UNINFORMED SEARCH STRATEGIES

Search Strategies

- Searching is the process to find the solution for a given set of problems. This in AI can be done by using either uninformed searching strategies or informed searching strategies.
- Uninformed Search / Blind Search
- Informed Search

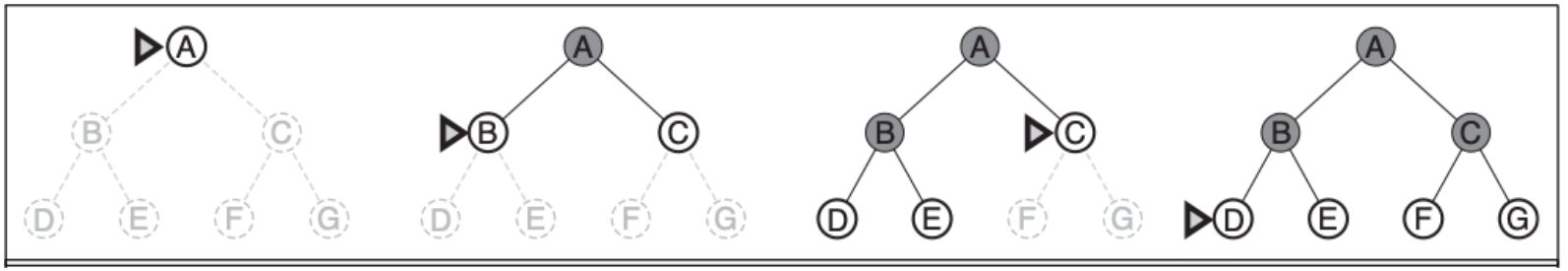
Breadth first search

- A basic technique of traversing a graph is breadth-first search or BFS.
- It first finds the shortest path always, however, this might also lead to using of more memory.
- The state space in form of a tree is used. The solution is obtained by traversing the tree.
- The nodes of the tree display the start value or starting state, various intermediate states and the final state.
- Queue data structure is used.
- It is level by level traversal.

Breadth First Search

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

```
node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
frontier ← a FIFO queue with node as the only element
explored ← an empty set
loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)   /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
            if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
            frontier ← INSERT(child, frontier)
```



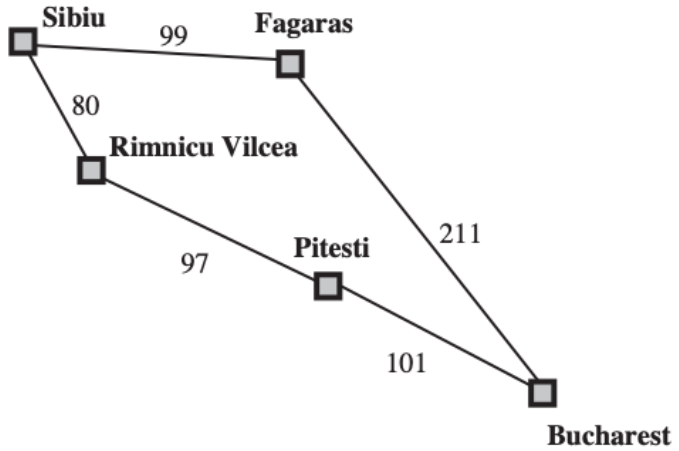
Time and Memory Requirements

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Uniform cost search

- It is also known as the Cheapest First Search.
- UCS modifies BFS by expanding lowest cost node on the fringe instead of the lowest deep node.
- It stores the frontier node with the least path cost in priority queue.
- If all the steps are equal UCS is identical to BFS because BFS always expands the shallowest unexpanded node.

Uniform cost search



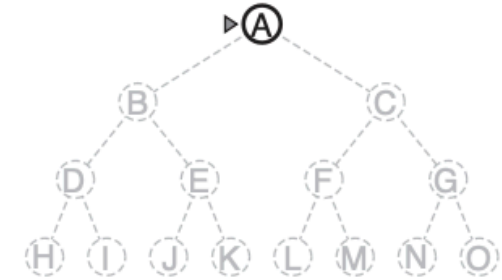
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

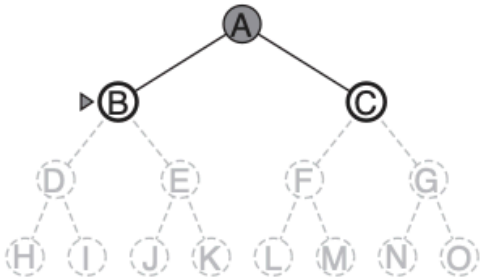
node ← a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with *node* as the only element
explored ← an empty set
loop do
  if EMPTY?(*frontier*) **then return** failure
  node ← POP(*frontier*) /* chooses the lowest-cost node in *frontier* */
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  add *node*.STATE to *explored*
  for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
    child ← CHILD-NODE(*problem*, *node*, *action*)
    if *child*.STATE is not in *explored* or *frontier* **then**
      frontier ← INSERT(*child*, *frontier*)
    else if *child*.STATE is in *frontier* with higher PATH-COST **then**
      replace that *frontier* node with *child*

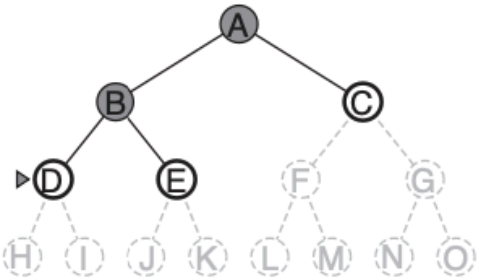
Depth first search

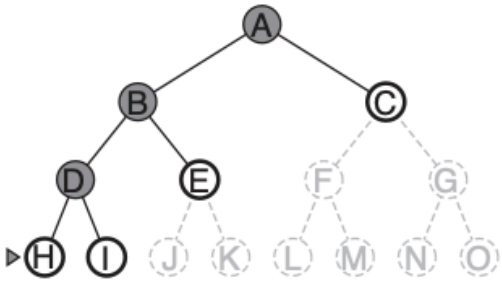
- Depth-first search always expands the deepest node in the current frontier of the search tree.
- Instead of probing the width, one branch of a tree can be explored until the solution is found.
- DFS always expands one of the nodes at the deepest level of the tree.
- The search is terminated if it reaches the leaf node, or encountering some previous state, or might be because the process becomes longer than the set time limit.
- If any of such situations is encountered, the process is terminated resulting in a backtracking.

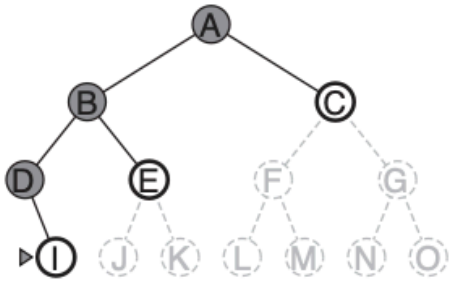
DFS

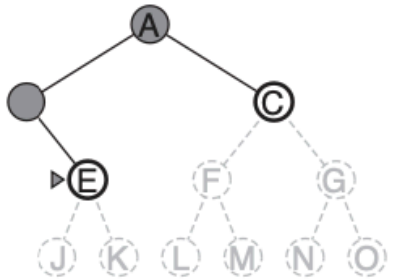

A
B C
D E F G
H I J K L M N O

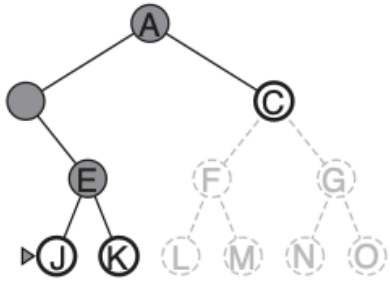

A
B C
D E F G
H I J K L M N O

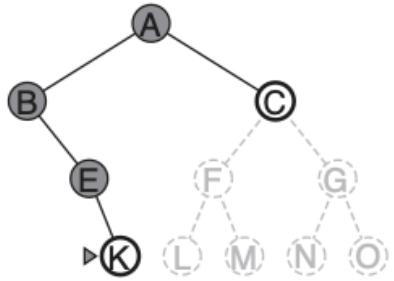

A
B C
D E F G
H I J K L M N O

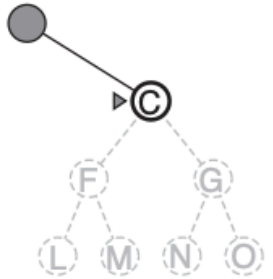

A
B C
D E F G
H I J K L M N O

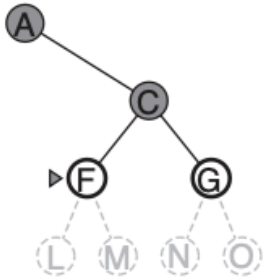

A
B C
D E F G
I J K L M N O

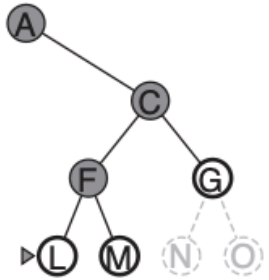

A
C
E F G
J K L M N O

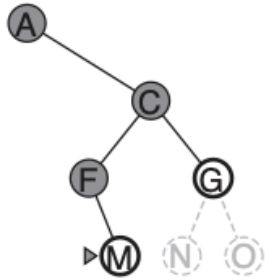

A
C
E F G
J K L M N O


A
B C
E F G
K L M N O


C
F G
L M N O


A
C
F G
L M N O


A
C
F G
L M N O


A
C
F G
M N O

Depth Limited Search

- The problem related to the unbounded tree can be reduced by providing the DFS with the predetermined depth limit of λ.
- The nodes at depth λ are considered to have no successor.
- This approach is called depth limited search and solves infinite path problem.
- Additional of incompleteness is introduced if we choose λ less than d (shallowest goal is beyond the depth limit).
- This search will also remain non optimal if λ is greater than d.
- DFS can be seen as the special case of depth limited search $\lambda=\infty$
- Three possible outcomes:
- Solution, Failure (no solution), Cutoff (no solution within cutoff)

Depth Limited Search

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Iterative Deeping DFS

- Iterative Deeping DFS is the strategy, most often used in combination with DFS which finds the best depth limit.
- This method is preferred when search space is large and depth of the solution is not known.
- This method repeatedly applies depth-limited search with the increasing limits by gradually increasing the limit from 0, then to 1 and then to 2 and so on.
- This happens when the depth limit reaches to ***d*** that is the depth of the shallowest goal node.
- Iterative deepening DFS combines the benefits of DFS and BFS. As compared to DFS, its memory requirements are modest.

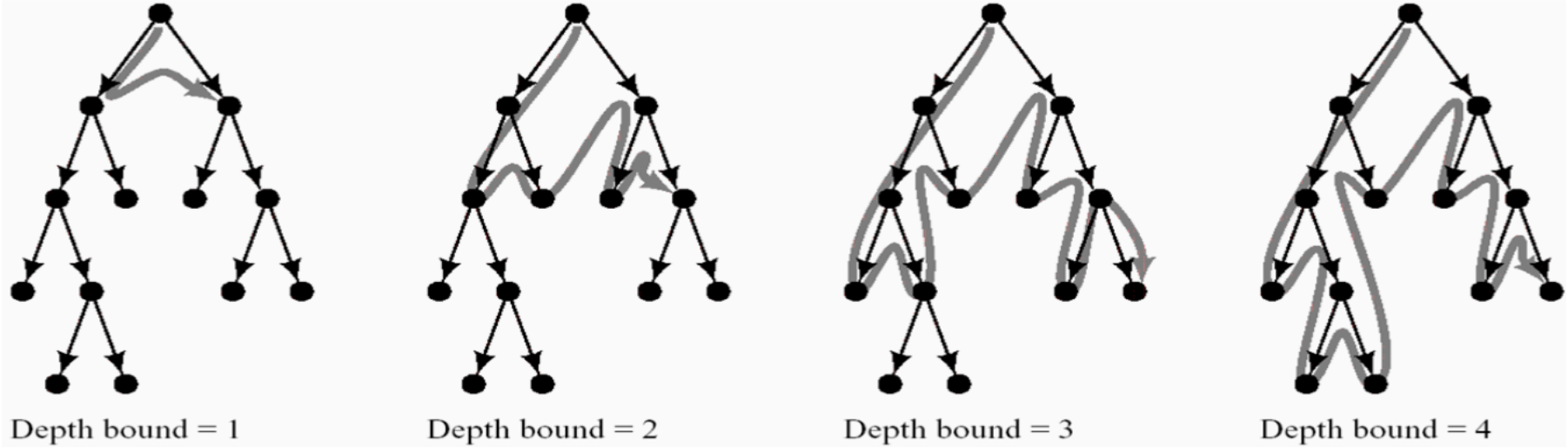
Iterative Deeping DFS

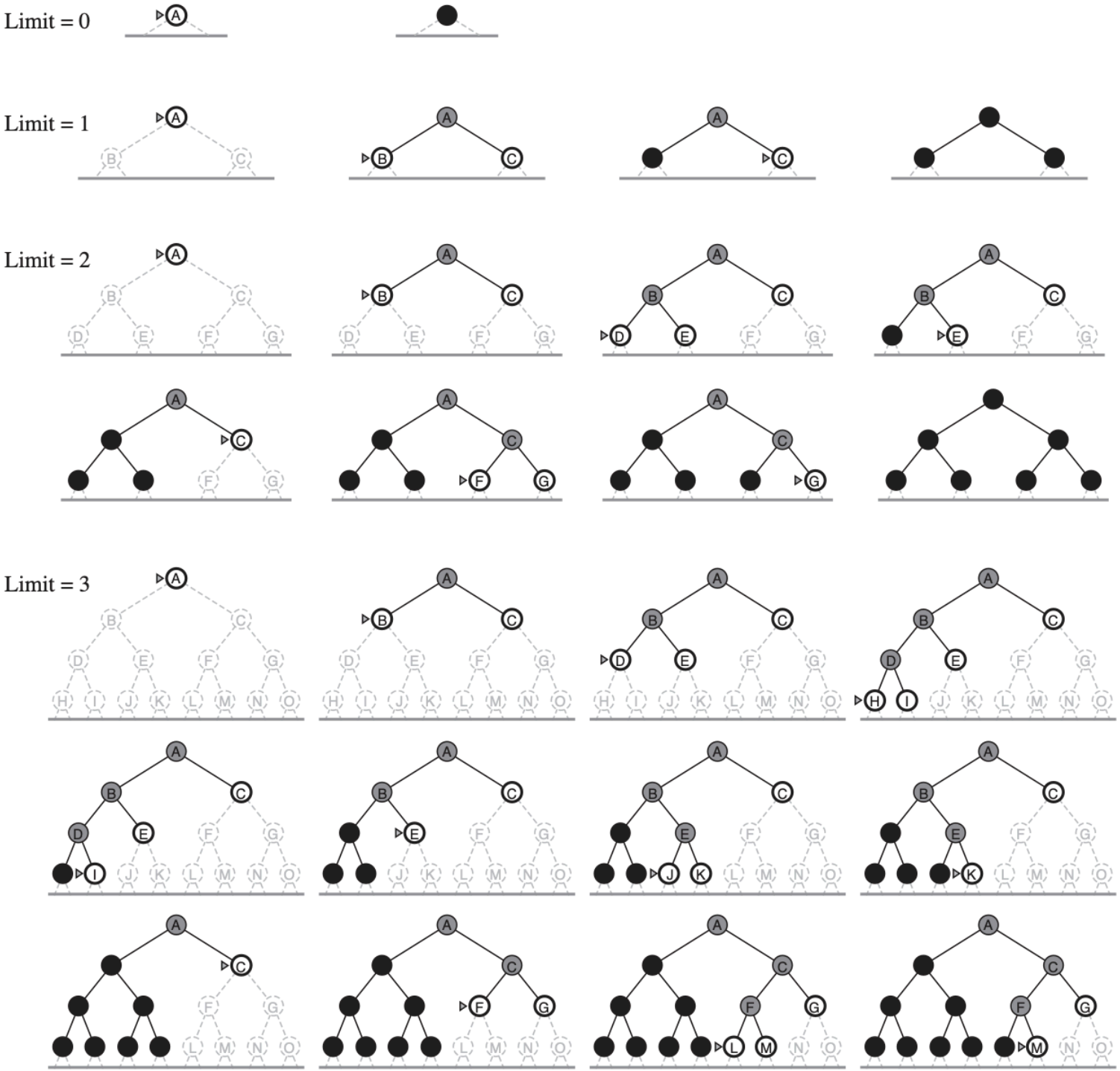
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure

  for *depth* = 0 **to** ∞ **do**

    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)

    if *result* $\neq$ cutoff **then return** *result*




Limit = 0
Limit = 1
Limit = 2
Limit = 3

Bidirectional Search

- Run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle.
- One should be able to generate predecessor states. Predecessors of node n are all the nodes that have n as successor.
- This idea works well when search meets in the middle.
- Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.
- For problem when branching factor is b in both direction, bidirectional search can make a big difference.

Comparing uninformed search strategies

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}

- Evaluation of tree-search strategies :
  - b is the branching factor.
  - d is the depth of the shallowest solution.
  - m is the maximum depth of the search tree.
  - l is the depth limit.
  - Superscript caveats are as follows: a complete if b is finite; b complete if step costs $\geq \varepsilon$ for positive ε; c optimal if step costs are all identical; d if both directions use breadth-first search.

Informed Search Strategies

Introduction - Heuristic

- Heuristic function - associate a value with every state in a given problem space thus helps to achieve goal oriented search.
- Strategies that know whether one goal state is more promising than the other are called *heuristic / informed search techniques.*
- This function will not always give optimal solution but will give solution in bounded time.

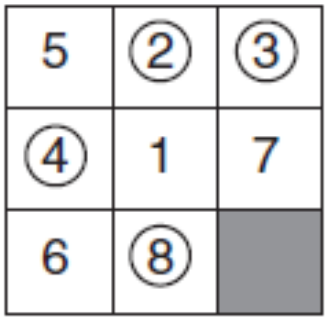
8 Puzzle Heuristics

Solution :

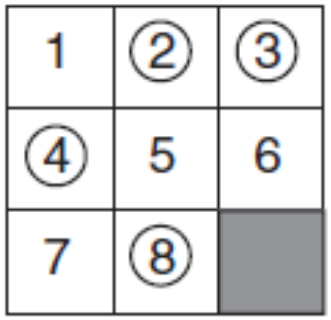
- The **heuristic function of the game may be as follows:** Count the number of tiles that are in place with respect to the goal state (marked in circle). Hence,

  h(Initial State) = 4
- Hence, the heuristic value for the initial state is 4 (Note, as a convention we do not consider the position of blank in the non goal state).

  h(Goal State) = 8
- Heuristic value for the goal state will always be eight as all the tiles are in correct place in the goal state.



I.S.



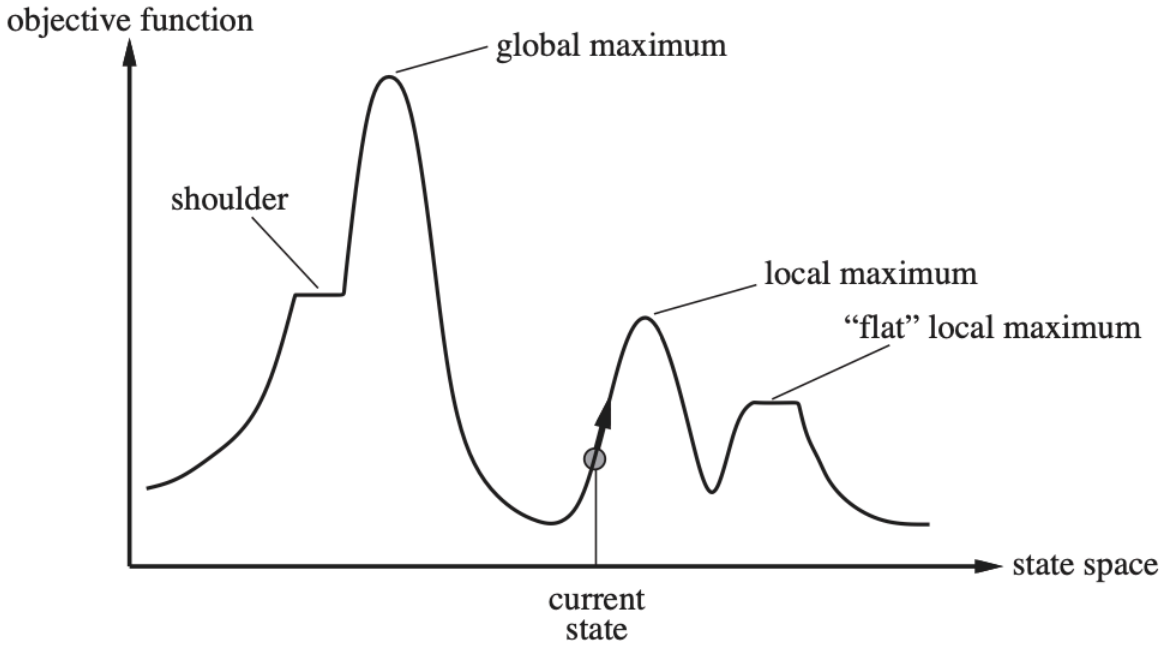
G.S.

Hill Climbing

- Hill climbing search algorithm moves in the direction of increasing value.
- It stops when it reaches a “peak” where no neighbour has higher value.
- It is considered to be one of the simplest procedures for implementing heuristic search.
- The hill climbing comes from the idea that if you are trying to find the top of the hill and you go up direction from wherever you are.
- This heuristic combines the advantages of both depth-first and breadth-first searches into a single method.
  - Simple Hill Climbing
  - Steepest-ascent Hill Climbing
  - Stochastic Hill Climbing

Different Regions in the State–Space Diagram

1. Current state
2. Local maximum
3. Global maximum
4. Plateau/Flat local maximum
5. Ridge
6. Shoulder



Greedy Best First Search

- It is graph search algorithm where node is selected for an expansion based on evaluation function f(n).
- Node with lowest evaluation is selected if it measures distance to goal. It is implemented via priority queue which maintain fringe in ascending order of value.
- Serves as combination of depth first and breadth first.
- Also referred to greedy algorithm as it quickly attacks most desirable path as soon as its heuristic weight becomes the most desirable.

Best First Search

- GBFS is more efficient than BFS and DFS. It allows us to switch between paths by gaining the benefit of both BFS & DFS.
- The solution can be found without computing all nodes.
- It does not get trapped in dead ends.
- If two states appear with the same heuristic values are at different levels and in different branches then they are not comparable using BFS.
- In case if two different state have the same heuristic value that choosing a state that is closer to the initial state would be better approach which is not adopted by GBFS.
- The heuristic value gives the distance of that state from the goal state, but BFS never consider the distance already travelled that is it never considers the distance of the state from the initial state.
- (e.g. Arad→Sibiu→Rimnicu→Virea→Pitesti→Bucharest is shorter)

Greedy best-first search

- Greedy best-first search expands nodes with minimal h(n).
- Is it complete?
- Is it optimal?
- Time and space complexity?
- The performance of heuristic search algorithms depends on the quality of the h(n) function.

A* Search

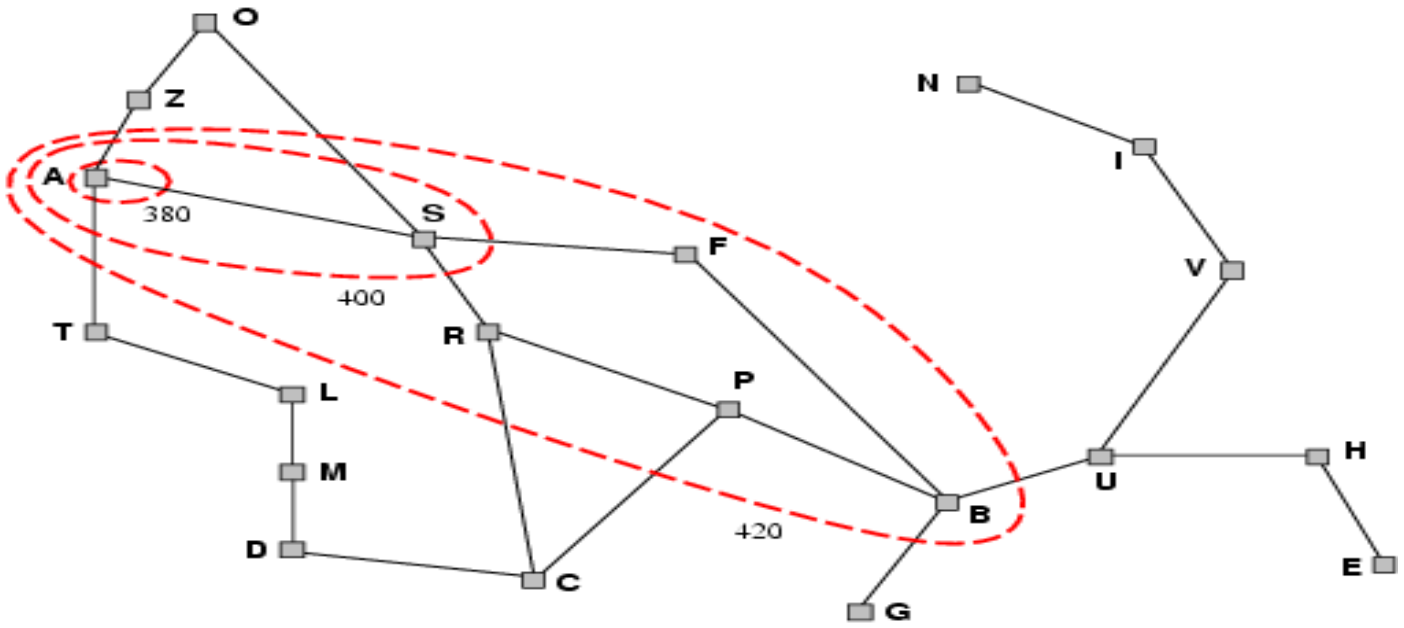
- One of the drawbacks of BFS is if two states appear with same heuristic value are at different levels and in different branches then they are not comparable using BFS.
- We can make them comparable by incorporating their distance from initial state along with heuristic value and choose the state closer to the root.
- 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

- The heuristic function is problem-specific. If the heuristic function is admissible, meaning that it does not overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.
- The estimated cost of the cheapest solution through n. Let h * (n) be the actual cost of the optimal path from n to the next goal. h is admissible if the following holds for all n : $h(n) \leq h^*(n)$
- We require that for A*, h is admissible. In other words, h is an optimistic estimate of the costs that actually occurs.

A* Search

- A* expands nodes in order of increasing f value.
- Gradually adds "f-contours" of nodes.
- Contour i contains all nodes with $f \leq f_i$ where $f_i < f_{i+1}$
- Map of Romania showing contours at f = 380, f = 400, and f = 420, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.



Advantages / Disadvantages

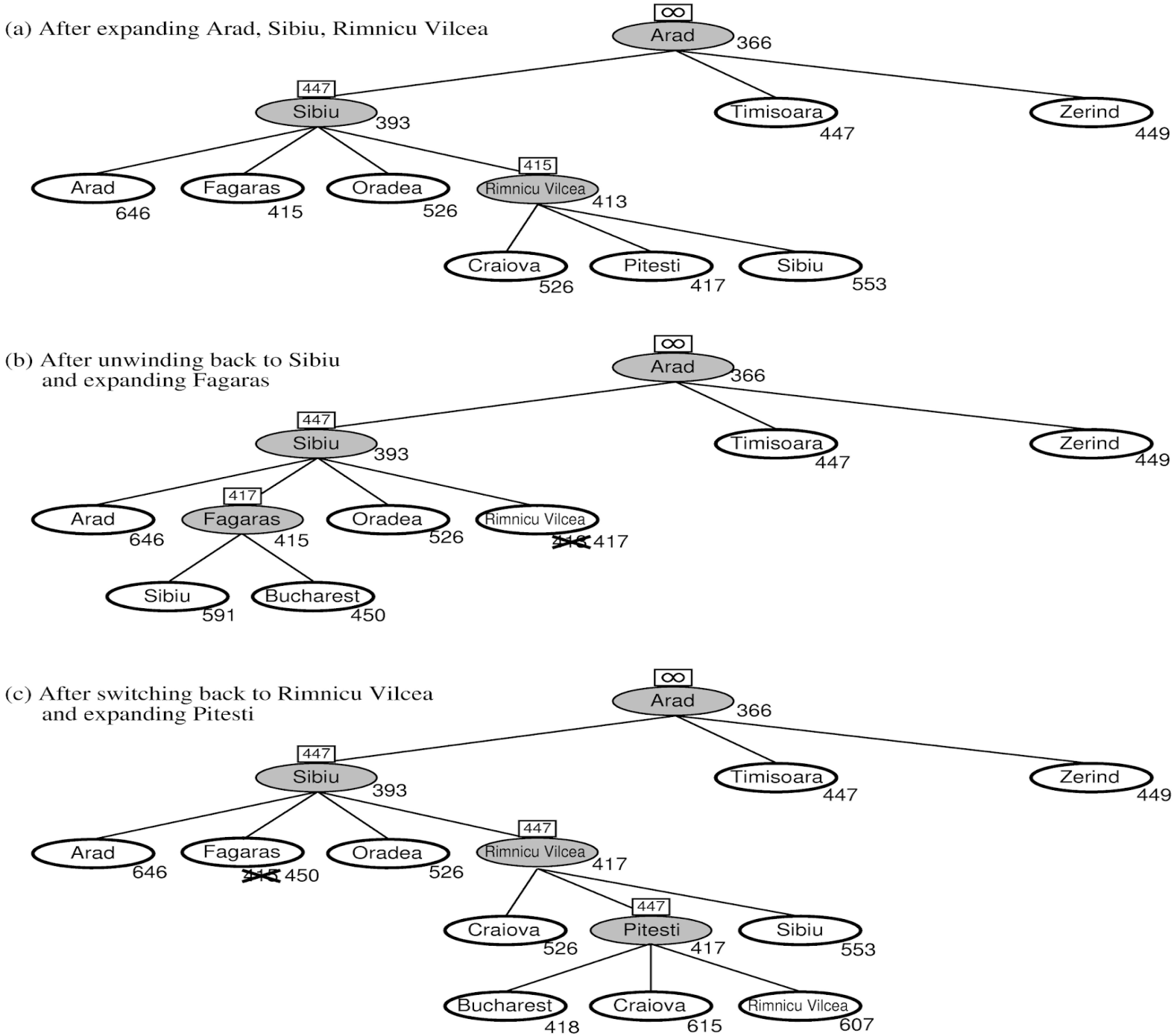
- It is Complete and optimal.
- It is efficient, i.e. no other optimal algorithm guaranteed to expand fewer nodes than A*
- Best one from other techniques.
- Used to solve very complex problem.
- A∗ is not practical for many large-scale problems as it runs out of space Because it keeps all generated nodes in memory
- Speed of execution is highly dependent on the accuracy of the heuristic algorithm that is used to compute h(n).

Iterative Deepening A* (IDA*)

- The key feature of the IDA* algorithm is that it doesn't keep track of each visited node which helps in saving memory consumption and can be used where memory is constrained.
- It is a variant of ID depth-first search that uses a heuristic function to evaluate the remaining cost to get to the goal from the A* search algorithm.
- Unlike A*, IDA* often ends up exploring the same nodes many times.
- Since it is a depth-first search algorithm, its memory usage is lower than in A*, but unlike ordinary ID search, it concentrates on exploring the most promising nodes and thus doesn't go to the same depth everywhere in the search tree.
- IDA* is a memory constrained version of A*.

Recursive BFS

- Another approach to solve the memory problem for A* search.
- RBFS mimic the operation of standard best-first search using only linear space.
- Similar to recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the ***f limit*** variable to keep track of the f-value of the best alternative path available from any ancestor of the current node.
- If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a backed-up value, the best f -value of its children.
- In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth.


(a) After expanding Arad, Sibiu, Rimnicu Vilcea
∞
Arad
366
447
Sibiu
393
Timisoara
447
Zerind
449
Arad
646
Fagaras
415
Oradea
526
415
Rimnicu Vilcea
413
Craiova
526
Pitesti
417
Sibiu
553
(b) After unwinding back to Sibiu and expanding Fagaras
∞
Arad
366
447
Sibiu
393
Timisoara
447
Zerind
449
Arad
646
417
Fagaras
415
Oradea
526
Rimnicu Vilcea
417
Sibiu
591
Bucharest
450
(c) After switching back to Rimnicu Vilcea and expanding Pitesti
∞
Arad
366
447
Sibiu
393
Timisoara
447
Zerind
449
Arad
646
Fagaras
450
Oradea
526
447
Rimnicu Vilcea
417
Craiova
526
447
Pitesti
417
Sibiu
553
Bucharest
418
Craiova
615
Rimnicu Vilcea
607

AND-OR Graph

- In a deterministic environment, the only branching is introduced by the agent's own choices in each state. We call these nodes OR nodes.
- In a nondeterministic environment, branching is introduced by the environment's choice of outcome for each action. We call these nodes AND nodes.
- AO* algorithm is a best first search algorithm. This algorithm uses the concept of AND-OR graphs to decompose any given complex problem into a set of smaller problems which are further solved.
- The decomposition of the problem or problem reduction generates AND arcs One AND may point to any number of successor nodes.
- A* algorithm cannot search AND-OR graphs efficiently.

AND-OR Graph

- The graph is known as AND-OR instead of AND Figure below shows an AND-OR graph.
- An algorithm to find a solution in an AND-OR graph must handle AND area appropriately.
- The formula for AO* algorithm is f(n) = g(n) + h(n)

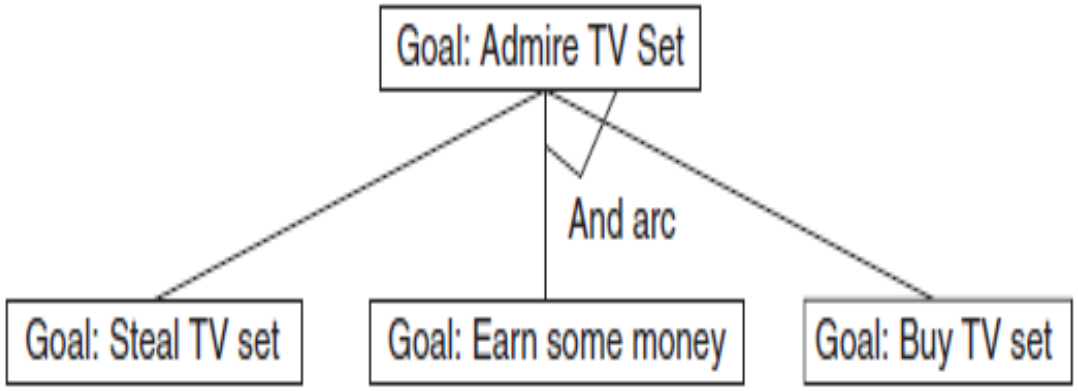


Figure shows AND-OR graph – an example

A* Vs AO*

A* algorithm :

1. It is used in pathfinding and graph traversal, also in the process of plotting an efficiently directed path between a number of nodes.
2. a* traverse the tree in depth and keep moving and adding up the total cost of reaching the cost from the current state to the goal state and add it to the cost of reaching the current state.

AO* algorithm :

1. AO* follow a similar procedure but there are constraints traversing specific paths.
2. When those paths are traversed, cost of all the paths which originate from the preceding node are added till that level, where you find the goal state regardless of the fact whether they take you to the goal state or not.

Simplified Memory Bounded A* (SMA*)

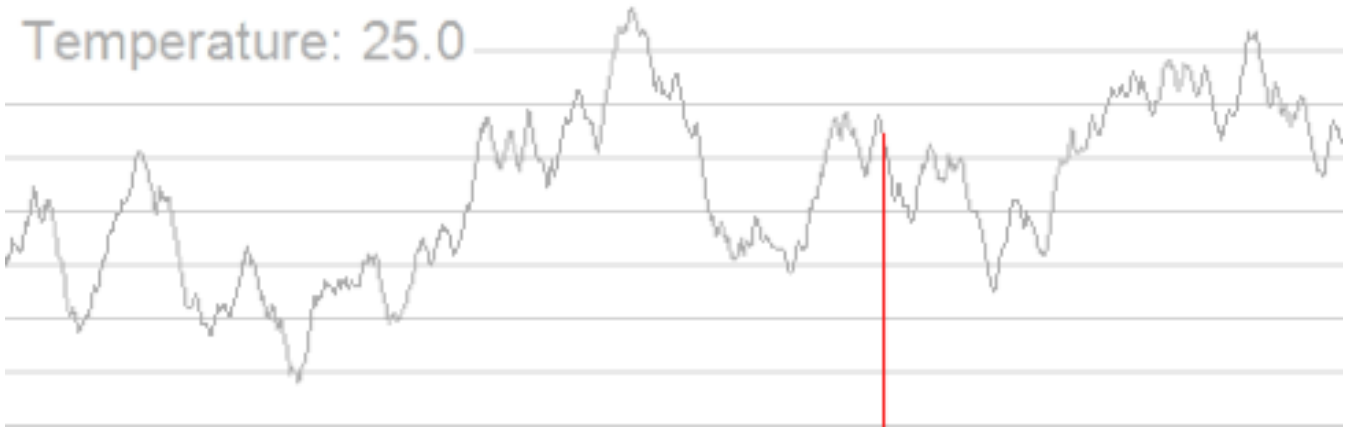
- SMA*, just like A* evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal : f(n) = g(n) + h(n).
- SMA* (Simplified Memory Bounded A*) is based on the A*, it expands the most promising branches according to the heuristic.
- The f value of the parent node will be updated to reflect changes to this estimate when its children are expanded.
- SMA* trims nodes whose expansion has revealed less promising than expected but remembers not just the f-cost limit, but as many nodes as the available memory affords.

Simulated Annealing Search

- Difficult optimisation problem : chip floor planning
- A naive algorithm would be a complete space search until we find the minimum.
- This may suffice for functions of few variables, but not for the computationally difficult problems.
- Simulated annealing is based on metallurgical practices by which a material is heated to a high temperature and cooled.
- At high temperatures, atoms may shift unpredictably, often eliminating impurities as the material cools into a pure crystal.
- This is replicated via the simulated annealing optimization algorithm, with energy state corresponding to current solution.

Simulated Annealing Search

- Let $s = s_0$
- For $k = 0$ through $k_{\max}$ (exclusive):
  - $T \leftarrow$ temperature(1 - $(k+1)/k_{\max}$)
  - Pick a random neighbour, $s_{\text{new}} \leftarrow$ neighbour(s)
  - If $P(E(s), E(s_{\text{new}}), T) \geq$ random(0, 1):
    - $s \leftarrow s_{\text{new}}$
- Output: the final state s



Local Beam Search

- Local Beam Search uses breadth-first search to build its search tree.
- At each level of the tree, it generates all successors of the states at the current level, sorting them in increasing order of heuristic cost.
- However, it only stores a predetermined number ß of best states at each level (called the beam width).
- Only those states are expanded next. The greater the beam width, the fewer states are pruned. With an infinite beam width, no states are pruned and beam search is identical to breadth first search.
- The beam width bounds the memory required to perform the search.

Branch and Bound Search

- Branch and bound is a method for solving optimisation problems by breaking them down into smaller sub-problems and using a bounding function to eliminate sub-problems that cannot contain the optimal solution (eg knapsack problem to optimise weight and value).
- These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.
- The Branch and Bound Algorithm technique solves these problems relatively quickly by keeping only 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.
- The usual technique for eliminating the subtrees from the search tree is known as pruning.
- Branch and Bound algorithm, uses stack data structure.

Other Methods

- Self Study topics
  - Genetic Algorithms
  - Searching With Partial Observations
  - Online Search Agents And Unknown Environments