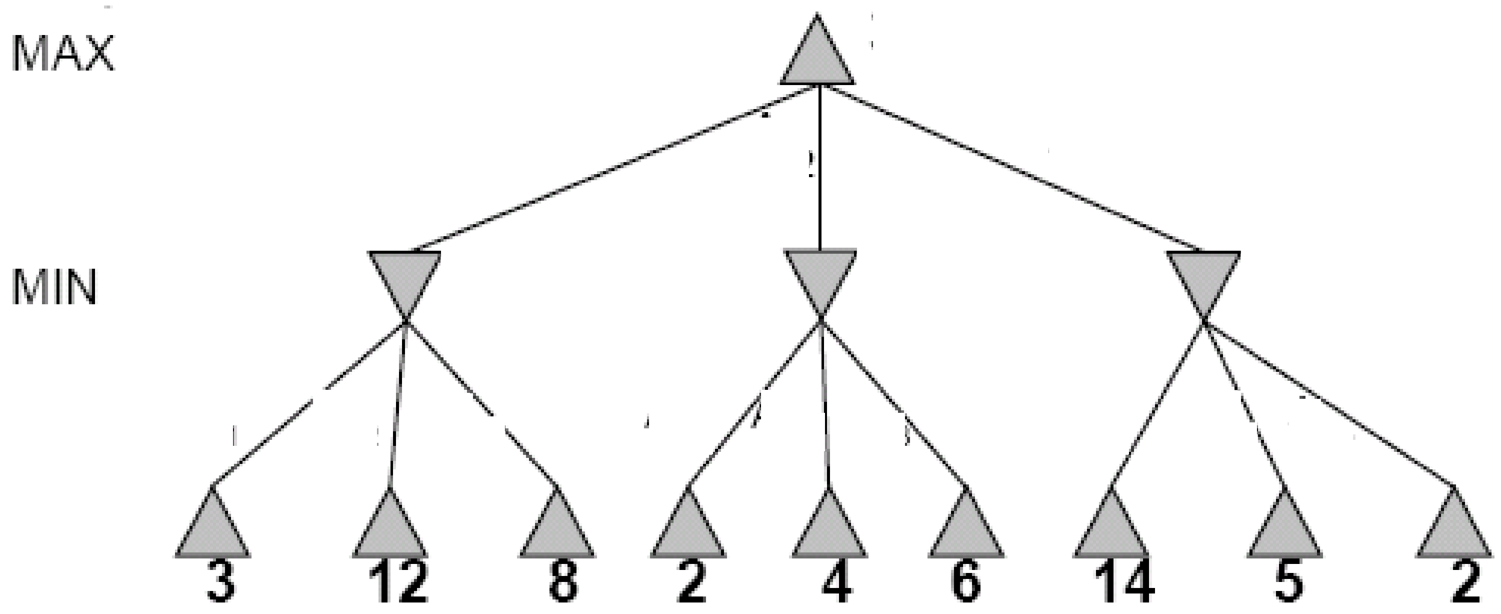
**MINIMAX-VALUE(*n*)=**

| | |
|---|---|
| **UTILITY(*n*)** | **If *n* is a terminal** |
| **maxs** ***successors(n)*** **MINIMAX-VALUE(*s*)** | **If *n* is a max node** |
| **mins** ***successors(n)*** **MINIMAX-VALUE(*s*)** | **If *n* is a min node** |

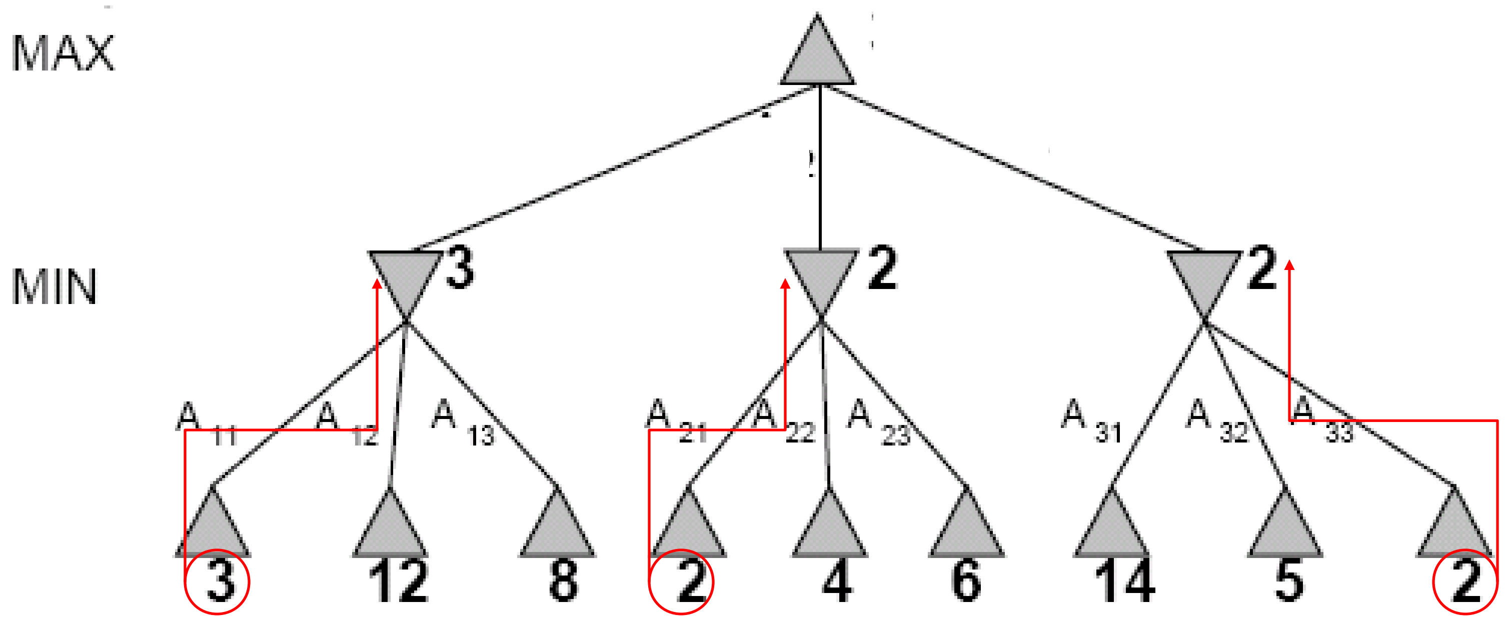
## Two-Ply Game Tree



## Two-Ply Game Tree

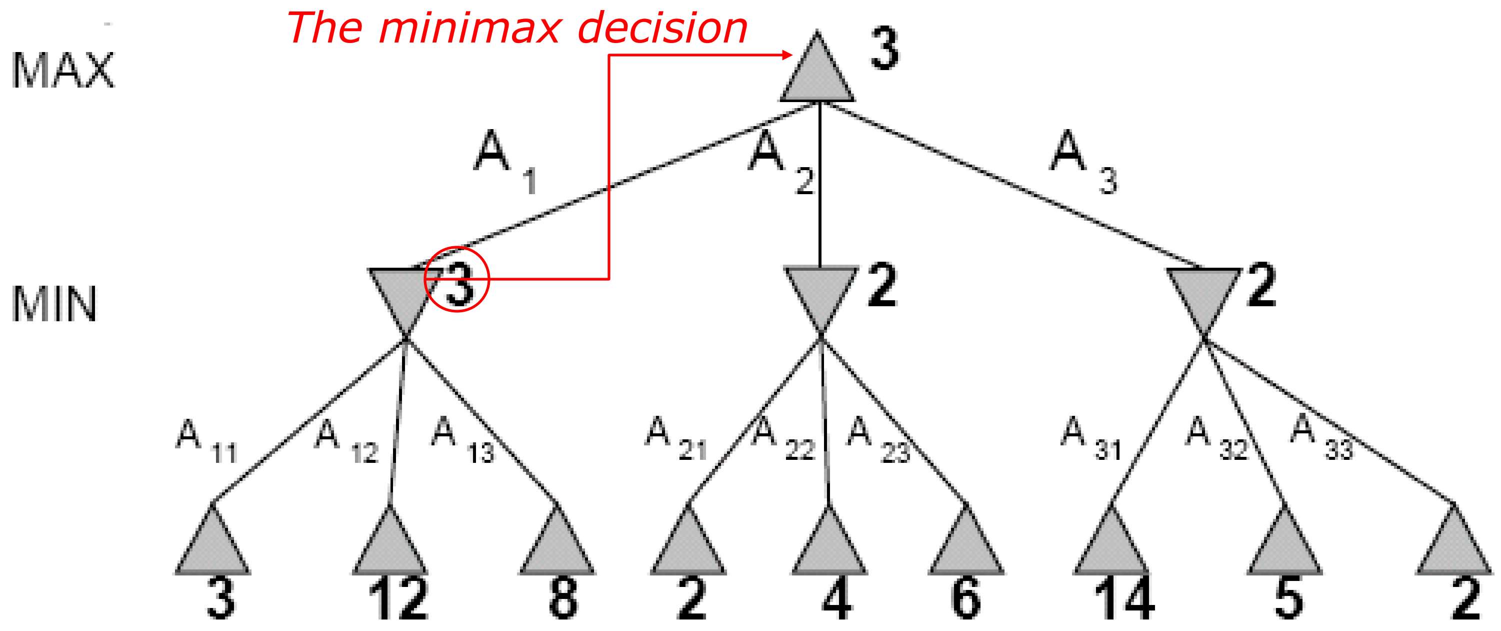


## Two-Ply Game Tree



## Two-Ply Game Tree

**Minimax maximizes the utility for the worst-case outcome for max**



## What if MIN does not play optimally?

- Definition of optimal play for MAX assumes MIN plays optimally:
  - maximizes worst-case outcome for MAX

- But if MIN does not play optimally, MAX will do even better

## Minimax Algorithm

- Complete depth-first exploration of the game tree
- Assumptions:
  - Max depth = d, b legal moves at each point
  - E.g., Chess: d ~ 100, b ~35

| **Criterion** | **Minimax** |
|---|---|
| Time | O(bm) |
| Space | O(bm) |

## Pseudocode for Minimax Algorithm

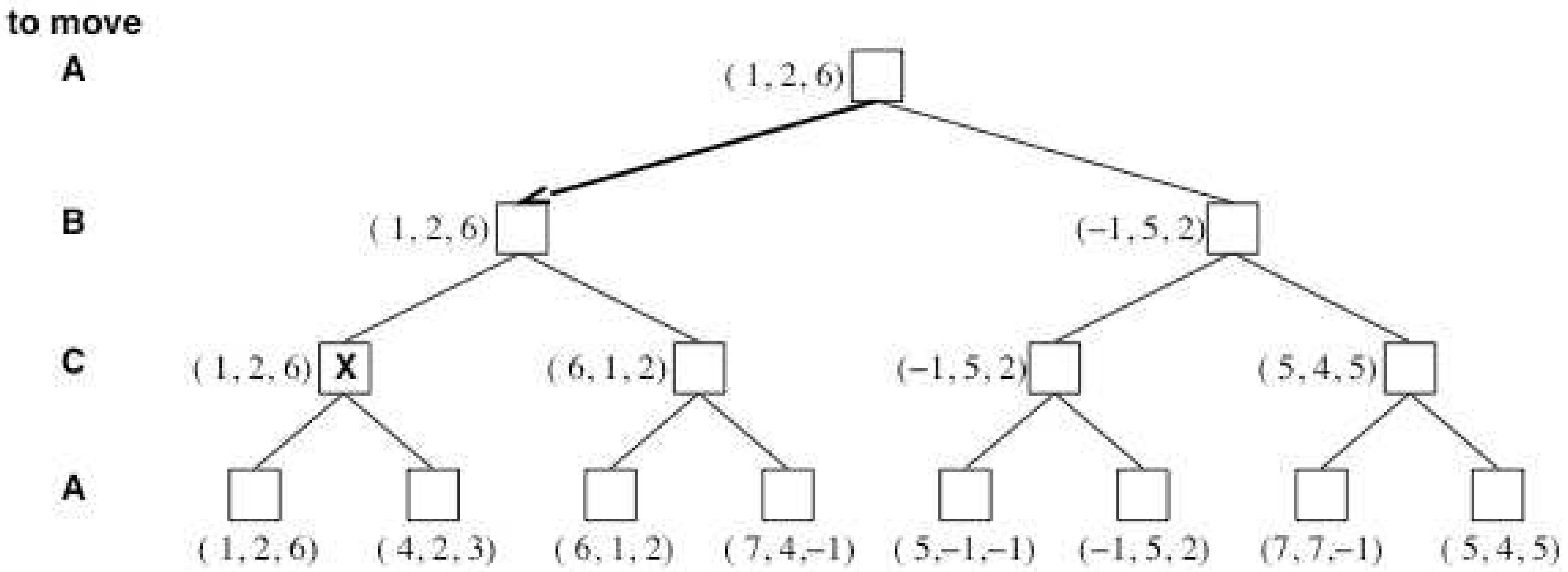
```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game
   $v$    MAX-VALUE(state)
   return the action in SUCCESSORS(state) with value $v$
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v$    $\infty$
   for a,s in SUCCESSORS(state) do
      $v$    MAX($v$,MIN-VALUE(s))
   return $v$
```

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v$    $\infty$
   for a,s in SUCCESSORS(state) do
      $v$    MIN($v$,MAX-VALUE(s))
   return $v$
```

## Multiplayer games

- Games allow more than two players
- Single minimax values become vectors



## Aspects of multiplayer games

- Previous slide (standard minimax analysis) assumes that each player operates to maximize only their own utility
- In practice, players make alliances
  - E.g, C strong, A and B both weak
  - May be best for A and B to attack C rather than each other
- If game is not zero-sum (i.e., utility(A) = - utility(B) then alliances can be useful even with 2 players
  - e.g., both cooperate to maximum the sum of the utilities

## Practical problem with minimax search

- Number of game states is exponential in the number of moves.
  - Solution: Do not examine every node

=> pruning

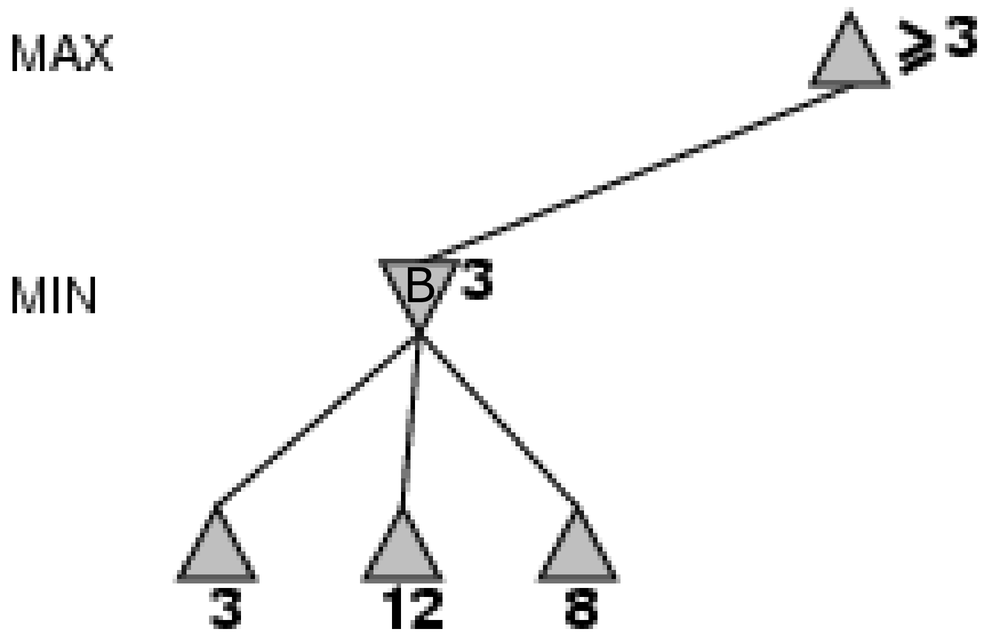
    - Remove branches that do not influence final decision

- Revisit example …

## Alpha-beta pruning

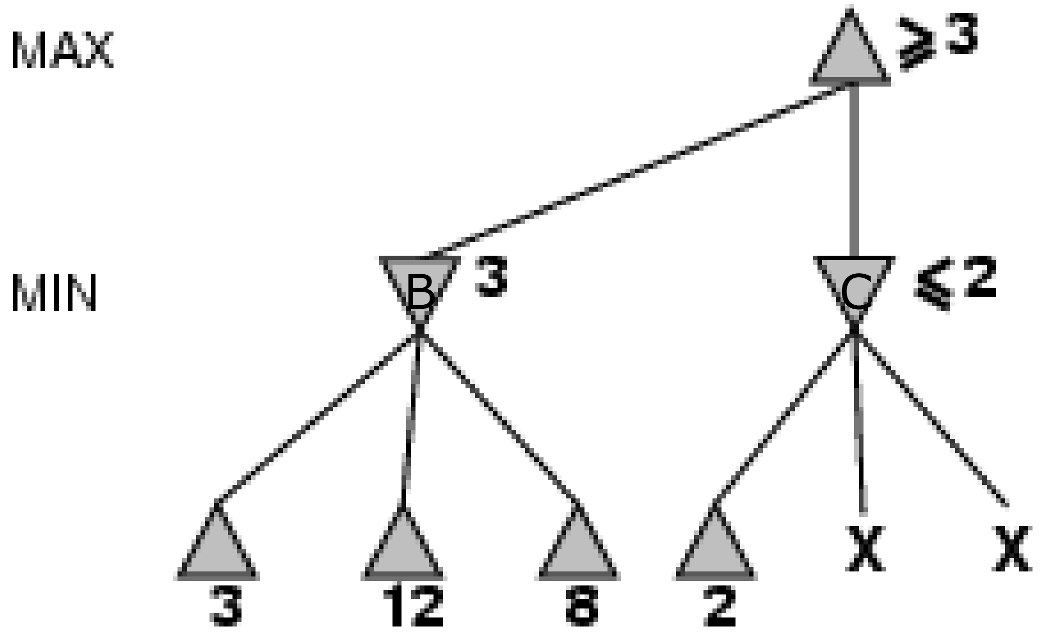
- It is possible to compute the correct minimax decision without looking at every node in the game tree.
- Alpha-beta pruning allows to eliminate large parts of the tree from consideration, without influencing the final decision.

## Alpha-beta pruning



- The value of B is exactly 3.
- It can be inferred that the value at the root is *at least* 3, because MAX has a choice worth 3.

## Alpha-beta pruning

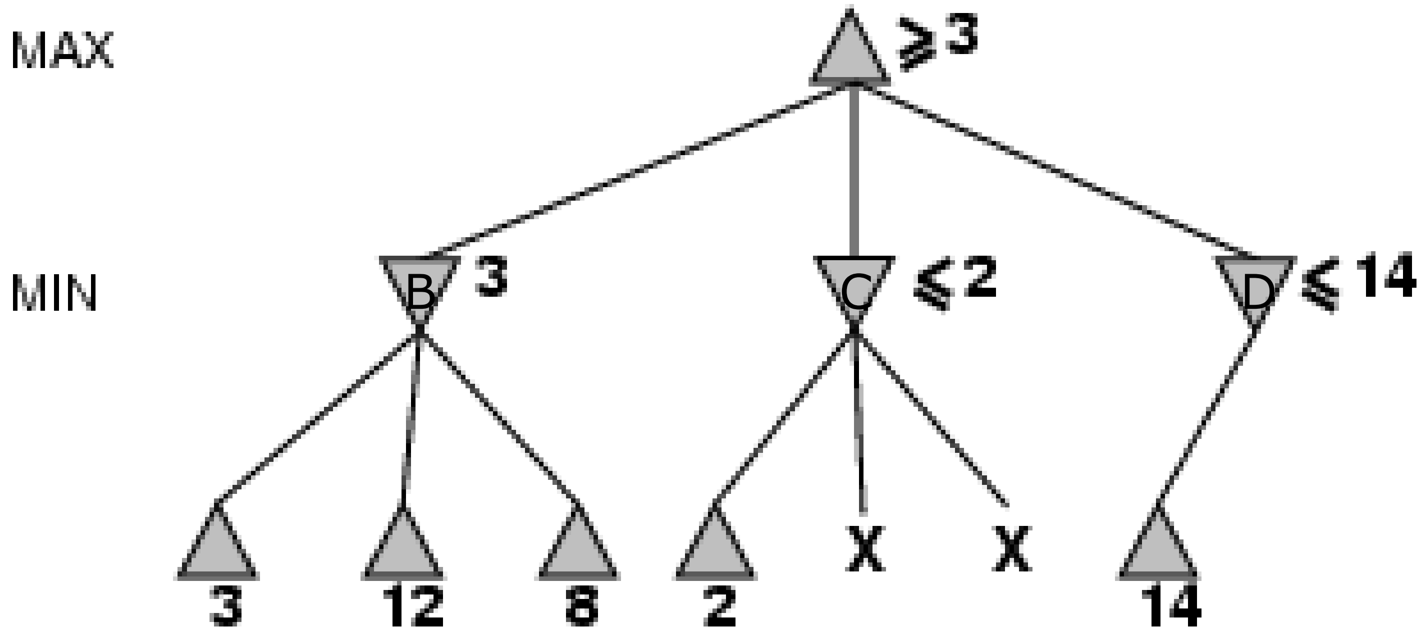


choose *C*.

- Therefore, there is no point in looking at the other successors of *C*.

21

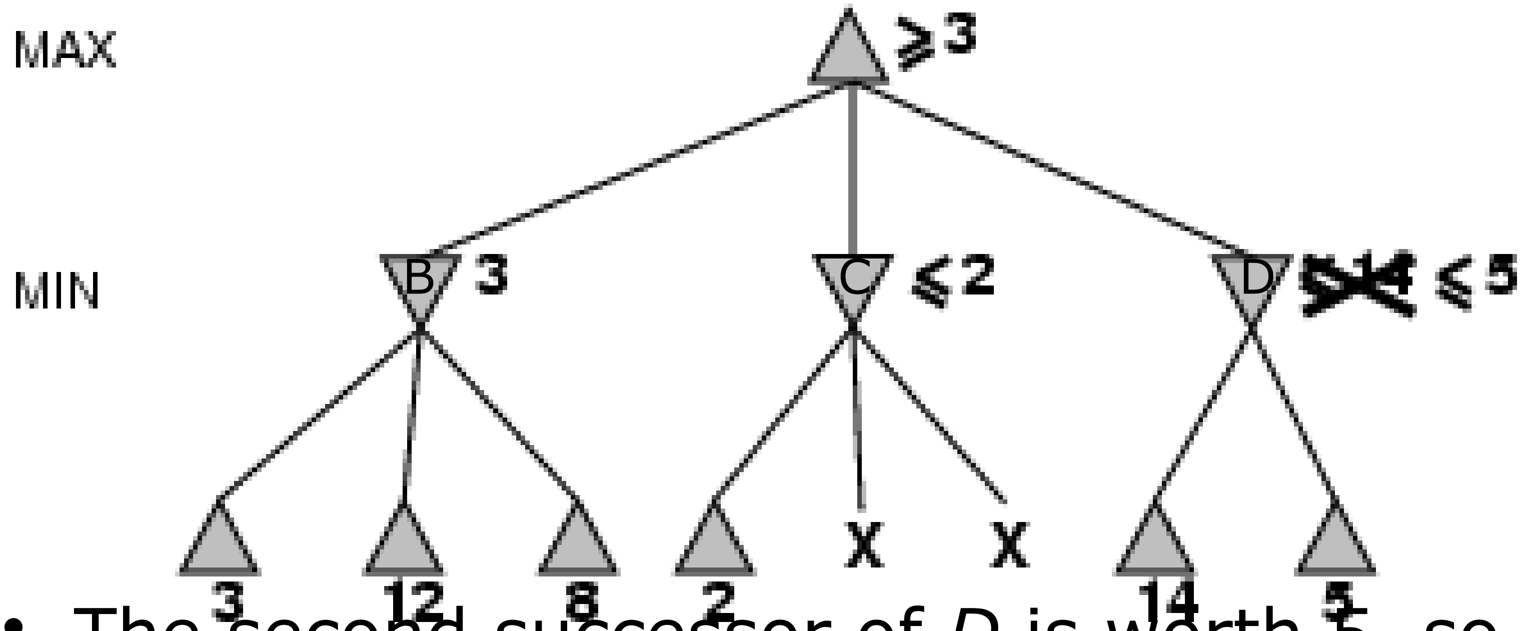
## Alpha-beta pruning



alternative (i.e., 3), so *D*'s other successors are explored.

22

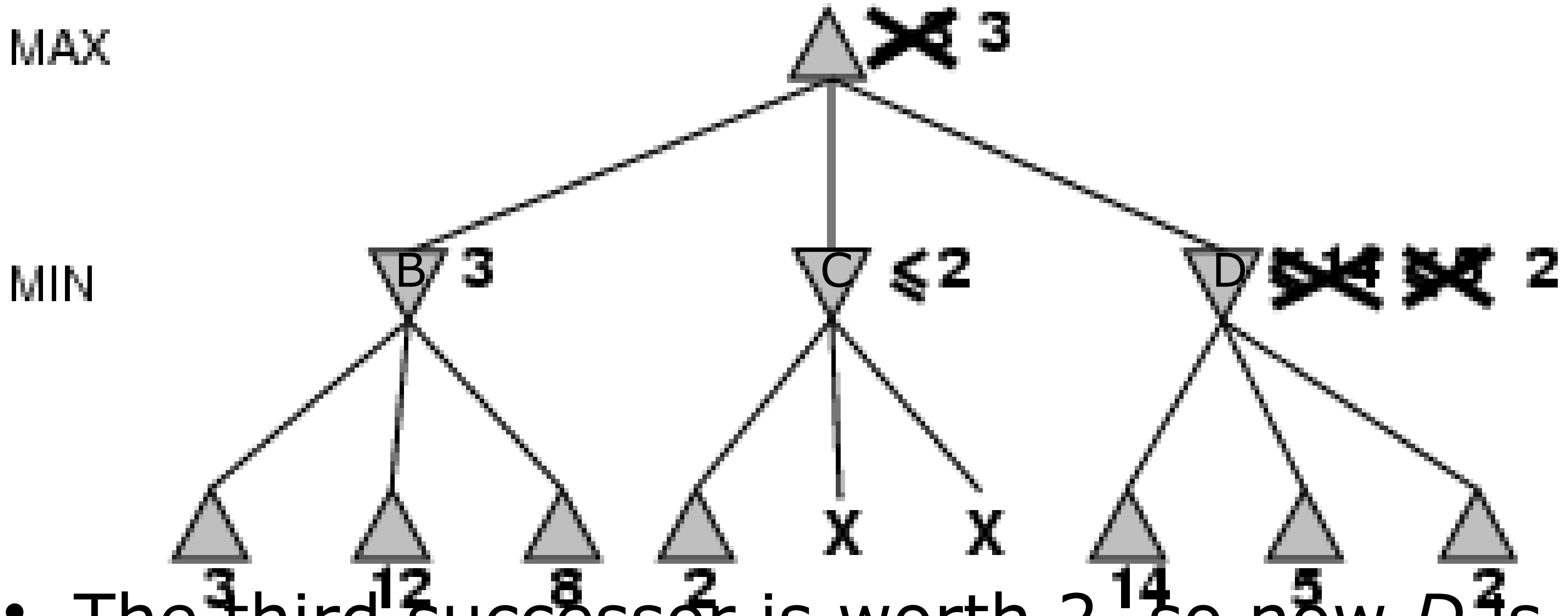
## Alpha-beta pruning



- The second successor of *D* is worth 5, so the exploration continues.

23

## Alpha-beta pruning



- The third successor is worth 2, so now *D* is worth exactly 2.
- MAX’s decision at the root is to move to *B*, giving a value of 3

24

## Alpha-beta pruning

- Alpha-beta pruning gets its name from two parameters.
  - They describe bounds on the values that appear anywhere along the path under consideration:
    - α = the value of the best (i.e., highest value) choice found so far along the path for MAX
    - β = the value of the best (i.e., lowest value) choice found so far along the path for MIN

25

## Alpha-beta pruning

- Alpha-beta search updates the values of *α* and *β* as it goes along.
- It prunes the remaining branches at a node (i.e., terminates the recursive call)
  - as soon as the value of the current node is known to be worse than the current *α* or *β* value for MAX or MIN, respectively.

26

## The alpha-beta search algorithm

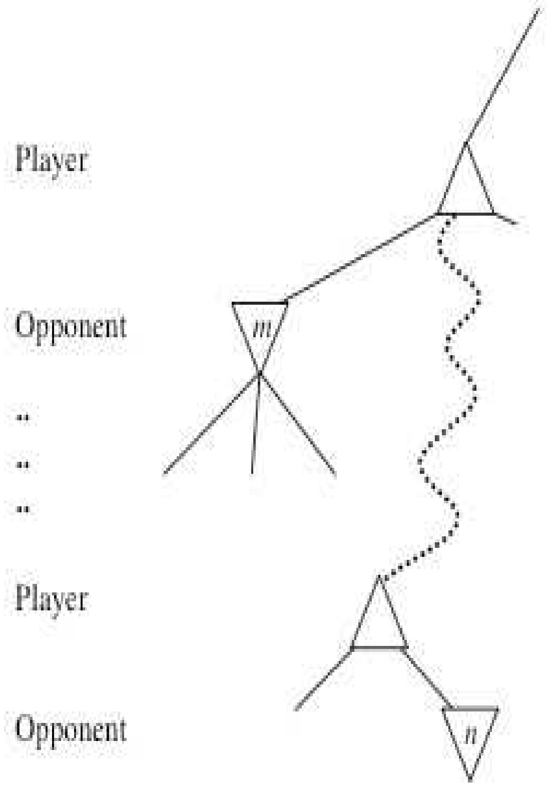
```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
  $v \leftarrow$ MAX-VALUE(state, $-\infty$, $+\infty$)
  return the action in SUCCESSORS(state) with value $v$
```

```
function MAX-VALUE(state, $\alpha$, $\beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for a,s in SUCCESSORS(state) do
    $v \leftarrow$ MAX($v$,MIN-VALUE(s, $\alpha$, $\beta$))
    if $v \geq \beta$ then return $v$
    $\alpha \leftarrow$ MAX($\alpha$, $v$)
  return $v$
```

```
function MIN-VALUE(state, $\alpha$, $\beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow +\infty$
  for a,s in SUCCESSORS(state) do
    $v \leftarrow$ MIN($v$,MAX-VALUE(s, $\alpha$, $\beta$))
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow$ MIN($\beta$, $v$)
  return $v$
```

## General alpha-beta pruning

- Consider a node *n* somewhere in the tree
- If player has a better choice at
  - Parent node of n
  - Or any choice point further up
- *n* will **never** be reached in actual play.
- Hence when enough is known about *n*, it can be pruned.



## Alpha-beta Algorithm

- Depth first search – only considers nodes along a single path at any time

$\alpha$ = highest-value choice we have found at any choice point along the path for MAX

$\beta$ = lowest-value choice we have found at any choice point along the path for MIN

- update values of $\alpha$ and $\beta$ during search and prunes remaining branches as soon as the value is known to be worse than the current $\alpha$ or $\beta$ value for MAX or MIN

## The Algorithm

- Visit the nodes in a depth-first manner
- Maintain bounds on nodes.
- A bound may change if one of its children obtains a unique value.
- A bound becomes a unique value when all its children have been checked or pruned.
- When a bound changes into a tighter bound or a unique value, it may become inconsistent with its parent.
- When an inconsistency occurs, prune the sub-tree by cutting the edge between the inconsistent bounds/values.

→ This is like propagating changes bottom-up in the tree.

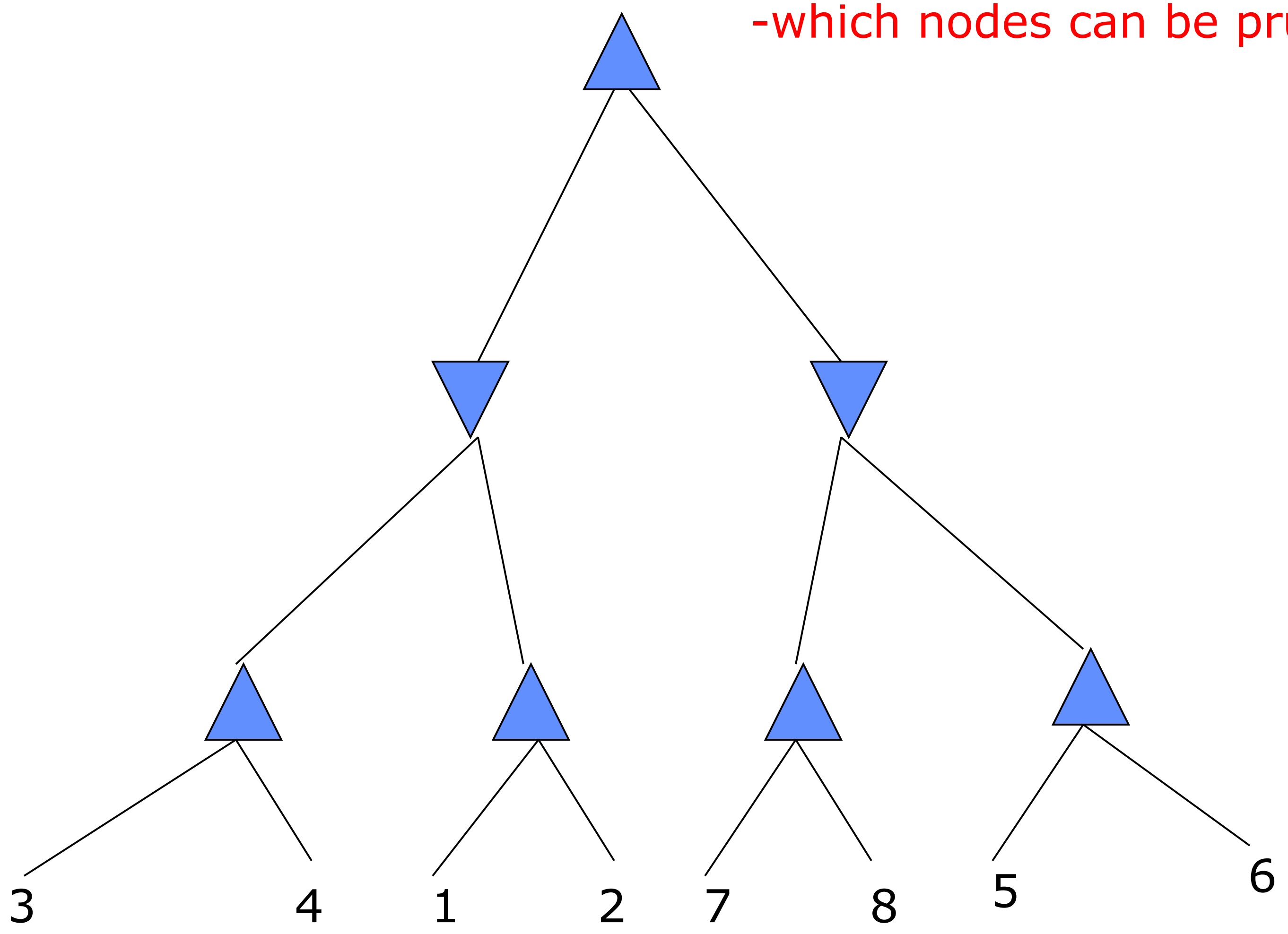
## Effectiveness of Alpha-Beta Search

- Worst-Case
  - branches are ordered so that no pruning takes place. In this case alpha-beta gives no improvement over exhaustive search
- Best-Case
  - each player’s best move is the left-most alternative (i.e., evaluated first)
  - in practice, performance is closer to best rather than worst-case
- In practice often get O(b(d/2)) rather than O(bd)
  - this is the same as having a branching factor of sqrt(b),
    - since (sqrt(b))d = b(d/2)
    - i.e., we have effectively gone from b to square root of b
  - e.g., in chess go from b ~ 35 to b ~ 6
    - this permits much deeper search in the same amount of time

## Final Comments about Alpha-Beta Pruning

- Pruning does not affect final results
- Entire subtrees can be pruned.
- Good move *ordering* improves effectiveness of pruning
- Repeated states are again possible.
  - Store them in memory = transposition table

## Example



## Practical Implementation

How do we make these ideas practical in real game trees?

Standard approach:

- cutoff test: (where do we stop descending the tree)
  - depth limit
  - better: iterative deepening
  - cutoff only when no big changes are expected to occur next (quiescence search).

- evaluation function
  - When the search is cut off, we evaluate the current state by estimating its utility. This estimate if captured by the

  evaluation function.

## Static (Heuristic) Evaluation Functions

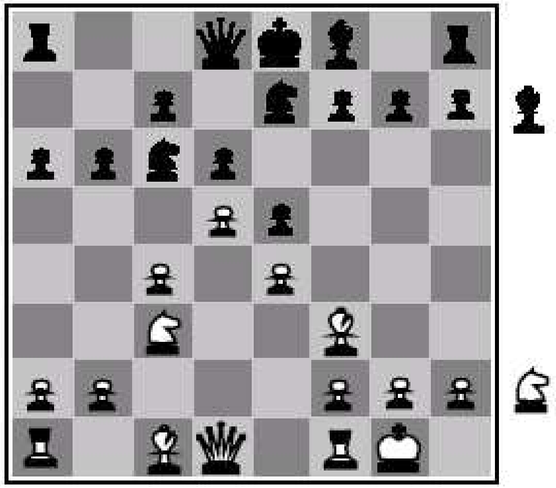
- An Evaluation Function:
  - estimates how good the current board configuration is for a player.
  - Typically, one figures how good it is for the player, and how good it is for the opponent, and subtracts the opponents score from the players
  - Othello: Number of white pieces - Number of black pieces
  - Chess: Value of all white pieces - Value of all black pieces

- Typical values from -infinity (loss) to +infinity (win) or [-1, +1].

- If the board evaluation is X for a player, it’s -X for the opponent

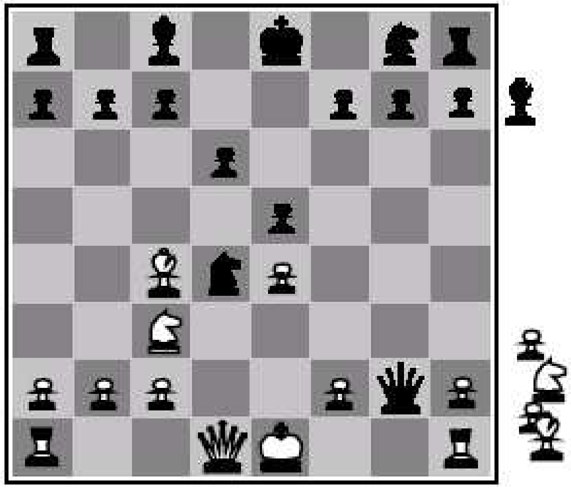
- Example:
  - Evaluating chess boards,
  - Checkers
  - Tic-tac-toe

## Evaluation functions



Black to move

White slightly better



White to move

Black winning

For chess, typically *linear* weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens), etc.

## Iterative (Progressive) Deepening

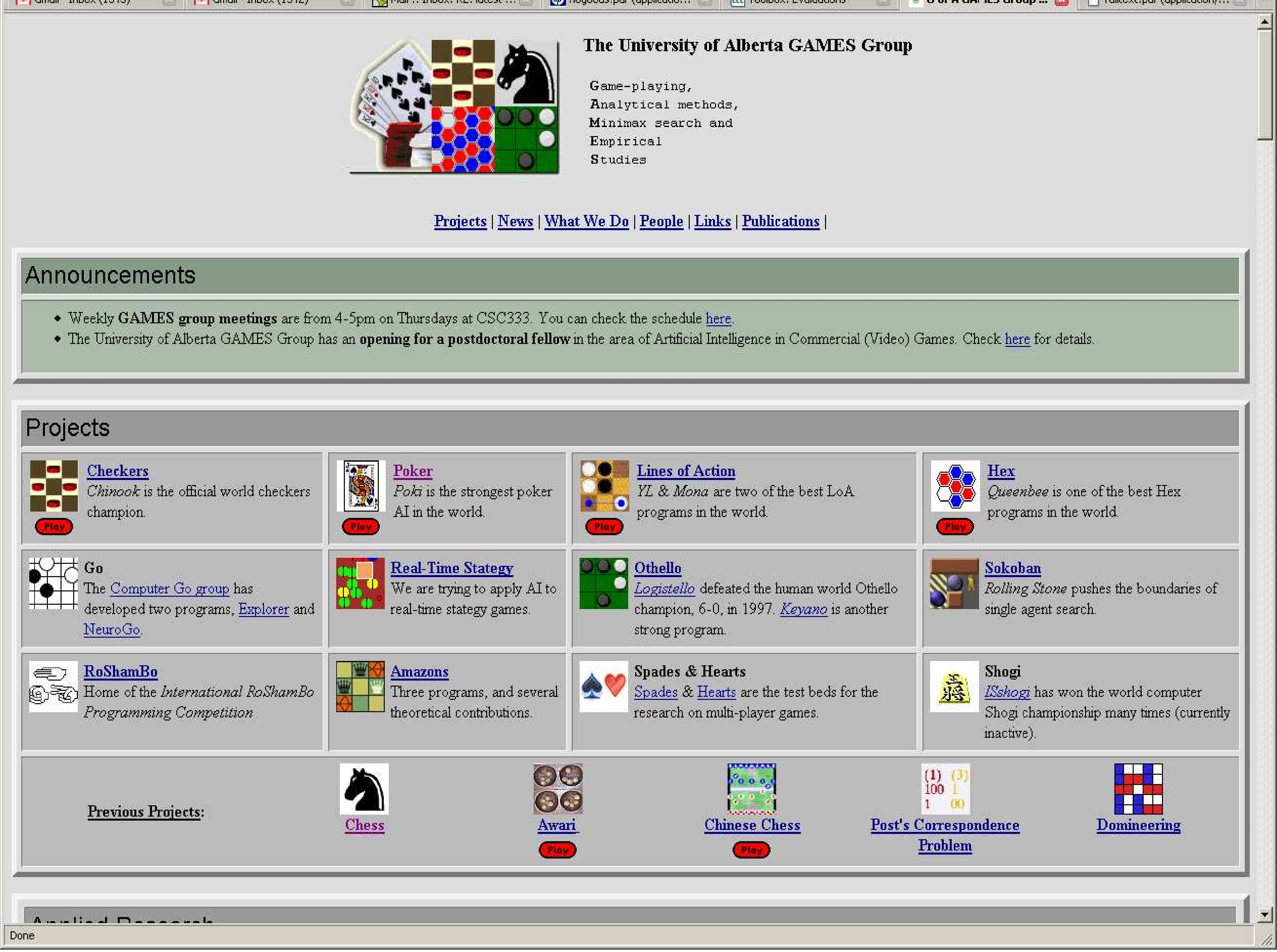
- In real games, there is usually a time limit T on making a move
- How do we take this into account?
  - using alpha-beta we cannot use “partial” results with any confidence unless the full breadth of the tree has been searched
  - So, we could be conservative and set a conservative depth-limit which guarantees that we will find a move in time < T
    - disadvantage is that we may finish early, could do more search
- In practice, iterative deepening search (IDS) is used
  - IDS runs depth-first search with an increasing depth-limit
  - when the clock runs out we use the solution found at the previous depth limit

## Heuristics and Game Tree Search

- The Horizon Effect
  - sometimes there's a major "effect" (such as a piece being captured) which is just "below" the depth to which the tree has been expanded
  - the computer cannot see that this major event could happen
  - it has a "limited horizon"
  - there are heuristics to try to follow certain branches more deeply to detect to such important events
  - this helps to avoid catastrophic losses due to "short-sightedness"
- Heuristics for Tree Exploration
  - it may be better to explore some branches more deeply in the allotted time
  - various heuristics exist to identify "promising" branches

## The State of Play

- Checkers:
  - Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994.
- Chess:
  - Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997.
- Othello:
  - human champions refuse to compete against computers: they are too good.
- Go:
  - human champions refuse to compete against computers: they are too bad
  - b > 300 (!)
- See (e.g.) http://www.cs.ualberta.ca/~games/ for more information



**The University of Alberta GAMES Group**

```
Game-playing,
Analytical methods,
Minimax search and
Empirical
Studies
```

Projects | News | What We Do | People | Links | Publications |

## Announcements

- Weekly **GAMES group meetings** are from 4-5pm on Thursdays at CSC333. You can check the schedule here.
- The University of Alberta GAMES Group has an **opening for a postdoctoral fellow** in the area of Artificial Intelligence in Commercial (Video) Games. Check here for details.

## Projects

| | | | |
|---|---|---|---|
| Checkers<br>*Chinook* is the official world checkers champion.<br>Play | Poker<br>*Poki* is the strongest poker AI in the world.<br>Play | Lines of Action<br>*YL* & *Mona* are two of the best LoA programs in the world.<br>Play | Hex<br>*Queenbee* is one of the best Hex programs in the world.<br>Play |
| **Go**<br>The Computer Go group has developed two programs, Explorer and NeuroGo. | Real-Time Stategy<br>We are trying to apply AI to real-time stategy games. | Othello<br>*Logistello* defeated the human world Othello champion, 6-0, in 1997. *Keyano* is another strong program. | Sokoban<br>*Rolling Stone* pushes the boundaries of single agent search. |
| RoShamBo<br>Home of the *International RoShamBo Programming Competition* | Amazons<br>Three programs, and several theoretical contributions. | **Spades & Hearts**<br>Spades & Hearts are the test beds for the research on multi-player games. | **Shogi**<br>*ISshogi* has won the world computer Shogi championship many times (currently inactive). |

**Previous Projects:** Chess | Awari (Play) | Chinese Chess (Play) | Post's Correspondence Problem | Domineering

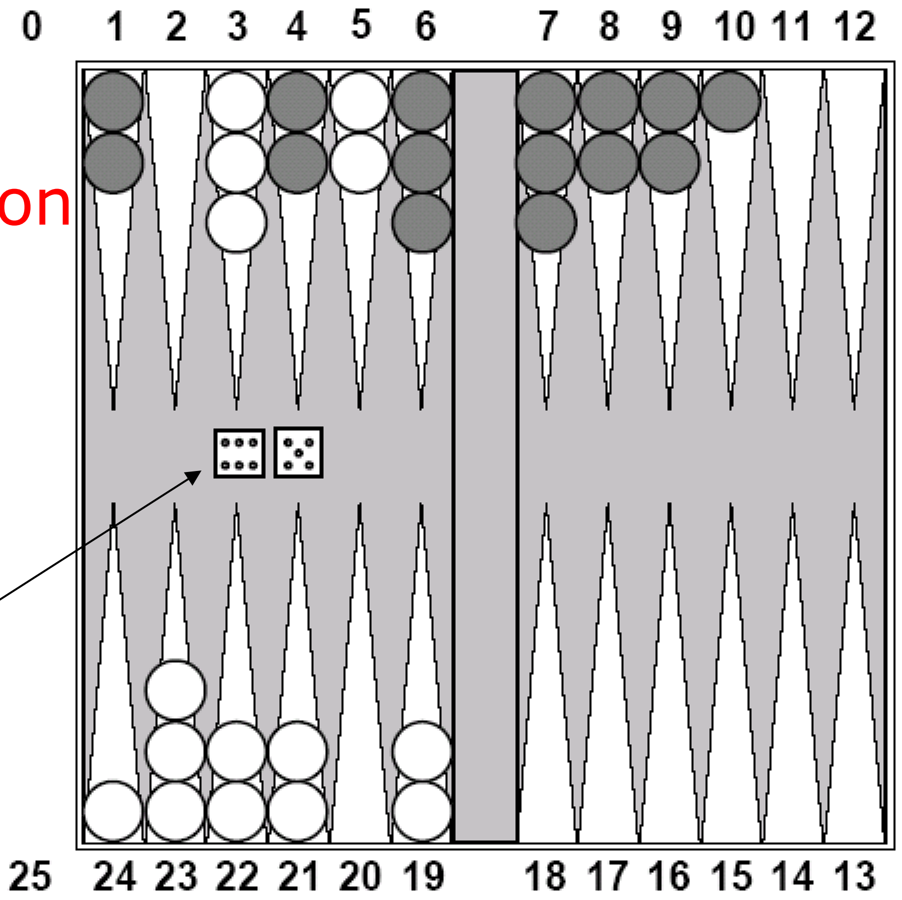
Done

## Deep Blue

- 1957: Herbert Simon
  - "within 10 years a computer will beat the world chess champion"
- 1997: Deep Blue beats Kasparov
- Parallel machine with 30 processors for "software" and 480 VLSI processors for "hardware search"
- Searched 126 million nodes per second on average
  - Generated up to 30 billion positions per move
  - Reached depth 14 routinely
- Uses iterative-deepening alpha-beta search with transpositioning
  - Can explore beyond depth-limit for interesting moves

## Chance Games.

Backgammon

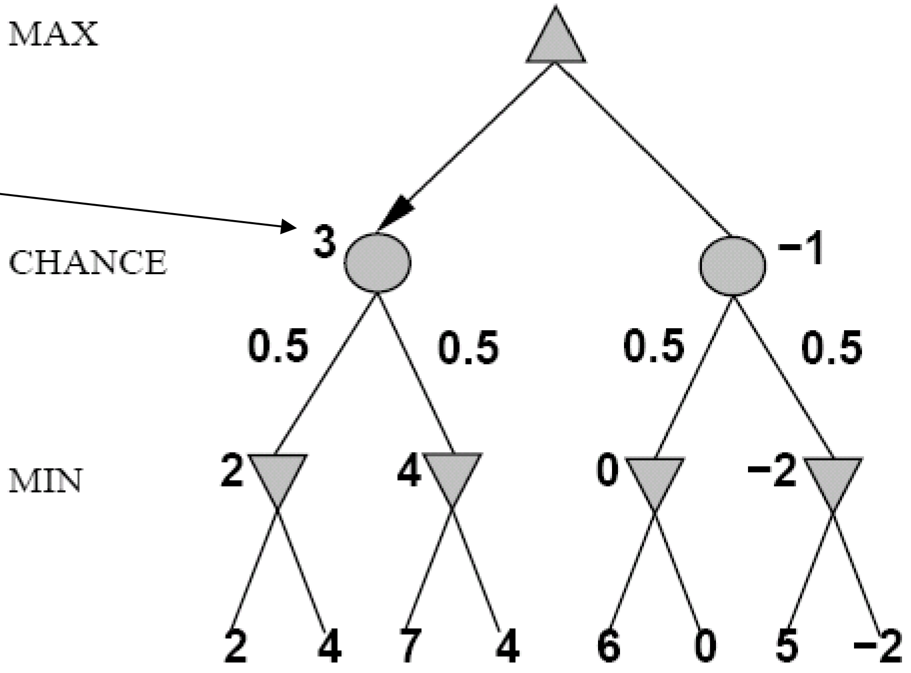


your element of chance

## Expected Minimax

$$v = \sum_{chance\ nodes} P(n) \times Minimax(n)$$

$$3 = 0.5 \times 4 + 0.5 \times 2$$



Interleave chance nodes with min/max nodes

Again, the tree is constructed bottom-up

## Summary

- Game playing can be effectively modeled as a search problem
- Game trees represent alternate computer/opponent moves
- Evaluation functions estimate the quality of a given board configuration for the Max player.
- Minimax is a procedure which chooses moves by assuming that the opponent will always choose the move which is best for them
- Alpha-Beta is a procedure which can prune large parts of the search tree and allow search to go deeper
- For many well-known games, computer algorithms based on heuristic search match or out-perform human world experts.
- Reading:R&N Chapter 6.