Chapter- 3: Searching

- Searching the process finding the required states or nodes.
- Searching is to be performed through the state space.
- Search process is carried out by constructing a search tree.
- Search is a universal problem-solving technique.
- Search involves systematic trial and error exploration of alternative solutions.
- Many problems don't have a simple algorithmic solution. Casting these problems as search problems is often the easiest way of solving them.
- Useful when the sequence of actions required to solve a problem is not known
    - Path finding problems, e.g, eight puzzle, travelling salesman problem
    - Two player games, e.g., chess and checkers
    - Constraint satisfaction problems, e.g., eight queens

Steps in Searching

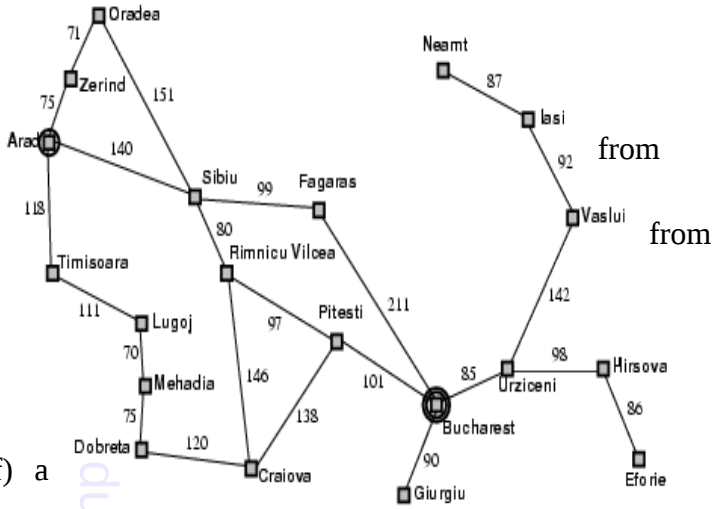
1. Check whether the current state is the goal state or not?
2. Expand the current state to generate the new sets of states.
3. Choose one of the new states generated for search depending upon search strategy.
4. Repeat step 1 to 3 until the goal state is reached or there are no more state to be expanded.

Problem Formulation

- A search problem is defined in terms of states, operators and goals
- A state is a complete description of the world for the purposes of problem-solving
    - Initial state is the state the world is in when problem solving begins
    - Goal state is a state in which the problem is solved
- An operator is an action that transforms one state of the world into another state
- Goal states
    - Depending on the number of solutions a problem has, there may be a single goal state or many goal states:
    - In the eight-puzzle there is a single solution and a single goal state
    - In chess, there are many winning positions, and hence many goal states
- Applicable operators
    - In general, not all operators can be applied in all states
    - In a given chess position, only some moves are legal (as defined by the rules of chess)
    - In a given eight-puzzle configuration, only some moves are physically possible
    - The set of operators which are *applicable in a state s determine the* states that can be reached from *s*

Example : Route Planning

- State: Being in any one city
- Initial State: Being in **Arad**
- Goal State: Be in **Bucharest**
- Operators: actions of driving city X to city Y along the connecting roads, e.g, driving from Arad to Sibiu



State vs Nodes

- A state is a (representation of) a physical configuration
- Nodes in the search tree are data structures maintained by a search procedure representing *paths to a particular state*
- The same state can appear in several nodes if there is more than one path to that state
- The path can be reconstructed by following edges back to the root of the tree

Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - ***Completeness***: does it generate to find a solution if there is any?
  - ***Optimality***: does it always find the highest quality (least-cost) solution?
  - ***Time complexity***: How long does it take to find a solution?
  - ***Space complexity***: How much memory does it need to perform the search?
- 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 ∞)

Types of Search

i. Uninformed (Blind) Search
ii. Informed (Heuristic)Search

Uninformed search strategies

- These types of search strategies are provided with the problem definition and these don't have additional information about the state space.
- These can only expand current state to get a new set of states and distinguish a goal state from non-goal state.
- Uninformed search strategies use only the information available in the problem definition
- Less effective than informed search.

- Distinguish goal state from non goal state
  - Breadth-first search

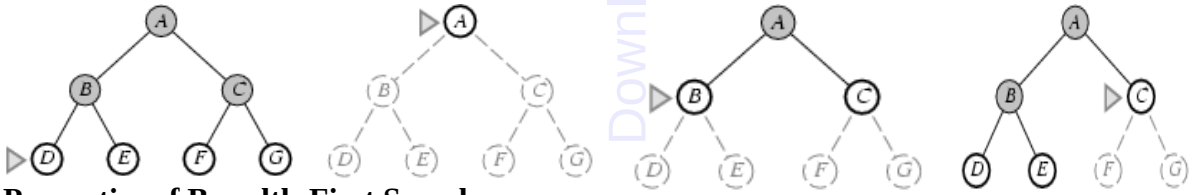
- Uniform cost search
- Depth-first search
- Depth-limited search
- Iterative Deepening Depth-first search
- Bidirectional Search

Breadth-First Search (BFS)

- Proceeds level by level down the search tree
- Starting from the root node (initial state) explores all children of the root node, left to right
- If no solution is found, expands the first (leftmost) child of the root node, then expands the second node at depth 1 and so on …
- Process
    - Place the start node in the queue
    - Examine the node at the front of the queue
        - If the queue is empty, stop
        - If the node is the goal, stop
        - Otherwise, add the children of the node to the end of the queue

BFS example (Find path from A to D)

Put the start node in the queue, Examine the first element of the queue. If it is the goal, stop, otherwise put its children in the queue



Properties of Breadth-First Search

- ***Completeness:*** Complete if the goal node is at finite depth
- ***Optimality:*** It is guaranteed to find the shortest path
- ***Time complexity***
    - For branching factor b and depth level d
    - Expand root yields b nodes at 1^{st} level
    - Expand 1^{st} level yields b^2 noes at 2^{nd} level.
    - If the goal is in d^{th} level, in the worst case, the goal node would be the last node in the d^{th} level.

- We should expand (b^d-1) nodes in the d^{th} level (except the goal node itself). Total nodes in d^{th} level = $b(b^d-1)= b^{d+1}-b$
- Total no of nodes generated = $1+b+b^2+b^3+…..+b^{d+1}-b$
- Time complexity = $O(b^{d+1})$

- ***Space Complexity***: $O(b^{d+1})$

Weakness of BFS

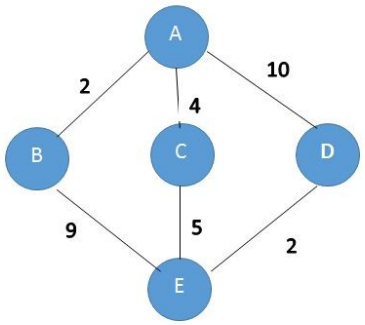
- *High time and memory requirements*

Uniform Cost Search

- Breadth first Search finds the shallowest goal but it's not always sure to find the optimal solution.
- Uniform cost search can be used if the cost of travelling from one node to another is available.
- Uniform cost search always expands the lowest cost node on the fringe (the collection of nodes that are waiting to be expanded.)
- The first solution is guaranteed to be the cheapest one because a cheaper one is expanded earlier and so would have been found first.

Uniform cost search example (Find path from A to E)

- Expand A to B,C,D
- The path to B is the cheapest one with path cost 2.
- Expand B to E
  - Total path cost = 2+9 =11
- This might not be the optimal solution since the path AC as path cost 4 (less than 11)
- Expand C to E
  - Total path cost = 4+5 =9
- Path cost from A to D is 10 (greater than path cost, 9)
- Hence optimal path is ACE



Disadvantage

Does not care about the no of steps a path has but only about their cost. Hence it might get stuck in an infinite loop if it expands a node that has a zero cost action leading back to same state

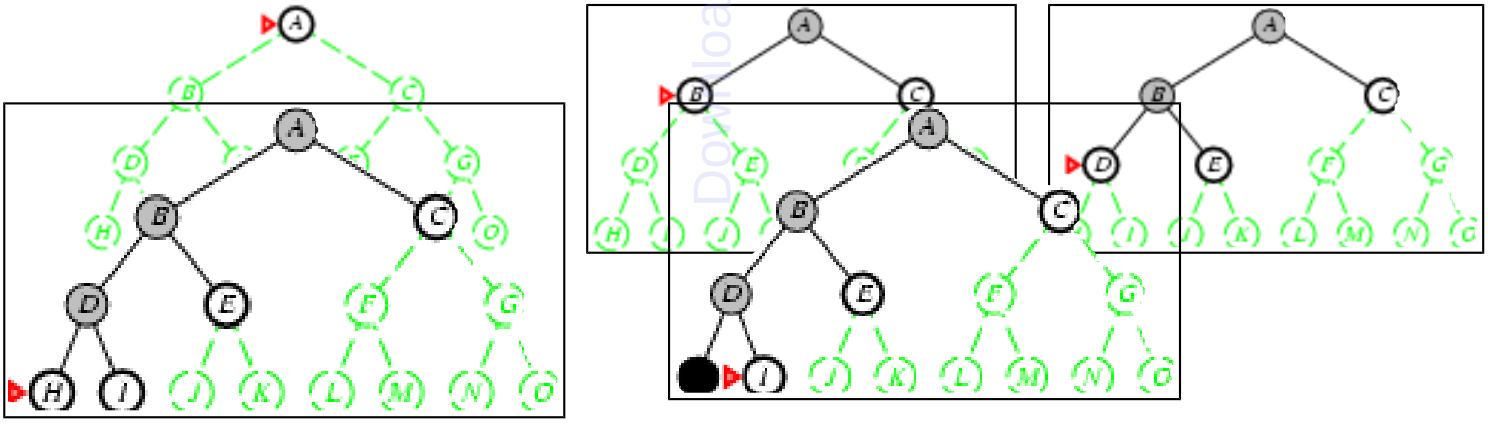
Properties of uniform cost search

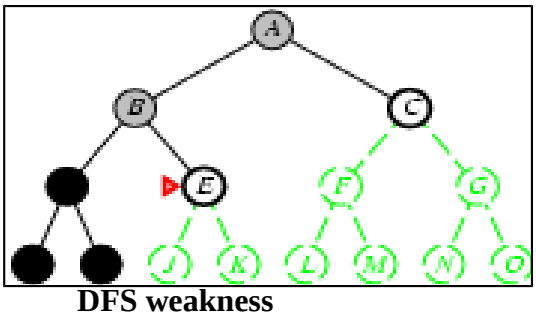
- ***Completeness*:** Complete if the cost of every step is greater than or equal to some small positive constant **e**
- ***Optimality*:** Optimal if the cost of every step is greater than or equal to some small positive constant **e**
- ***Time complexity* :** $O(b^{c^*/e})$ where c* is cost of optimal path, e is small positive constant
- ***Space Complexity :*** $O(b^{c^*/e})$

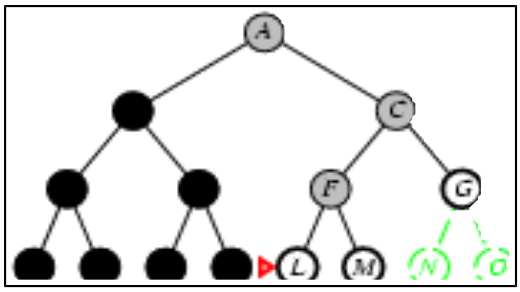
Depth-First Search (DFS)

- DFS proceeds down a single branch of the tree at a time.
- It expands the root node, then the leftmost child of the root node, then the leftmost child of that node etc.
- Always expands a node at the deepest level of the tree
- Only when the search hits a dead end (a partial solution which can't be extended) does the search *backtrack and expand nodes at higher levels.*
- **Process: Use stack to keep track of nodes. (LIFO)**
    - Put the start node on the stack
    - While stack is not empty
        - Pop the stack
        - If the top of stack is the goal, stop
        - Other wiser push the nodes connected to the top of the stack on the stack (provided they are not already on the stack)

DFS example (find path from A to L)







DFS weakness

- We may get stuck going down an infinite branch that doesn't lead to a solution.
- As in given figure, suppose our solution is on right branch at depth level 1, but DFS search proceeds down left branch and continues its search upto 1000's of nodes. It might get stuck in that branch without solution.

Properties of depth-first search

- ***Completeness***: Incomplete as it may get stuck down going down an infinite branch that doesn't leads to solution.
- ***Optimality***: The first solution found by the DFS may not be shortest.
- ***Space complexity***: b as branching factor and d as tree depth level, Space complexity = **O(b.d)**
- ***Time Complexity*: O(b^d)**

Depth–Limited Search

- Breadth first has computational, especially, space problems.
- Depth first can run off down a very long (or infinite) path. Solution may be not be optimal.
- Depth limited search.
  - Perform depth first search but only to a pre-specified depth limit L.
  - No node on a path that is more than L steps from the initial state is placed on the Frontier.
  - We "truncate" the search by looking only at paths of length L or less.
- Now infinite length paths are not a problem.
- But will only find a solution if a solution of length ≤ L exists.

Properties of depth limit search

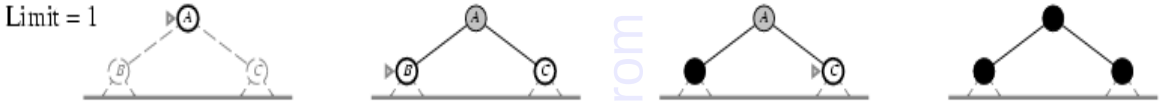
- ***Completeness***: Incomplete as solution may be beyond specified depth level.
- ***Optimality***: not optimal
- ***Space complexity***: b as branching factor and l as tree depth level,
  Space complexity = **O(b.l)**
- ***Time Complexity***: **O(b^l)**

Iterative Deepening DFS

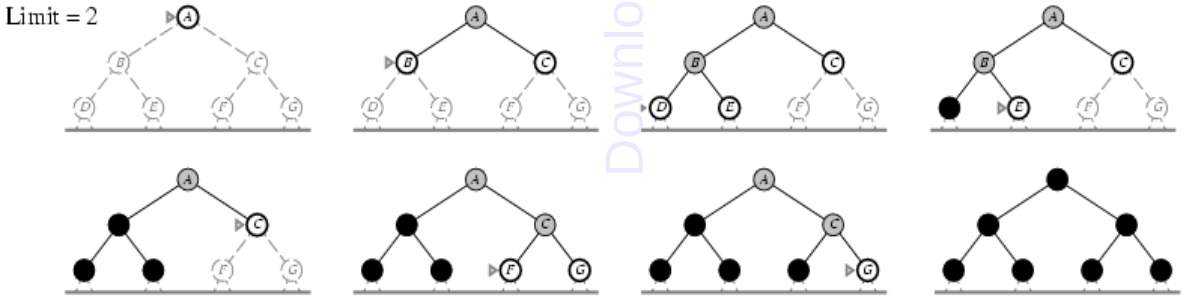
- Take the idea of depth limited search one step further.
- Starting at depth limit L = 0, we iteratively increase the depth limit, performing a depth limited search for each depth limit.
- Stop if no solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit.
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms
- Search is helpful only if the solution is at given depth level.

An example of Iterative deepening DFS (depth level 3)

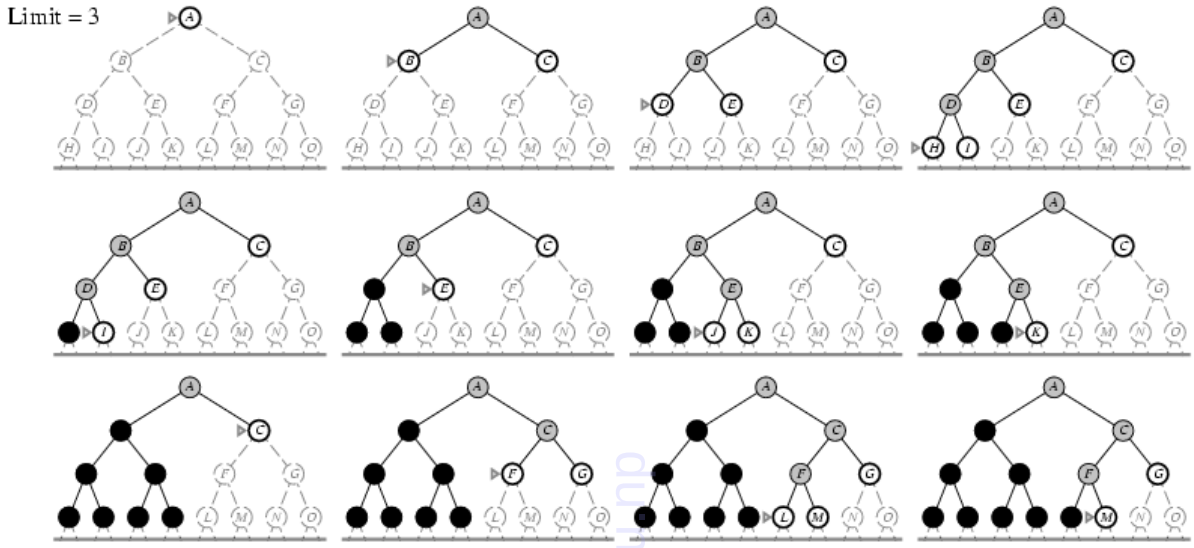
First Iteration Search at level l=0



Second Iteration Search at level l=1



Third Iteration Search at level l=2



Fourth Iteration Search at level l=3

Informed Search (Heuristic Search)

- In uninformed search, we don't try to evaluate which of the nodes on the frontier are most promising. We never "look-ahead" to the goal
- Informed search have problem specific knowledge apart from problem definition.
- Use of Heuristic improves efficiency of search process.
- The idea is to develop a domain specific heuristic function h(n).
- h(n) guesses the cost of getting to the goal from node n.

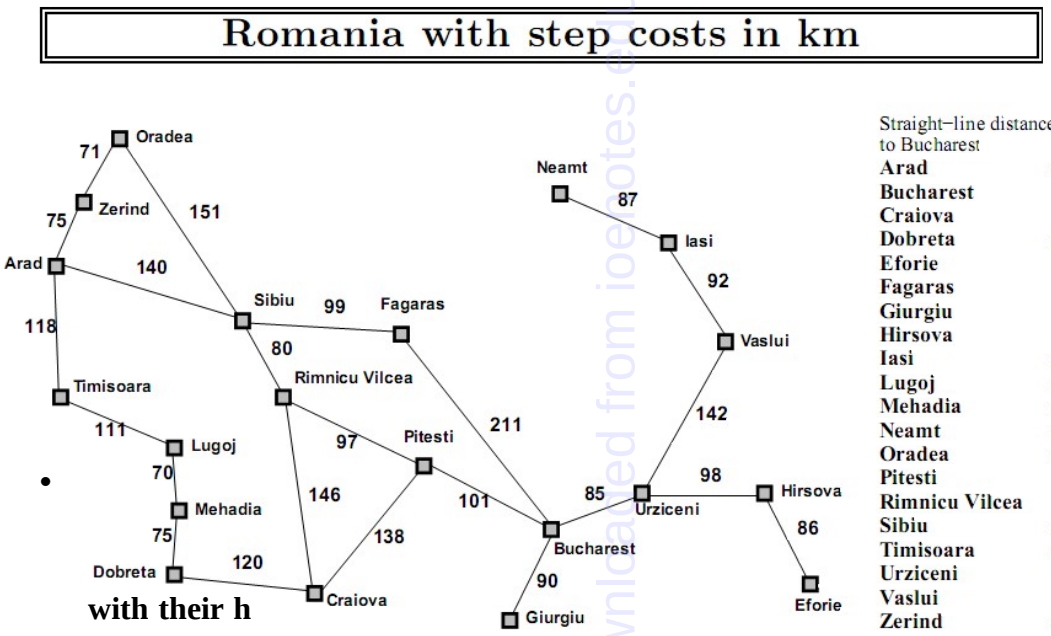
Best first Search

- A node is selected for expansion based on evaluation function f(n)
- Node with lowest evaluation function is expanded first.
- The evaluation function must represent some estimate of the cost of the path from state to the closest goal state.
- One important heuristic function is h(n)
- h(n) is the estimate cost of the cheapest path from node to the goal
- Types
    - Greedy Best First Search
    - A* search

Greedy best-first search

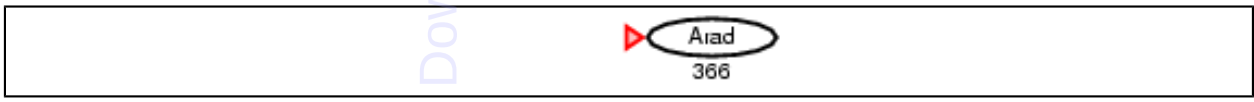
- It tries to get as close as it can to the goal.
- It expands the node that appears to be closest to the goal
- It evaluates the node by using heuristic function only.
- Evaluation function *f(n) = h(n)*
    - (heuristic)= (estimate of cost from *n* to *goal)*
    - *h(n) = 0 for goal state*

example (find path from Arad to Bucharest)



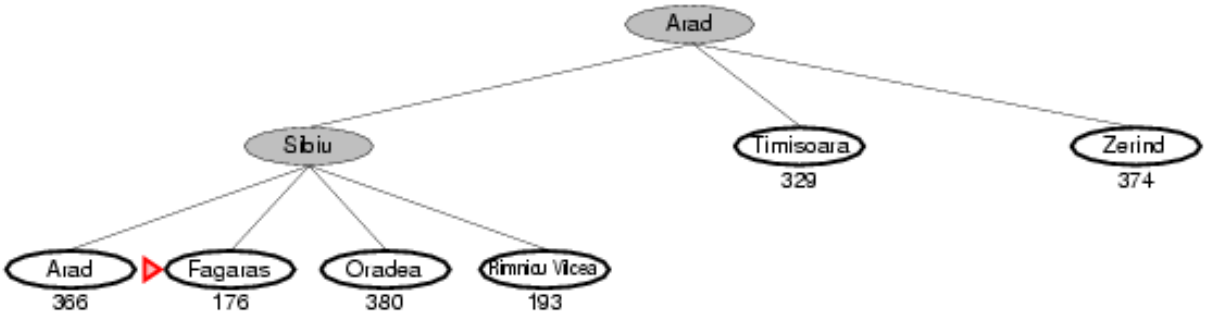
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

- **Goal: to reach Bucharest from Arad. Nodes are labeled with their h value**

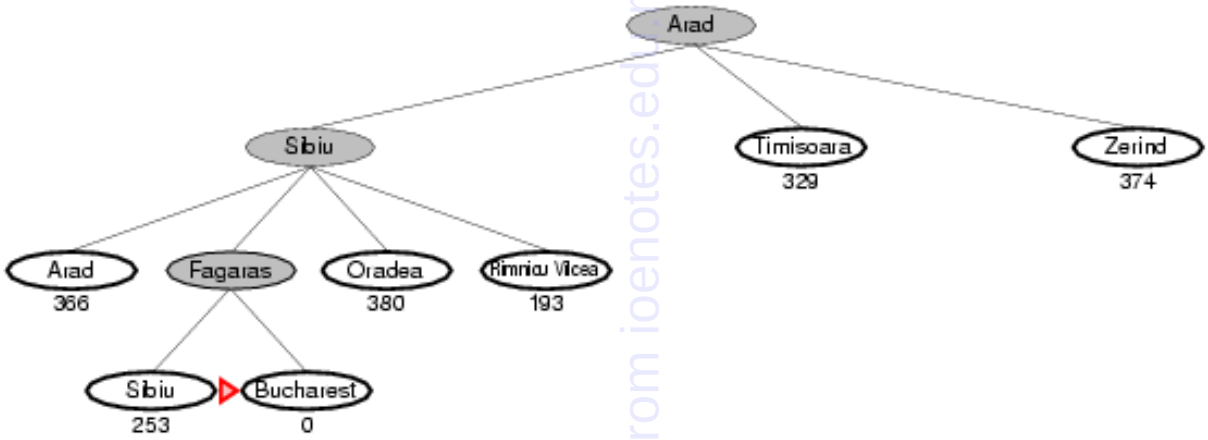


Start from source node

Evaluate cities connected to source and choose the one with lowest h(distance) value



Continue evaluating cities with their distance to destination



Continue evaluating until destination is reached

Properties of greedy best-first search

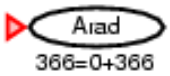
- Complete? No – can get stuck in loops, e.g., Iasi Neamt Iasi Neamt
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

A* search

- Greedy best first search analyses nodes with lowest cost of node n to reach goal. And it may get stuck in search of goal.

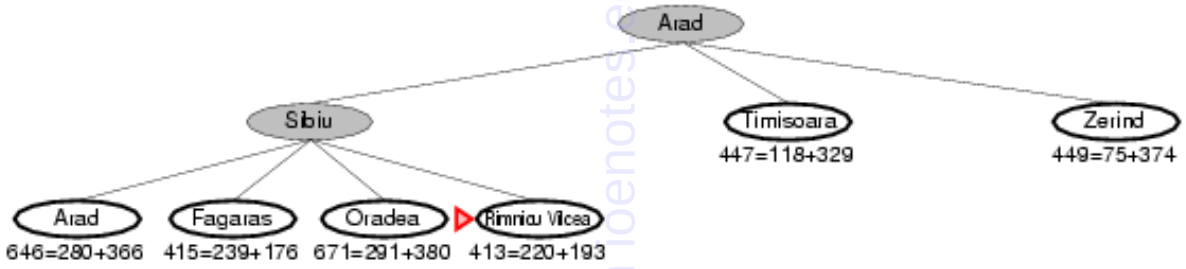
- Idea: avoid expanding paths that **are already expensive**
  Evaluation function *f(n)* = *g(n)* + *h(n)*
  g(n) = cost so far to reach *n*
  h(n) = estimated cost from *n* to goal
  f(n) = estimated total cost of path through *n* to goal

A* search example (find path from Arad to Bucharest)

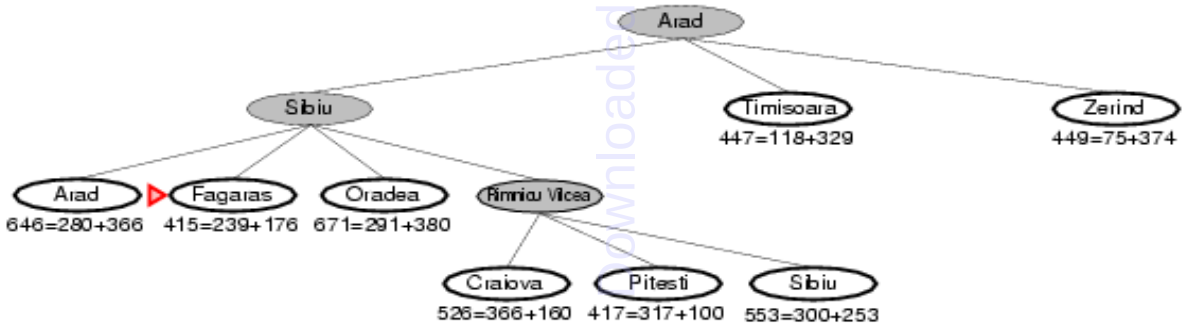


Start with source node

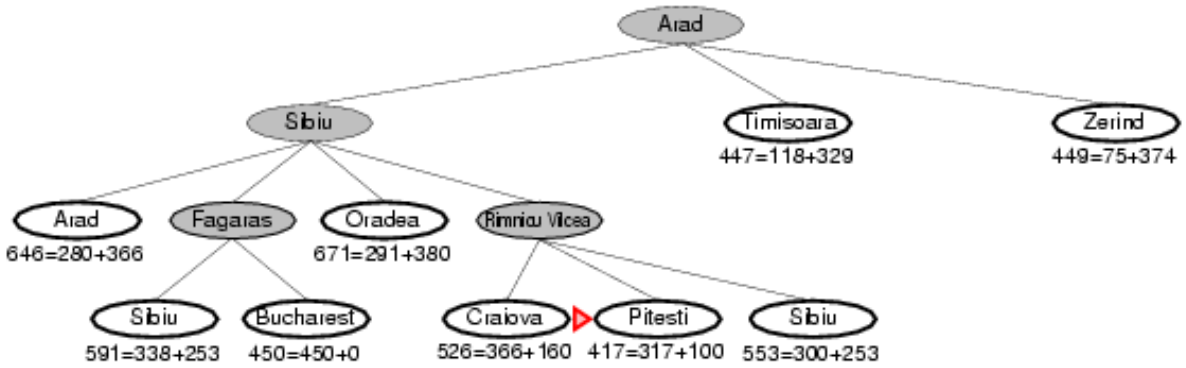
Evaluate nodes connected to source. Evaluate f(n)=g(n)+h(n) for each node. Select node with lowest f(n) value. *Sibiu node is chosen*



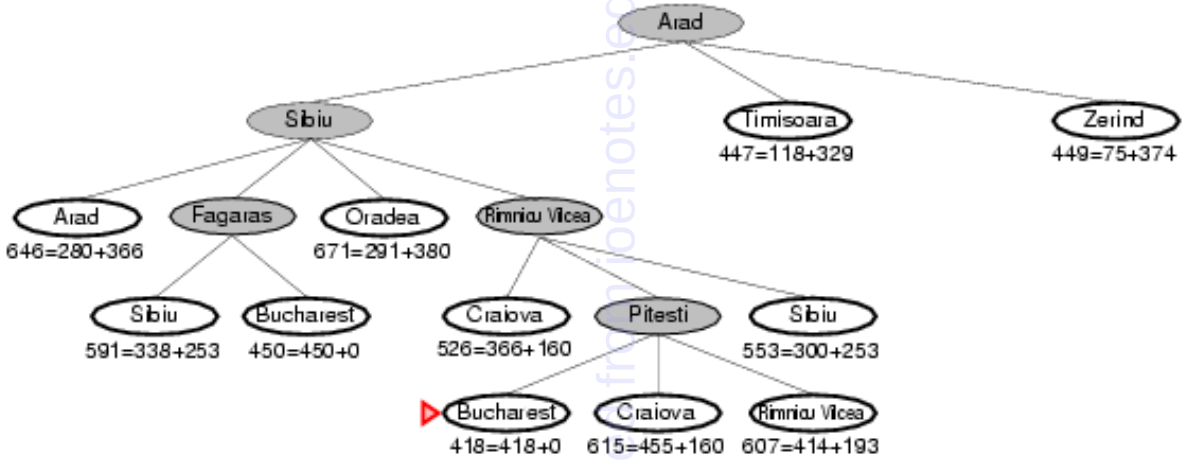
Rimniou vicea node has lowest f(n) value, so this city is chosen



In all nodes expanded *Fagaras* has lowest f(n) value, so this node is expanded next



Expanding *Fagaras* node reaches our destination Bucharest with f(n) value 450. So all the nodes with f(n) values expensive than 450 are avoided as evaluating them would result in cost of path expensive than our path already achieved. The nodes with f(n) value lower than 450 are evaluated next.



Evaluating next node with *Petesti* with cost 417 (lower than 450 –Arad-Sibiu-Fagaras-Bucharest) we reach Bucharest with cost 418 which is lower than our earlier search result. All other nodes are expensive than our new result i.e. cost of 418. Hence our search is stopped here.

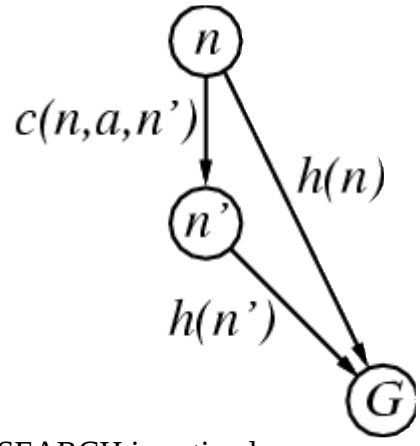
Admissible heuristics

- A heuristic *h(n)* is admissible if for every node *n*,
  $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from *n*.
  Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- **Theorem**: If *h(n)* is admissible, A* using GRAPH-SEARCH is complete
- Space Complexity = $O(b^d)$
- Time Complexity = $O(b^d)$

Consistent heuristics

- A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*,

  $h(n) \leq c(n,a,n') + h(n')$



- If *h* is consistent, we have

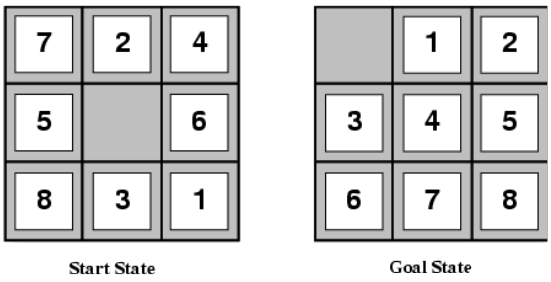
  f(n') = g(n') + h(n')
  = g(n) + c(n,a,n') + h(n')
  ≥ g(n) + h(n)
  = f(n)

- i.e., *f(n)* is non-decreasing along any path.
- **Theorem**: If *h(n)* is consistent, A* using GRAPH-SEARCH is optimal

Admissible heuristics example (8 puzzle)

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance (i.e., the sum of the distances of the tiles from their goal positions)
- $h_1(S) = ?$ 8
- $h_2(S) = ?$ 3+1+2+2+2+3+3+2 = 18



Local search and Optimization

- Local search
  - Keep track of single current state
  - Move only to neighboring states
- Advantages
  - Use very little memory

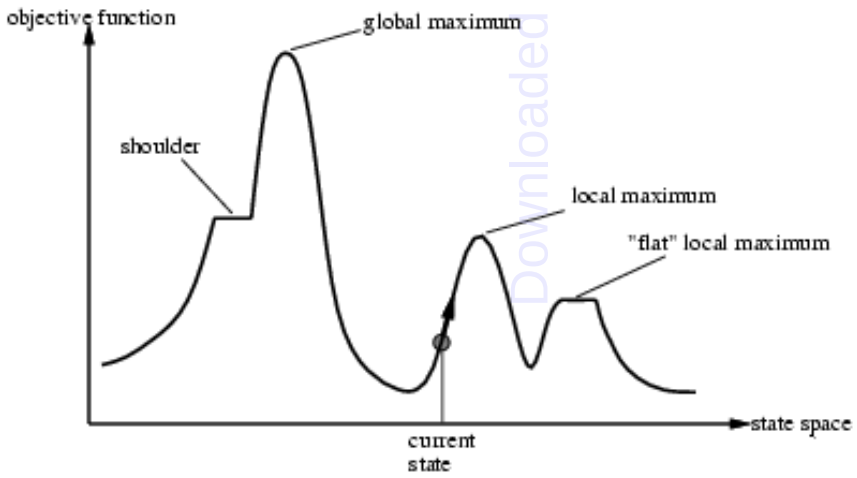
Iterative Improvement Algorithm

1. **Hill climbing search**
2. **Simulated Annealing search**

Hill climbing search

Hill climbing search is sometimes compared to climbing Mount Everest in thick fog with amnesia i.e. destined to reach top of Everest but due to thick fog (poor visibility) and forgetting what top of Everest is like, peak lower than top of top of Everest might be considered as top of Everest.

- Hill Climbing Search initiates a loop that continuously moves in the direction of increasing value
- Terminates when it reaches a “Peak”
- Doesn’t maintain a search tree so the current node data structure needs only record the state and its objective function value.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state.
- Hill climbing is also called greedy local search sometimes because it grabs a good neighbor state without thinking ahead about where to go next
- One move is selected and all other nodes are rejected and are never considered.
- Halts if there is no successor.
- **Problem: depending on initial state, can get stuck in local maxima**



Algorithm

- determine successors of current state
- choose successor of maximum goodness (break ties randomly)
- if goodness of best successor is less than current state's goodness, stop
- otherwise make best successor the current state and go to step 1

Drawback of Hill Climbing Search

This simple policy has three well-known drawbacks:

a. Local Maxima: a local maximum as opposed to global maximum.
b. Plateaus: An area of the search space where evaluation function is flat, thus requiring random walk. Search might be unable to find its way of plateau.

c. Ridge: Where there are steep slopes and the search direction is not towards the top but towards the side.

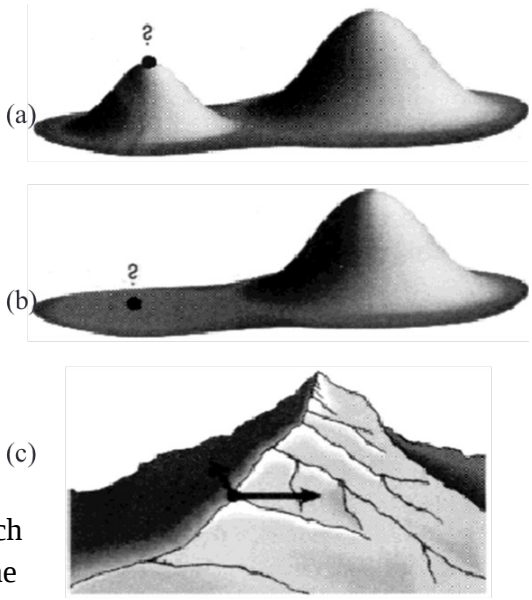


Figure 5.9 Local maxima, Plateaus and ridge situation for Hill Climbing

Solution

- Backtrack to some earlier node and try going to different direction (**for local maxima**)
- Make a big jump in some direction to try to get a new section of the search space (**for plateau**)
- Apply two or more rules such as bi-direction search before doing the test (**for ridge**)
    - Moving in several directions at once

Simulated Annealing Search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency
- Instead of restarting from a random point, we allow the search to take some downhill steps to try to escape local maxima.
- A random pick is made for the move
    - If it improves the situation, it is accepted straight away
    - If it worsens the situation, it is accepted with some probability less than 1 which decreases exponentially with the badness of the move i.e. for bad moves the probability is low and for comparatively less bad moves, it is higher.
- Probability of downward steps is controlled by **temperature parameter**.

- High temperature implies high chance of trying locally "bad" moves, allowing nondeterministic exploration.
- Low temperature makes search more deterministic (like hill-climbing).
- Temperature begins high and gradually decreases according to a predetermined annealing schedule.
- Initially we are willing to try out lots of possible paths, but over time we gradually settle in on the most promising path.
- If temperature is lowered slowly enough, an optimal solution will be found.
- In practice, this schedule is often too slow and we have to accept suboptimal solutions.

Applications of Simulated Annealing Search

- VLSI layout problem
- Factory scheduling
- Travelling salesman problem

Local beam search

- A path based algorithm
- Keeps track of *k* states rather than just one
- Starts with k randomly generated states
- At each iteration, all the successors of all k states are generated.
- If anyone is a goal state, stop, else select the *k* best successors from the complete list and repeat.

Adversarial Search

Competitive environments in which the agents goals are in conflict, give rise to **adversarial search**, often known as **games**.

In AI, games means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which utility values at the end of the game are always equal and opposite.

- Eg. **If first player wins, the other player necessarily loses**

Opposition between the agent's utility functions make the situation adversarial.

Game

A game can be formally be defined as a kind of search problem with the following components

- Initial state

- A successor function
- A terminal test
- A utility function

Minimax Algorithm

- It is a **recursive algorithm** for choosing the next move in a n-player game, **usually a two player game**
- A value is associated with each position or state of the game
- The value is computed by means of a **position evaluation function** and **it indicates how good it would be for a player to reach the position.**
- The player then makes the move that maximizes the minimum value of the position from the opponents possible moves called maximizing player and other player minimize the maximum value of the position called minimizing player.

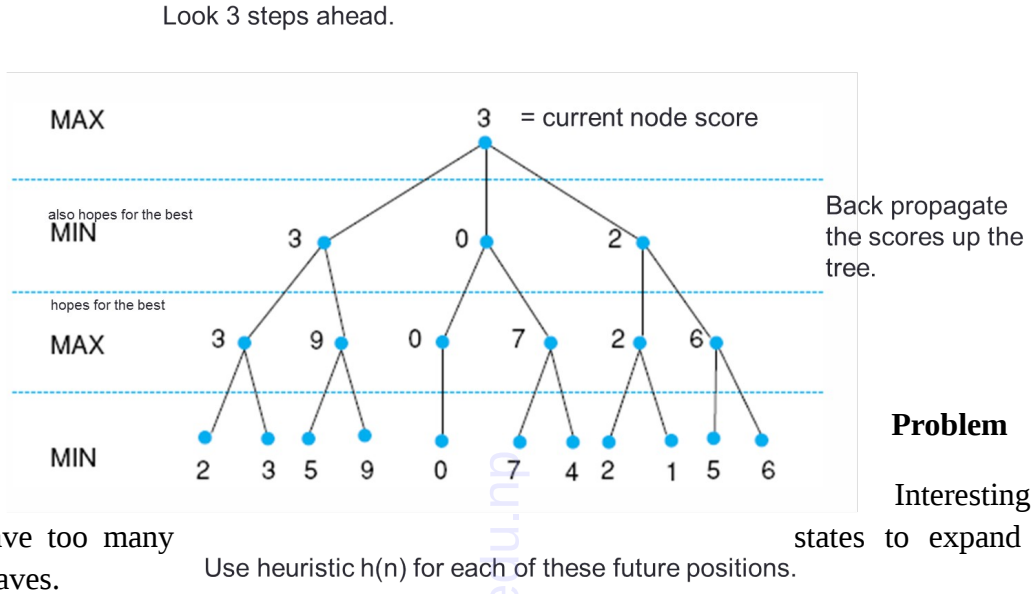
MiniMax Game Search

- It is a Depth-first search with limited depth.
- Use a static evaluation function for all leaf states.
- Assume the opponent will make the best move possible.

Algorithm

```
minimax(player,board)
    if(game over in current board position)
        return winner
    children = all legal moves for player from this board
    if(max's turn)
        return maximal score of calling minimax on all the children
    else (min's turn)
        return minimal score of calling minimax on all the children
```

example



Problem

- Interesting games have too many states to expand to the leaves.
- Number of nodes to expand is exponential in the depth of the tree and the branching factor.

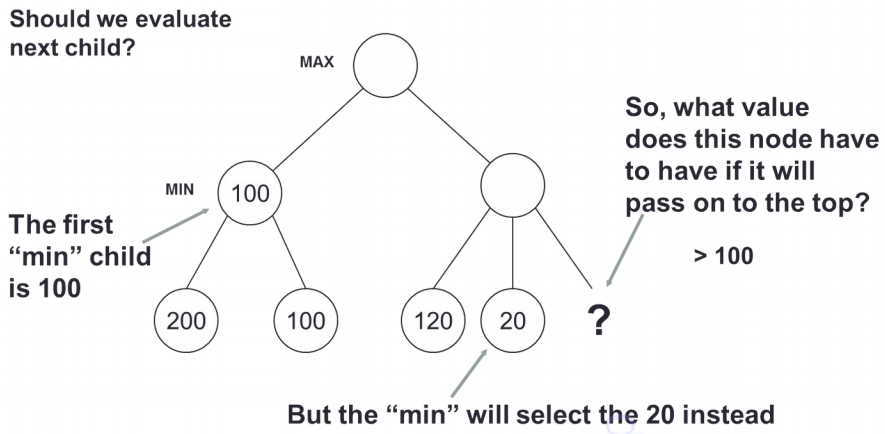
Properties of minimax

- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity? $O(b^m)$
- Space complexity? O(bm) (depth-first exploration)
- For chess, b ≈ 35, m ≈100 for "reasonable" games exact solution completely infeasible

α-β pruning

- **Minimax search explores some parts of tree it doesn't have to**
- To avoid this, minimax algorithm is modified with two values **alpha** and **beta**, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured respectively.
- Alpha-beta pruning algorithm **explores a branch only if there is possibility of producing superior alternatives**.
- It **reduces the time** required for the search and it must be restricted so that no time is to be wasted searching moves that are obviously bad for the current player.
- The exact implementation of alpha-beta keeps track of the best move for each side as it moves throughout the tree.
- It adapts minimax to consider the values of the nodes and whether exploration down a branch could possibly be fruitful.
- It proceeds from the idea "**If you have an idea that is surely bad, don't take the time to see how truly awful it is** (Pat Winston)"

Example



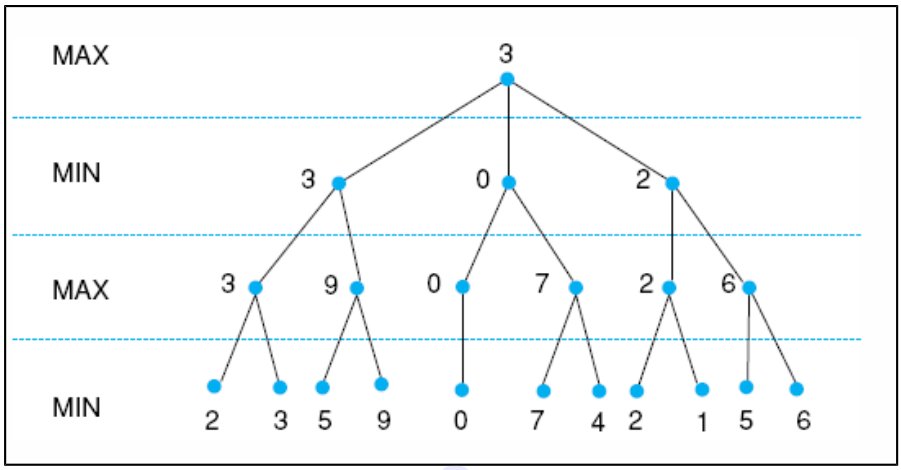
Alpha Beta Procedure

- At each non-leaf node, store two values— *alpha* and *beta*.
- Let *alpha* be the best (i.e., maximum) value found so far at a "max" node.
- Let *beta* be the best (i.e., minimum) value found so far at a "min" node.
- Initially assign *alpha* = –◊ and *beta* = ∞ at the root.
- Note *alpha* is monotonically non-decreasing and *beta* is monotonically non-increasing as you travel *up* the tree.
- Given a node *n*, cut off the search below that node (i.e., generate no more children) if
  - *n* is a "max" node and ***alpha*(*n*) ∞ *beta*(*i*)** for some "min" ancestor *i* of *n*, or
  - *n* is a "min" node and ***beta*(*n*) ≥ *alpha*(*j*)** for some "max" ancestor *j* of *n*.

Algorithm

```
function alphabeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        for each child of node
            α := max(α, alphabeta(child, depth - 1, α, β, FALSE))
            if β ≤ α
                break (* β cut-off *)
        return α
    else
        for each child of node
            β := min(β, alphabeta(child, depth - 1, α, β, TRUE))
            if β ≤ α
                break (* α cut-off *)
        return β
```

Example

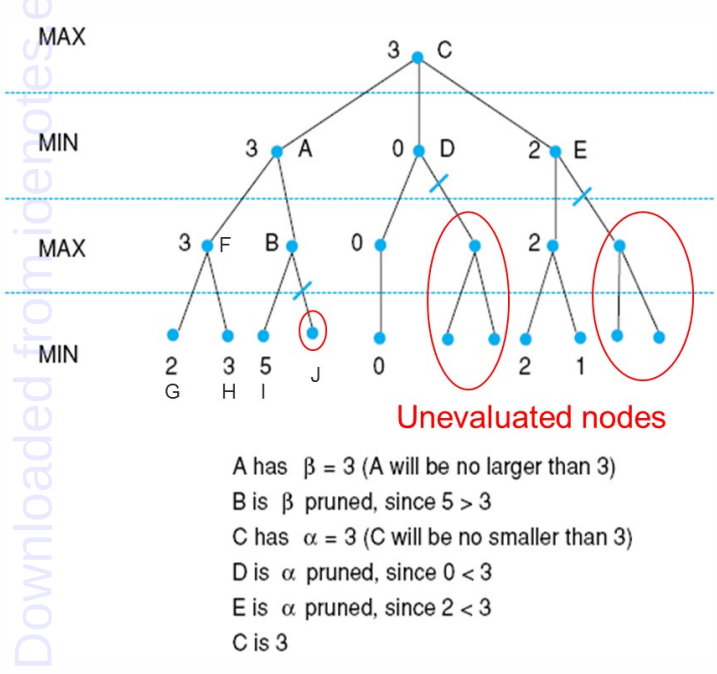


Minimax without pruning

Alpha-Beta run

- Depth-first search
  Visit C, A, F,
  Visit G, heuristics evaluates to 2
  Visit H, heuristics evaluates to 3
- Back up {2,3} to F. max(F)=3
- Back up to A. β(A)=3. Temporary min(A) is 3.
  3 is the ceiling for node A's score.
- Visit B according to depth-first order.
- Visit I. Evaluates to 5.
- Max(B)>=5. α(B)=5.
  It does not matter what the value of J is, min(A)=3. β-prune J.

Alpha-beta pruning improves search efficiency of minimax without sacrificing accuracy.



Effectiveness

- The effectiveness depends on the order in which children are visited.
- In the best case, the effective branching factor will be reduced from *b* to *sqrt(b)*.
- In an average case (random values of leaves) the branching factor is reduced to *b/logb*.

Properties of α-β

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering," time complexity = $O(b^{m/2})$
  □ doubles depth of search
- A simple example of the value of reasoning about which computations are relevant

Example