FIRST THREE WEEKS REVIEW

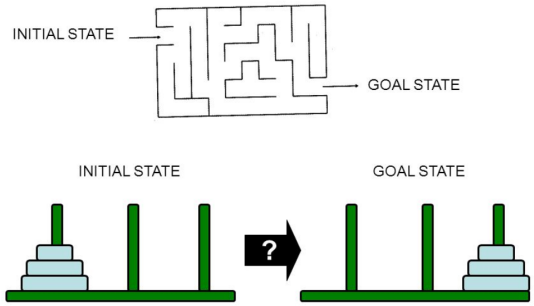
PROBLEMS: TYPES AND ATTRIBUTES.

Summary

PROBLEM TYPES: WELL-DEFINED VS ILL-DEFINED

Well-defined problem (well-structured): any problem in which the initial state or starting position, the allowable operations, and the goal state are clearly specified, and a unique solution can be shown to exist.
A problem that lacks one or more of these specified properties is an ill-defined problem, and most problems that we encounter in our everyday life tend to be from this category.

Well-defined Problem



Towers of Hanoi: Well-defined

Running A Business: ill-defined

An option for you to know: A problem is said to be well-posed if the problem is stable, which is determined by whether it meets the three Hadamard criteria, which tests whether or not the problem has:

- **A solution**: *s* exists for all *d* (for every *d* relevant to the problem).
- **A unique solution**: *s* is unique for all d; for every data point *d* there is at most one value of *s*.
- **A stable solution**: *s* depends continuously on *d* (a tiny change in *d* will lead to a tiny change in *s*; and a large change in d will lead to a proportionally larger change in *s*).

The Hadamard criteria tells us how well a problem lends itself to mathematical analysis.

PROBLEM ATTRIBUTES

Observable Environment: We assume that the environment is is observable, so that the agent always knows the current state– initial state. There are situations where our knowledge of the current state is partial, and an estimate at best. State estimation is a big problem in control theory. Quite possibly, unobservable: If the agent has no sensors at all then the environment is unobservable.

Discrete / continuous: If in an environment there are a finite number of percepts/inputs and actions that can be performed within it, then such an environment is called a discrete environment.. A chess game comes under discrete environment as there is a finite number of moves that can be performed. A drone motion control is an example of a continuous environment.

Deterministic/Stochastic : We have assumed that the environment is deterministic. That is, each action has exactly one outcome, and that the future is perfectly predictable. That is if an agent's current state and selected action can completely determine the next state of the environment, then such environment is called a deterministic environment.

Static/Dynamic: If while the agent is contemplating an action the environment can change itself, then such environment is called dynamic environment; Otherwise, it is called a static environment. Static environments are easier to deal with compared to dynamic environments, an agent in this case does not need to continue observing the environment, while selecting his next action. Driving a car in traffic is an example of a dynamic environment, whereas Crossword puzzles are an example of a static environment.
Static vs dynamic: no attention to changes in the environment. Planning a trip to Toronto, no consideration to traffic conditions/accidents/weather, ect.

Episodic vs Sequential: In an episodic environment, there is a sequence of one–shot actions, and only the current percept/input is required for the action. However, in Sequential environment, an agent requires memory of past actions to determine the next best actions.

PROBLEM SOLVING STRATEGIES

Summary

PROBLEM SOLVING PROCESS

Intelligent agents can solve problems by searching a state–space

State–space Model:

- the agent's model of the world
- usually a set of discrete states, for example, in driving, the states in the model could be towns/cities/stops

Goal State(s)

- a goal is defined as a desirable state for an agent
- there could be many states that satisfy the goal, for example, drive to a town with a ski–resort
- or just one state that satisfies the goal • e.g., drive to Toronto
- Goal formulation, is the first step in the problem solving process, based on the current situation and the agent's performance measure,
- Goal is a set of states. The agent's task is to find out which sequence of actions will get it to a goal state
- Problem formulation is the process of deciding what sorts of actions and states to consider, given a goal.

Operators(actions)

– operators are legal actions that the agent can take to move from one state to another

MEASURING PROBLEM-SOLVING PERFORMANCE

We evaluate a strategy's performance in four ways:

Completeness: Is the strategy guaranteed to find a solution when there is one?

Optimality: Does the strategy find the optimal solution?

Time Complexity: The upper bound on the time required to find a solution, as a function of the complexity of the problem.

Space Complexity: The upper bound on the storage space (memory) required at any point during the search, as a function of the complexity of the problem.

Performance measures are evaluated with respect to some problem difficulty parameters.

It is typical to use the size of the state space graph as a measure, |V | + |E|. where V is the set of vertices (nodes) of the graph and E is the set of edges (links).

Thus, complexity is expressed in terms of three quantities: b, the branching factor or maximum number of successors of any node; d, the depth of the shallowest goal node (i.e., the number of steps along the path from the root); and m, the maximum length of any path in the state space.

The aim is to search for a route, or sequence of transitions, through the state space graph from our initial state to a ***goal state***.
There could be more than one possible goal state.
We define a goal test to determine if a goal state has been achieved.
The solution can be represented as a sequence of edges (or transitions) on the state space graph.
Sometimes there may be more than one path to a goal state, and we may want to find the optimal (best possible) path.
We can define edge costs and path costs for measuring the cost of going along a particular path, e.g. the path cost may just equal the number of edges, or could be the sum of individual edge costs.
For most realistic problems, the state space graph will be too large for us to hold all of it explicitly in memory at any one time.

INFORMED VS UNINFORMED SEARCH

BASIS FOR COMPARISON	INFORMED SEARCH	UNIFORMED SEARCH
Basic	Uses knowledge to find the steps to the solution.	No use of knowledge
Efficiency	Highly efficient as consumes less time and cost.	Efficiency is mediatory
Cost	Low	Comparatively high
Performance	Finds solution more quickly	Speed is slower than informed search
Algorithms	Heuristic depth first and breadth-first search, and A* search	Depth-first search, breadth-first search and lowest cost first search

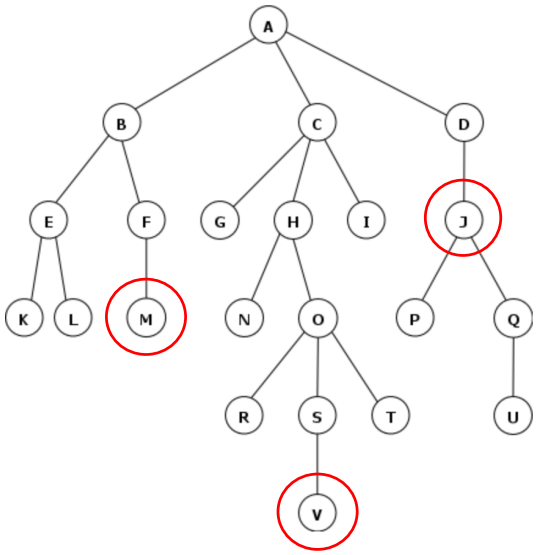
- ✓ Breadth-first
  - ✓ Uniform-cost
- ✓ Depth-first
  - ✓ Depth-limited
  - ✓ Iterative deepening

UNINFORMED SEARCH STRATEGIES

Summary

BREADTH-FIRST SEARCH

Strategy: expand the shallowest unexpanded node. Implementation: The fringe is a FIFO queue.
For the following search tree:



The queue evolves like this

A
B C D
C D E F
D E F G H I
E F G H I J
F G H I J K L
G H I J K L M
H I J K L M
I J K L M N O
J K L M N O
K L M N O P Q
L M N O P Q
M N O P Q
N O P Q
O P Q
P Q R S T
Q R S T
R S T U
S T U
T U V
U V
V

The order of generation is ABCDEFGHIJKLMNOPQRSTUV and the order of expansion is the same.
If the goal nodes were M, V, and J, Breadth–First search would find J, the shallowest.

Optimal (i.e., admissible) if all operators have the same cost. Otherwise, not optimal but finds solution with shortest path length.

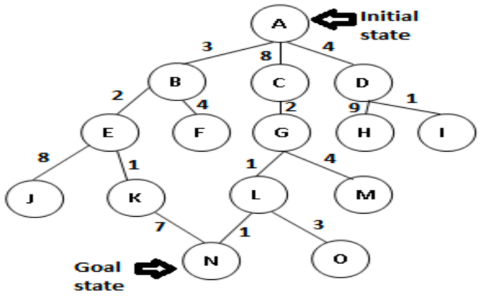
Exponential time and space complexity, $O(b^d)$, where d is the depth of the solution and b is the branching factor (i.e., number of children) at each node

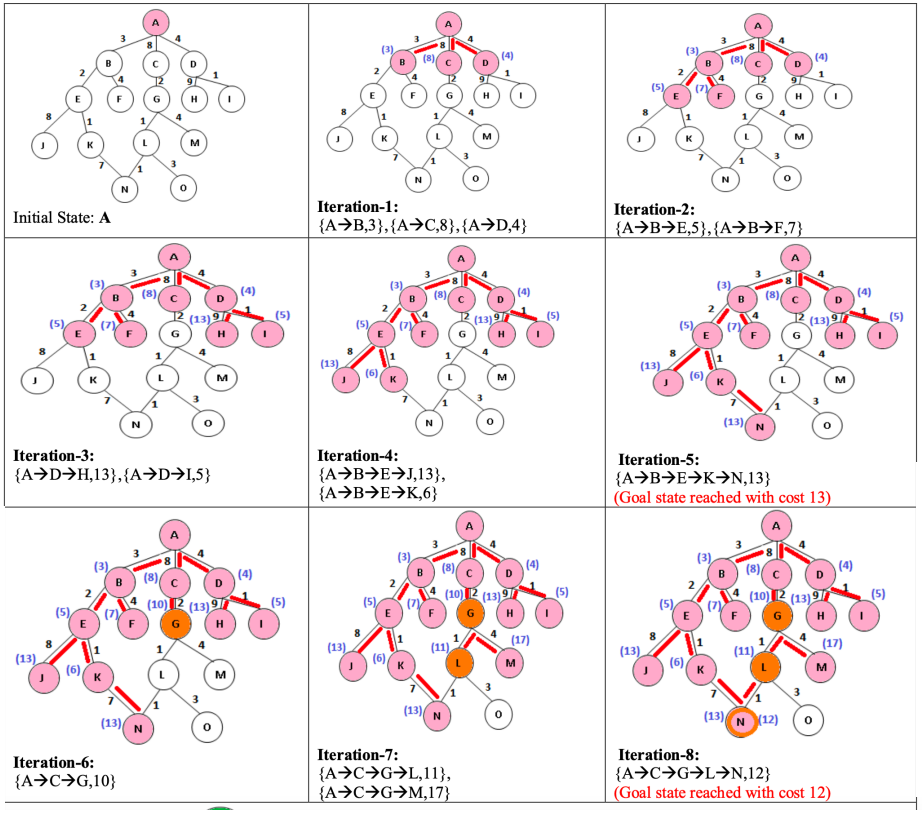
Will take a **long time to find solutions** with a large number of steps because must look at all shorter length possibilities first

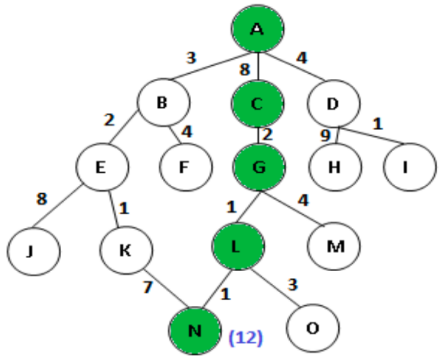
± A complete search tree of depth d where each non-leaf node has b children, has a total of $1 + b + b^2 + \ldots + b^d = (b^{(d+1)} - 1)/(b-1)$ nodes

UNIFORM COST SEARCH

The Uniform Cost Search (UCS) is a state space search algorithm in which it finds the goal state based on the cost to the node. That is, the nodes are expanded with minimum cost path.







Minimum cost from A to N is
{A→C→G→L→N, 12}

- **Complete?**
  Yes, if step cost is greater than some positive constant ε (we don't want infinite sequences of steps that have a finite total cost)
- **Optimal!**
  - if the cost of each step exceeds some positive bound ε.
- **Time complexity:** $O(b^{1 + C^*/\varepsilon})$
- **Space complexity:** $O(b^{1 + C^*/\varepsilon})$

  where C is the cost of the optimal solution*

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

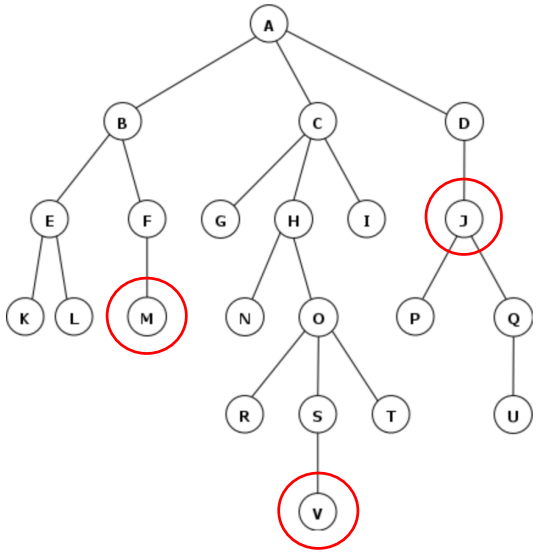
TABLE. IV ITERATIONS FOR UCS

Iteration	UCS Traversal (Path with cost)
Iteration – 1	{A→B,3},{A→C,8},{A→D,4}
Iteration – 2	{A→B→E,5},{A→B→F,7}
Iteration – 3	{A→D→H,13},{A→D→I,5}
Iteration – 4	{A→B→E→J,13},{A→B→E→K,6}
Iteration – 5	{A→B→E→K→N,13}
Iteration – 6	{A→C→G,10}
Iteration – 7	{A→C→G→L,11},{A→C→G→M,17}
Iteration – 8	{A→C→G→L→N,12}

DEPTH-FIRST SEARCH

Strategy: expand the deepest unexpanded node. Implementation: The fringe is a LIFO queue (stack)

For the following search tree: The stack evolves like this




```
A
B C D
E F C D
K L F C D
L F C D
F C D
M C D
C D
G H I D
H I D
N O I D
O I D
R S T I D
S T I D
V T I D
T I D
I D
D
J
P Q
Q
U
```

- is not complete (because of infinite depth and loops)
- is not optimal
- Time complexity (worst case: solution is at m): $O(b^m)$ — regardless of whether we test at generation or expansion time.
- Can be better than BFS for dense solution space
- Space complexity is $O(bm)$ — linear space

Again, If the goal nodes were M, V, and J, the Depth–First search above would find M

DEPTH-LIMITED SEARCH ALGORITHM:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is **$O(b^{\ell})$**.

Space Complexity: Space complexity of DLS algorithm is **$O(b \times \ell)$**.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.

DLS

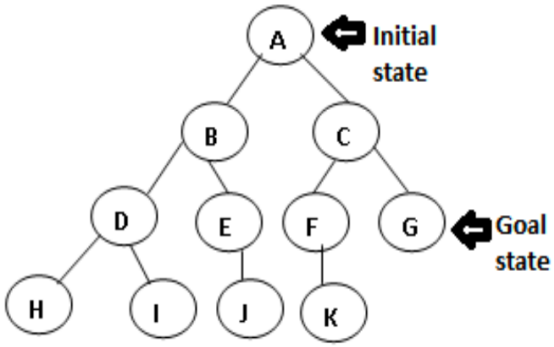


Fig. 9 Graph for DLS

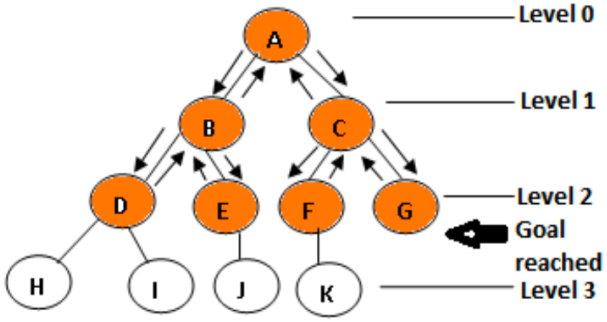


Fig. 10 Depth Limit Search (DLS) with $l = 2$

TABLE – V. OPEN AND CLOSE LIST FOR DLS

Open list (Unexplored nodes)	Close list (Visited nodes)
Depth bound ($l = 2$)	
Open=[A]	Close=[ ]
Open=[B,C]	Close=[A]
Open=[D,E,C]	Close=[A,B]
Open=[E,C]	Close=[A,B,D]
Open=[C]	Close=[A,B,D,E]
Open=[F,G]	Close=[A,B,D,E,C]
Open=[G]	Close=[A,B,D,E,C,F]
Open=[ ]	Close=[A,B,D,E,C,F,G]

ITERATIVE DEEPENING SEARCH

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

- Itcombines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Completeness:
This algorithm is complete is if the branching factor is finite.
Time Complexity:
If b is the branching factor and depth is d then the worst-case time complexity is **$O(b^d)$**.
Space Complexity:
The space complexity of IDDFS will be **O(bd)**.
Optimal:
IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node

IDS

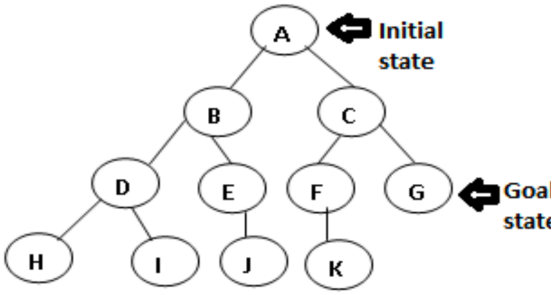
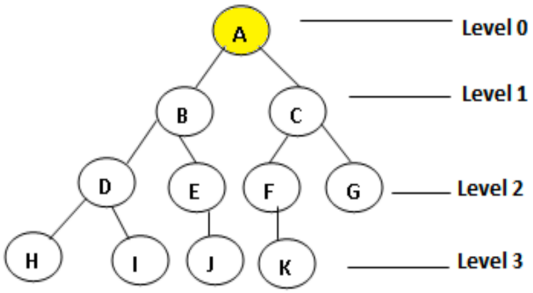
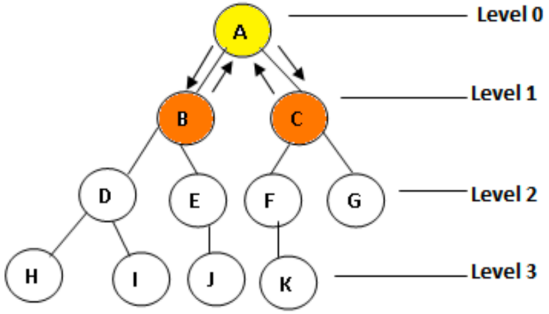


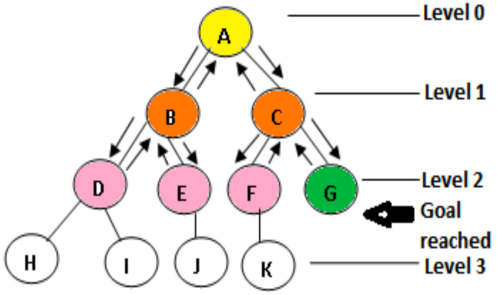
Fig. 5 Graph using IDS



IDS: Depth bound 0 (Level – 0)



IDS: Depth bound 1 (Level – 1)



IDS: Depth bound 0 (Level – 0)

TABLE.III RESULT TABLE FOR IDS:

Depth (Level)	Iterative Deepening Search
Level – 0	A
Level – 1	A,B,C
Level – 2	A,B,C,D,E,F,G

- ✓ Best First Search
- ✓ Greedy Search
- ✓ Beam Search
- ✓ A* Search
- ✓ Hill-Climbing Search

INFORMED HEURISTIC-BASED SEARCH STRATEGIES

Summary

UNINFORMED SEARCH VS INFORMED HEURISTIC-BASED SEARCH

Informed search strategies have some information on the goal state that helps them in performing more efficient searching. This information is obtained by means of a function that estimates how close a given state is to the goal state.

Uninformed search strategies, as the ones we have already covered, have no additional information on the goal state other than the one provided in the problem definition.

An Informed search strategy use a heuristic function to find the most promising path. A heuristic function takes the current state of the search agent as its input and produces the estimation of how close an agent is from the goal state.

A heuristic function is admissible if it never overestimates the true cost to a nearest goal.

A heuristic function is consistent if, when going from neighboring nodes a to b, the heuristic difference/step cost never overestimates the actual step cost.

Perfect heuristic: If $h(n) = h^*(n)$ for all n, then only the nodes on the optimal solution path will be expanded. So, no extra work will be performed.

Null heuristic: If $h(n) = 0$ for all n, then this is an admissible heuristic and the strategy acts like Uniform−Cost Search.

Better heuristic: If $h1(n) < h2(n) <= h^*(n)$ for all non−goal nodes, then $h2$ is a better heuristic than $h1$

If $A1$ uses $h1$, and $A2$ uses $h2$, then every node expanded by $A2$ is also expanded by $A1$. In other words, $A1$ expands at least as many nodes as $A2$.

- We say that $A2$ is better informed than $A1$.
- The closer $h(n)$ is to $h^*(n)$, the fewer extra nodes that will be expanded

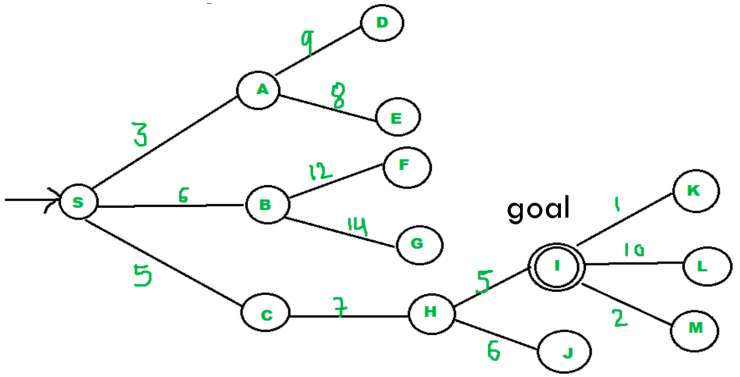
$h^*(n)$ is true cost of the minimal cost path from node n to a goal node.

BEST FIRST SEARCH

- Idea: use an evaluation function f(n) for each node
  - f(n) provides an estimate for the total cost.
  - Expand the node n with smallest f(n).
- We use a priority queue to store costs of nodes. So the implementation is a variation of BFS, we just need to change Queue to. a PriorityQueue.
- Implementation:
  Order the nodes in fringe increasing order of cost.

Analysis :

- In the **worst** case, the time and **space complexity** for **best- first search** is the same as with BFS: O(bd+1) for time and $O(b^{d+1})$ for **space.** Performance of the algorithm depends on how well the cost or evaluation function is designed.
- **Complete: Best First Search:** it is **complete** (finds a solution in finite graphs) like BFS?
- Optimal? No. it is not **optimal** (to find the least cost solution) as DFS; it is **optimal** when the cost of each edge is the same.



- We start from source "S" and search for goal "I" using given costs and Best First search.
- pq initially contains S
  We remove s from pq and process unvisited neighbors of S to pq.
  pq now contains {A, C, B} (C is put before B because C has lesser cost)
- We remove A from pq and process unvisited neighbors of A to pq. pq now contains {C, B, E, D}
- We remove C from pq and process unvisited neighbors of C to pq. pq now contains {B, H, E, D}
- We remove B from pq and process unvisited neighbors of B to pq. pq now contains {H, E, D, F, G}
- We remove H from pq.
- Since our goal "I" is a neighbor of H, we return.

BEST FIRST SEARCH VS GREEDY BEST FIRST SEARCH

The lecture notes did not make a clear distinction between Best First Search and Greedy Best Search.

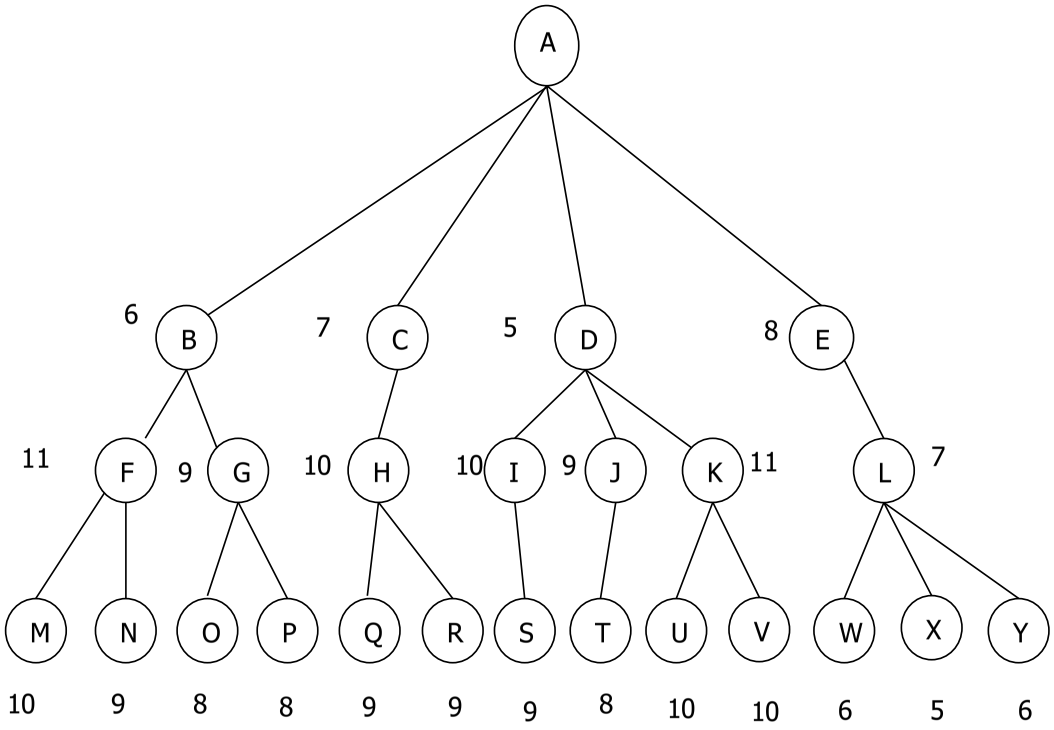
- BFS is an instance of tree search and graph search algorithms in which a node is selected for expansion based on the evaluation function f(n) = g(n) + h(n), where g(n) is length of the path from the root to n and h(n) is an estimate of the length of the path from n to the goal node.
- In a BFS algorithm, the node with the lowest evaluation (i.e. lowest f(n)) is selected for expansion.
- Greedy BFS uses the following evaluation function f(n) = h(n), which is just the heuristic function h(n), which estimates the closeness of n to the goal.
- Hence, greedy BFS tries to expand the node that is thought to be closest to the goal, without taking into account previously gathered knowledge (i.e. g(n)).

To summarize, the main difference between these (similar) search methods is the evaluation function.

- Greedy best-first search is an interesting variant of best-first search. However, since it only considers h(n), that is f(n) = h(n), to determine which node to select next (identified as the closeness to the goal), it's defined as greedy. Because of this, greedy best-first is not complete as the heuristic is not admissible (because it can overestimate the path to the goal).
- **GBFS**
  - **is not Complete**– can get stuck in loops
    Optimal: No
    Time Complexity: $O(b^{m+1})$, but a good heuristic can give dramatic improvement
  - **Space Complexity:** $O(b^{m+1})$ -- keeps all nodes in memory

BEAM SEARCH

- Beam search attempts to minimize the memory requirement of the Breadth-First algorithm.
- It is an informed Breadth-First algorithm.
- It expands only the first ß promising nodes at each level. ß is called Beam Width
- Use an evaluation function f(n) = h(n), but the maximum size of the nodes list is ß.
- It only keeps ß best nodes as candidates for expansion, and throws the rest away
- More space efficient than greedy search, but may throw away a node that is on a solution path.
- Optimal: no
- Complete: no
- Time Complexity: $O(ß^d)$
- Space Complexity: $O(ß^d)$
- Not admissible



Assume J to be the target state, list the order of expansion with ß=2

A* SEARCH STRATEGY

- A* strategy is one of the best and popular techniques used for path finding and graph traversals.
- A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.
- It is essentially a best first search algorithm.
- It maintains a tree of paths originating at the start node.
- It extends those paths one edge at a time.
- It continues until its termination criterion is satisfied.

A* Algorithm extends the path that minimizes the following function **f(n)** = **g(n)** + **h(n)**

- 'n' is the last node on the path
- g(n) is the cost of the path from start node to node 'n'
- h(n) is a heuristic function that estimates cost of the cheapest path from node 'n' to the goal node

Time and space complexity: $\mathcal{O}(\min(b^{d+1}, b|S|))$
(especially space complexity is problematic in practice)

Completeness: yes, because it accounts for path costs

Optimality: yes, but provided the heuristic h is optimistic:

- The catch with A* is that even though it is complete, optimal and optimally efficient (no algorithm with the same heuristic is guaranteed to expand fewer nodes), it still can't always be used, because for most problems, the number of nodes within the goal contour search space is still exponential in the length of the solution.

EXAMPLE

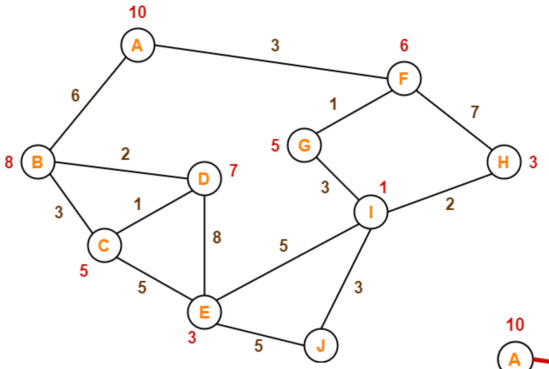
Step 01: We start with node A.
Node B and Node F can be reached from node A.
A* Algorithm calculates f(B) and f(F).
f(B) = 6 + 8 = 14
f(F) = 3 + 6 = 9
Since f(F) < f(B), so it decides to go to node F.
Path- A → F

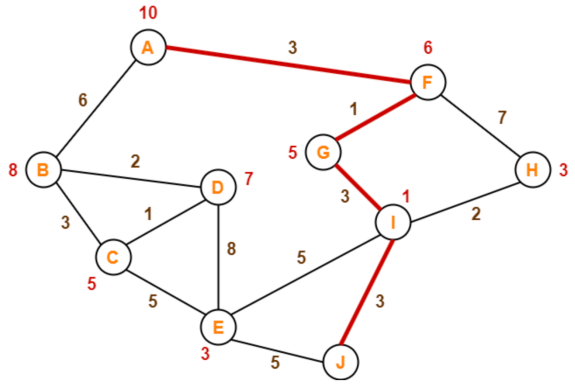
Step-02: Node G and Node H can be reached from node F.
A* Algorithm calculates f(G) and f(H).
f(G) = (3+1) + 5 = 9
f(H) = (3+7) + 3 = 13
Since f(G) < f(H), so it decides to go to node G.
Path- A → F → G

Step-03:
Node I can be reached from node G.
A* Algorithm calculates f(I).
f(I) = (3+1+3) + 1 = 8
It decides to go to node I.
Path- A → F → G → I

Step-04:
Node E, Node H and Node J can be reached from node I.
A* Algorithm calculates f(E), f(H) and f(J).
f(E) = (3+1+3+5) + 3 = 15
f(H) = (3+1+3+2) + 3 = 12
f(J) = (3+1+3+3) + 0 = 10
Since f(J) is least, so it decides to go to node J.
Path- A → F → G → I → J
This is the required shortest path from node A to node J

Find the most cost–effective path to reach from start state A to final state J using A* Algorithm.



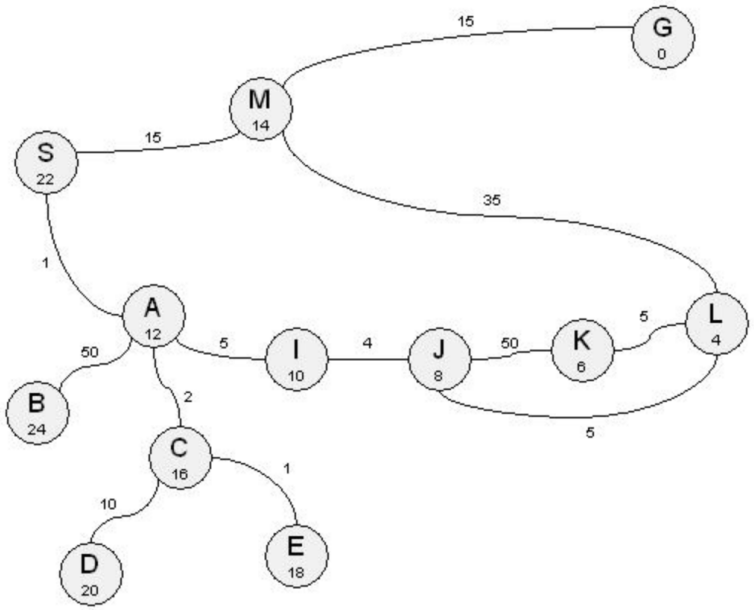


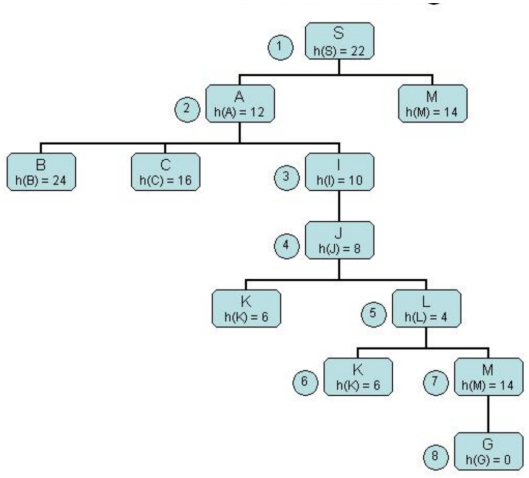
completeness, optimality, and optimal efficiency

HILL CLIMBING SEARCH

Algorithm for Simple Hill Climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
  a. If it is goal state, then return success and quit.
  b. Else if it is better than the current state then assign new state as a current state.
  c. Else if not better than the current state, then return to step2.
- **Step 5:** Exit.






```
ill Climbing
  Expanded  Queue
     S      [22<S>]
     A      [12<A,S> 14<M,S>]
     I      [10<I,A,S> 16<C,A,S> 24<B,A,S> 14<M,S>]
     J      [8<J,I,A,S> 16<C,A,S> 24<B,A,S> 14<M,S>]
     L      [4<L,J,I,A,S> 6<K,J,I,A,S> 16<C,A,S> 24<B,A,S> 14<M,S>]
     K      [6<K,L,J,I,A,S> 14<M,L,J,I,A,S> 6<K,J,I,A,S> 16<C,A,S> 24<B,A,S> 14<M,S>]
     M      [14<M,L,J,I,A,S> 6<K,J,I,A,S> 16<C,A,S> 24<B,A,S> 14<M,S>]
     G      [0<G,M,L,J,I,A,S> 6<K,J,I,A,S> 16<C,A,S> 24<B,A,S> 14<M,S>]
     goal reached!
```

Hill Climbing Example

n-queens

- Starting from a randomly generated 8-queens state, **steepest-ascent hill climbing** gets stuck 86% of the time, solving only 14% of problem instances.

- The Hill Climbing algorithm halts if it reaches a **plateau.**
  - One possible solution is to allow **sideways move** in the hope that the **plateau** is really a **shoulder**.
  - If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder.
  - One common solution is to put a limit on the number of consecutive sideways moves allowed.
  - For example, we could allow up to 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill climbing from 14% to 94%.

Hill Climbing Example

8-puzzle

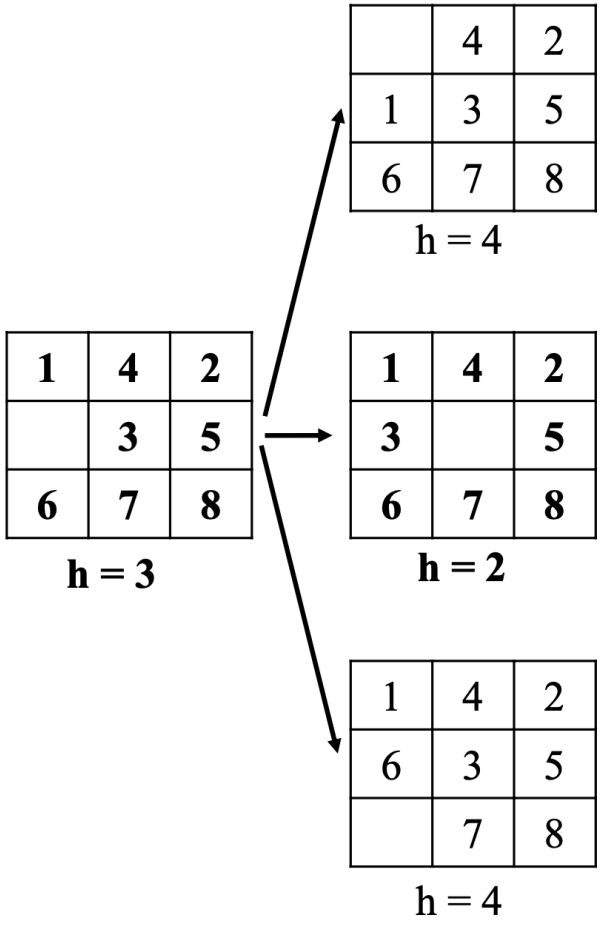
Heuristic function is Manhattan Distance

1	4	2
	3	5
6	7	8

h = 3

Hill Climbing Example

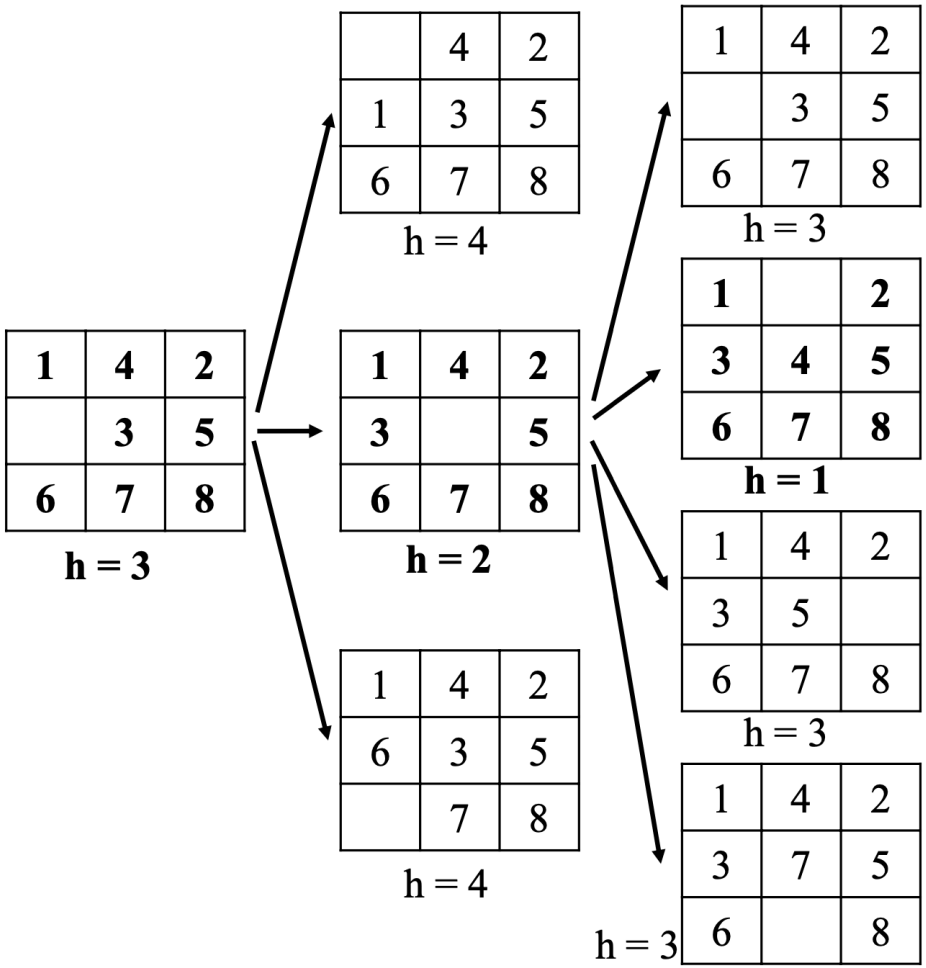
8-puzzle



Heuristic function is Manhattan Distance

Hill Climbing Example

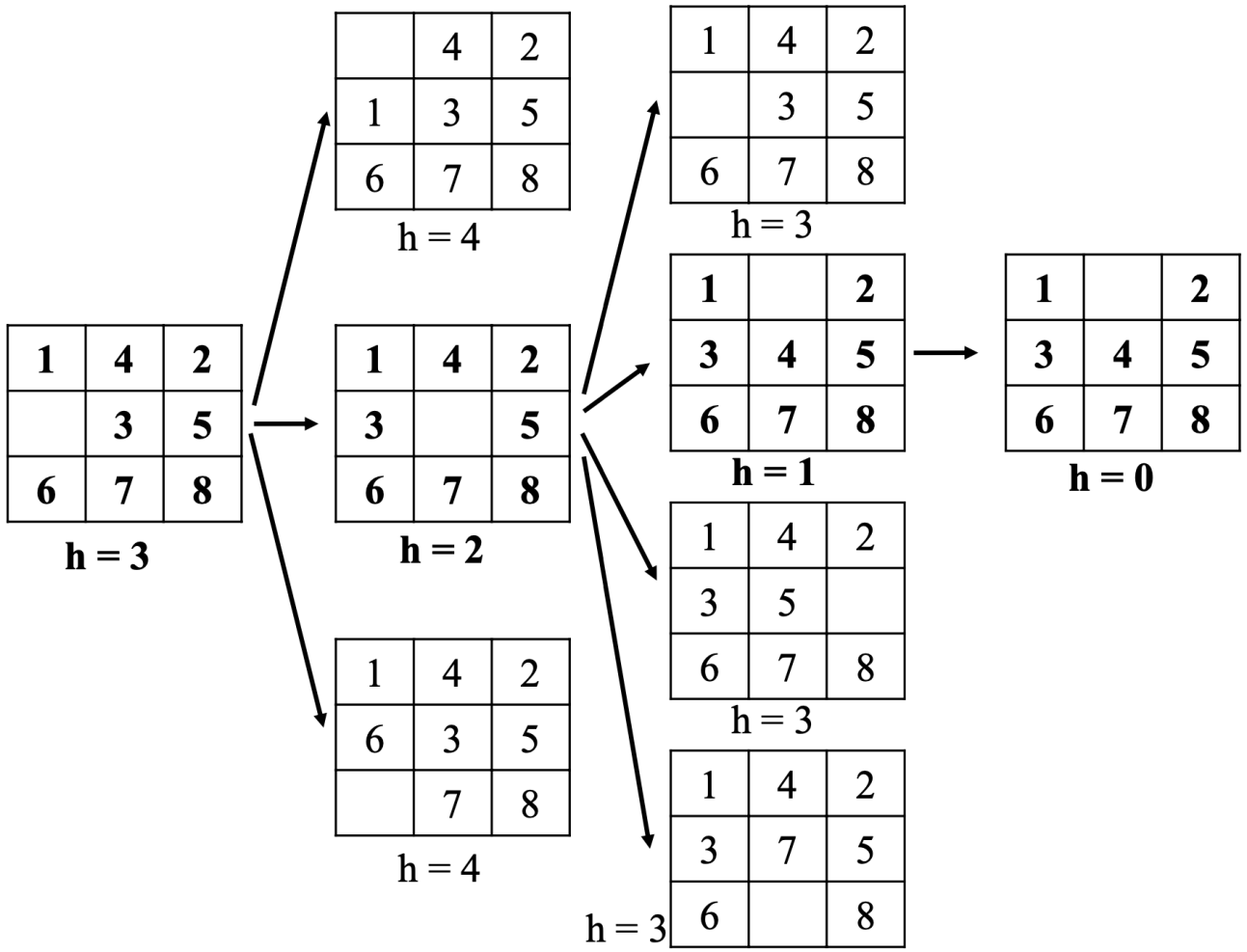
8-puzzle



Heuristic function is Manhattan Distance

Hill Climbing Example

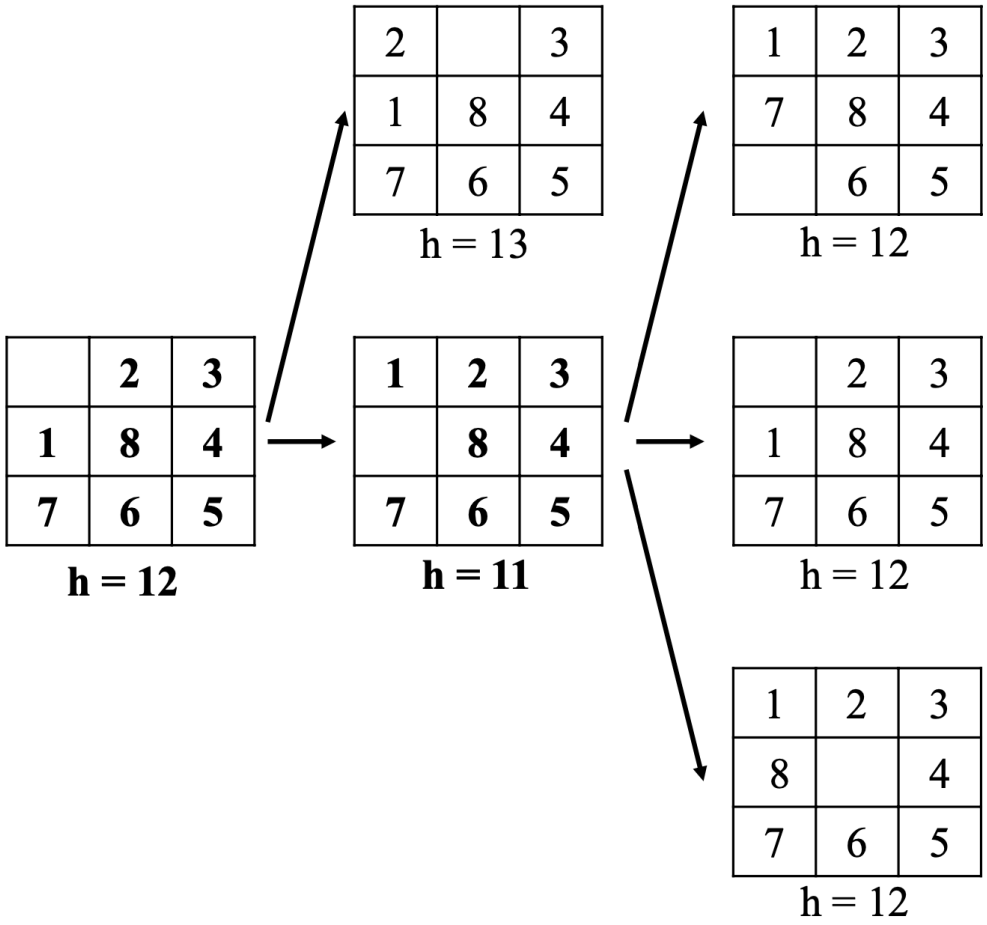
8-puzzle: a solution case



Heuristic function is Manhattan Distance

Hill Climbing Example

8-puzzle: stuck at local maximum



Heuristic function is Manhattan Distance

We are stuck with a local maximum.

- Hill Climbing is **NOT complete**.
- Hill Climbing is **NOT optimal**.
- has a time **complexity** of O(∞)
- but a space **complexity** of O(b).

TWO- PLAYER GAMES AS SEARCH

Summary

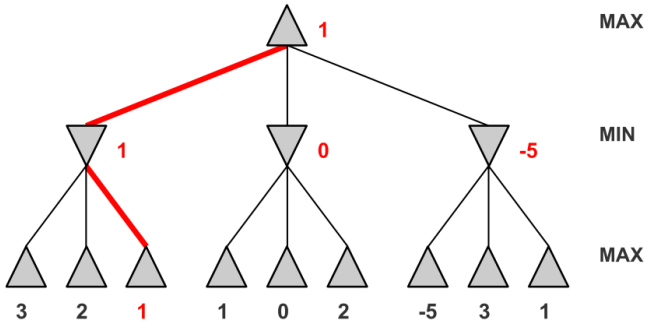
MINMAX GAME PLAYING

- Think of player MAX to be the computer and MIN to be the opponent
- MAX tries to maximize its win, whereas MIN tries to minimize MAX's win
- Players take turn
- **Minimax** algorithm finds an optimal strategy that minimizes the maximum expected loss that an opponent can inflict
- In reality it is impossible to search through the complete game tree, thus we cut off the search at a certain depth and use a heuristic **function** to estimate the values of game states
- Different players use different heuristic functions. The opponent's heuristic is unknown.

To determine the optimal strategy of a player whose turn is next, we compute the minimax value of the current node.

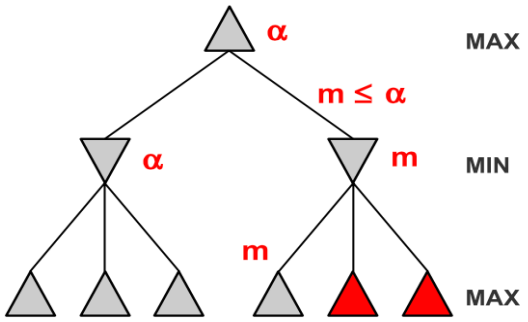
Optimal strategy

- ✓ MAX player's optimal strategy is the one that ensures the highest win, assuming that MIN player uses the same strategy
- ✓ Each player chooses a strategy so as to minimize the maximum loss

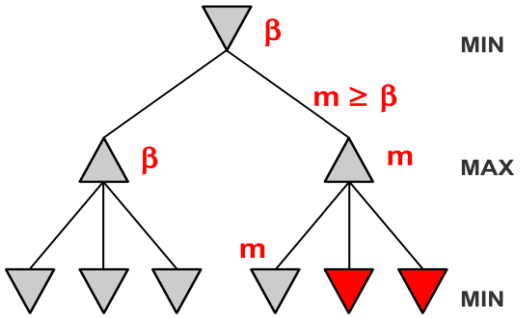


ALPHA-BETA PRUNING

- We prune every time we're certain that the unexplored moves **can under no circumstances be better** than the best move found so far
- If pruning below the MIN node: **alpha pruning**
- If pruning below the MAX node: **beta pruning**



α – the largest MAX value found



β – the smallest MIN value found

- ✓ Alpha-beta pruning reduces the number of nodes to traverse
- ✓ Things we didn't get in to multiplayer games, games that include

an element of chance, etc

COMPUTATIONAL COMPLEXITY

- In practice, the opponent's strategy is unknown (most probably different from that of MAX player) and therefore the opponent's moves cannot be predicted perfectly.
- Therefore, in order to make the optimal move, in each turn the players need to re-compute their optimal strategy, starting from the current position as the root of the game tree
- Minimax is a depth-first search, thus its space complexity is O(m), where m is the depth of the game-tree
- However, time complexity is $O(b^m)$, where b is the game branching factor.

Alpha-Beta Complexity improvement

- if you choose to expand the tree down to d or m levels, then the depth first search strategy (the search strategy we use in minmax) is O(b^m), O(b^d), respectively).
- Alpha-Beta pruning is expected to reduce that depth to $m/2$, or $d/2$, leading to an order of complexity O(b^(m/2), O(b^(d/2), respectively.
- However, in large space complex games, it is hard to expand the tree down to its maximum depth, rather we limit the depth to a reasonable level m.
- With limited memory resources, with Alpha-beta this depth m could be twice the m w/o the alpha-beta pruning;
- Thus, if we are driven by the desire to explore more moves without increasing the search cost, then alpha-beta will lead to more moves and no incremental complexity cost. If the desire is to reduce the search cost for our target depth m. Then alpha-beta would bring the cost down to O(b^(m/2) as opposed to O(b^m), the later being the cost for depth m w/o alpha-beta. what matters is the depth x you choose to be the depth limit.