Marwadi University

State Space, Heuristic, and Random Search

Department of CE

Unit no 2: State Space, Heuristic, and Random Search

DR. ANJALI DIWAN

Problems with Symbolic AI Approaches

Scalability

- It can take dozens or more man-years to create a useful systems
- It is often the case that systems perform well up to a certain threshold of knowledge (approx. 10,000 rules), after which performance (accuracy and efficiency) degrade

Brittleness

- Most symbolic AI systems are programmed to solve a specific problem, move away from that domain area and the system's accuracy drops rapidly rather than achieving a graceful degradation
  - this is often attributed to lack of common sense, but in truth, it is a lack of any knowledge outside of the domain area
- No or little capacity to learn, so performance (accuracy) is static

Lack of real-time performance

Problems with Connectionist AI Approaches

No “memory” or sense of temporality
- The first problem can be solved to some extent
- The second problem arises because of a fixed sized input but leads to poor performance in areas like speech recognition

Learning is problematic
- Learning times can greatly vary
- Overtraining leads to a system that only performs well on the training set and undertraining leads to a system that has not generalized

No explicit knowledge-base
- So there is no way to tell what a system truly knows or how it knows something

No capacity to explain its output
- Explanation is often useful in an AI system so that the user can trust the system’s answer

So What Does AI Do?

Most AI research has fallen into one of two categories

- Select a specific problem to solve
  - study the problem (perhaps how humans solve it)
  - come up with the proper representation for any knowledge needed to solve the problem
  - acquire and codify that knowledge
  - build a problem solving system
- Select a category of problem or cognitive activity (e.g., learning, natural language understanding)
  - theorize a way to solve the given problem
  - build systems based on the model behind your theory as experiments
  - modify as needed

Both approaches require

- one or more representational forms for the knowledge
- some way to select proper knowledge, that is, search

Problem, Problem Space and Search

In AI we want to build a system to solve a particular problem.

We need four things for this:

- Define the problem precisely.
  - It included precise specification of what the initial situation will be.
  - What final situation constitute acceptable solution to the problem.
- Analyze the problem.
- Isolate and represent the task knowledge that is necessary to solve the problem.
- Choose the best problem solving technique and apply it to a particular problem.

Problem, Problem Space and Search… Cont.

There are two ways by which AI problem can be represented.

1. State space representation
2. Problem Reduction

State Space

State space representation consists of defining an Initial state (from where to Start), the Goal State (the destination) and then follow certain set of sequence of steps (Called States)

- ❑State: AI problems can be represented as well formed set of all possible state. State can be Initial state, Goal state and various other possible state between them.
- ❑Space: In AI problem the exhaustive search of all possible states is called Space.

Search: Search is a technique which takes the initial state to the Goal state by applying certain set of valid rules while moving through space of all possible state

We can say that to do the search process we need the following:

1. Initial State
2. Set of valid rule
3. Goal State

Defining the problem as a state space search

A **state** is a representation of problem elements at a given moment.

A State space is the set of all states reachable from the initial state.

A state space forms a graph in which the nodes are states and the arcs between nodes are actions.

In the state space, a path is a sequence of states connected by a sequence of actions.

The solution of a problem is part of the graph formed by the state space.

The state space representation forms the basis of most of the AI methods.

Defining the problem as a state space search .. Cont.

Its structure corresponds to the structure of problem solving in two important ways:

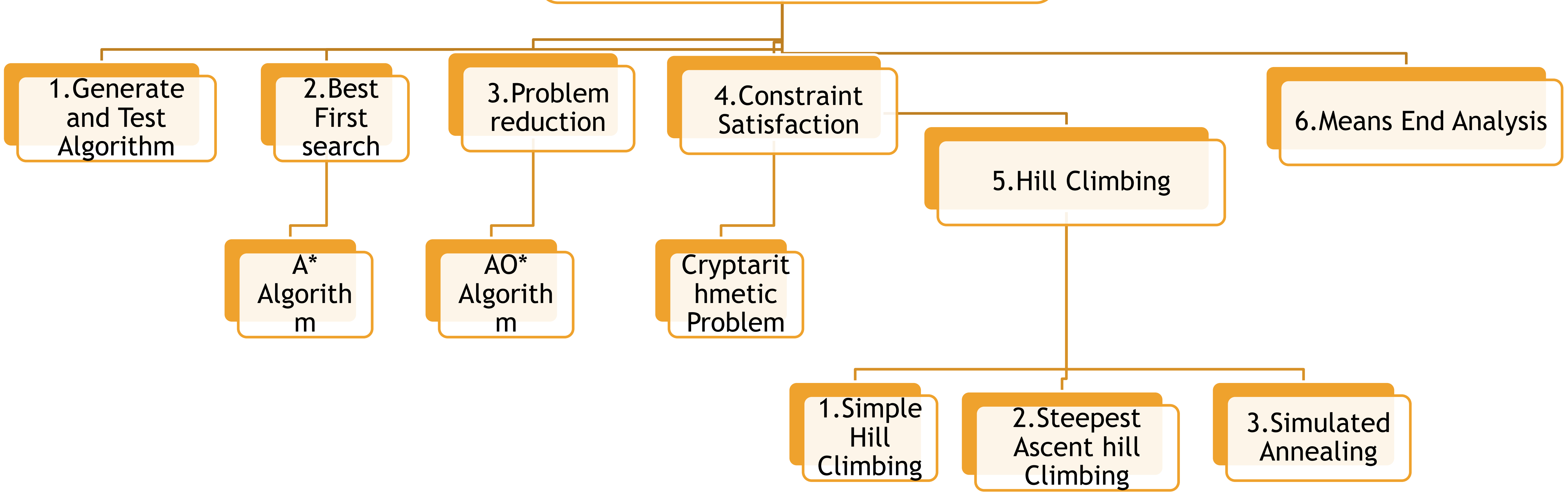
- It allows for a formal definition of a problem as per the need to convert some given situation into some desired situation using a set of permissible operations.
- It permits the problem to be solved with the help of known techniques and control strategies to move through the problem space until goal state is found.

Search Algorithms and Representations

1. Breadth-first
2. Depth-first
3. Best-first (Heuristic Search)
4. A*
5. Hill Climbing
6. Limiting the number of Plies
7. Minimax
8. Alpha-Beta Pruning
9. Adding Constraints
10. Genetic Algorithms
11. Forward vs Backward Chaining

- ✓ We will study various forms of representation and uncertainty handling in the next class period
- ✓ Knowledge needs to be represented
  - ✓ Production systems of some form are very common
    - ✓ If-then rules
    - ✓ Predicate calculus rules
    - ✓ Operators
  - ✓ Other general forms include semantic networks, frames, scripts
  - ✓ Knowledge groups
  - ✓ Models, cases
  - ✓ Agents
  - ✓ Ontologies

Heuristic Search Strategies/Informed Search



There are three important AI techniques:

Search — Provides a way of solving problems for which no direct approach is available. It also provides a framework into which any direct techniques that are available can be embedded.

Use of knowledge — Provides a way of solving complex problems by exploiting the structure of the objects that are involved.

Abstraction — Provides a way of separating important features and variations from many unimportant ones that would otherwise overwhelm any process.

Searching the State Space

The set of all possible sequences of legal moves form a *tree:*

- The *nodes* of the tree are labelled with states (the same state could label many different nodes).
- The initial state is the *root* of the tree.
- For each of the legal follow-up moves of a given state, any node labelled with that state will have a *child* labelled with the follow-up state.
- Each *branch* corresponds to a sequence of states (and thereby also a sequence of moves).
- There are, at least, two ways of moving through such a tree:
- *depth-first* and *breadth-first* search . . .

Search and Optimisation Problems

All these problems have got a common structure:

- We are faced with an initial situation and we would like to achieve a certain goal.
- At any point in time we have different simple actions available to us (e.g. “turn left” vs. “turn right”). Executing a particular sequence of such actions may or may not achieve the goal.
- *Search* is the process of inspecting several such sequences and choosing one that achieves the goal.
- For some applications, each sequence of actions may be associated with a certain cost. A search problem where we aim not only at reaching our goal but also at doing so at minimal cost is an *optimisation* problem.

Visualizing Search Space as Tree

- States are nodes
- Actions are edges
- Initial state is root
- Solution is path from root to goal node
- Edges sometimes have associated costs
- States resulting from operator are children.

Directed Graphs: A graph is also a set of nodes connected by links but where loops are allowed and a node can have multiple parents.
We have two kinds of graphs to deal with directed graph, where the links have direction (one-way streets).
Undirected Graphs: undirected graphs where the links go both ways. You can think of an undirected graph as shorthand for a graph with directed links going each way between connected nodes**.**

Searching for solutions (Graphs and trees)

The map of all paths within a state-space is a graph of nodes which are connected by links.

Now if we trace out all possible paths through the graph, and terminate paths before they return to nodes already visited on that path, we produce a search tree. Like graphs, trees have nodes, but they are linked by branches.

The start node is called the root and nodes at the other ends are leaves.

Nodes have generations of descendants.

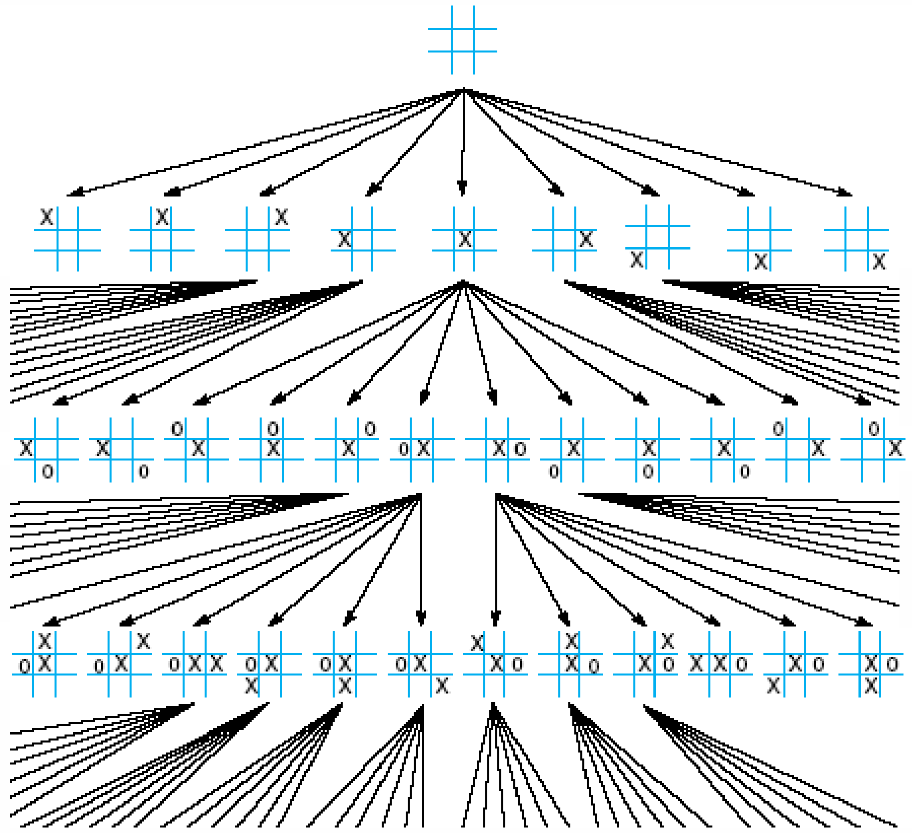
The aim of search is not to produce complete physical trees in memory, but rather explore as little of the virtual tree looking for root-goal paths.

What is Search?

We define the state of the problem being solved as the values of the active variables

- This will include any partial solutions, previous conclusions, user answers to questions, etc

- While humans are often able to make intuitive leaps, or recall solutions with little thought, the computer must search through various combinations to find a solution
- To the right is a search space for a tic-tac-toe game



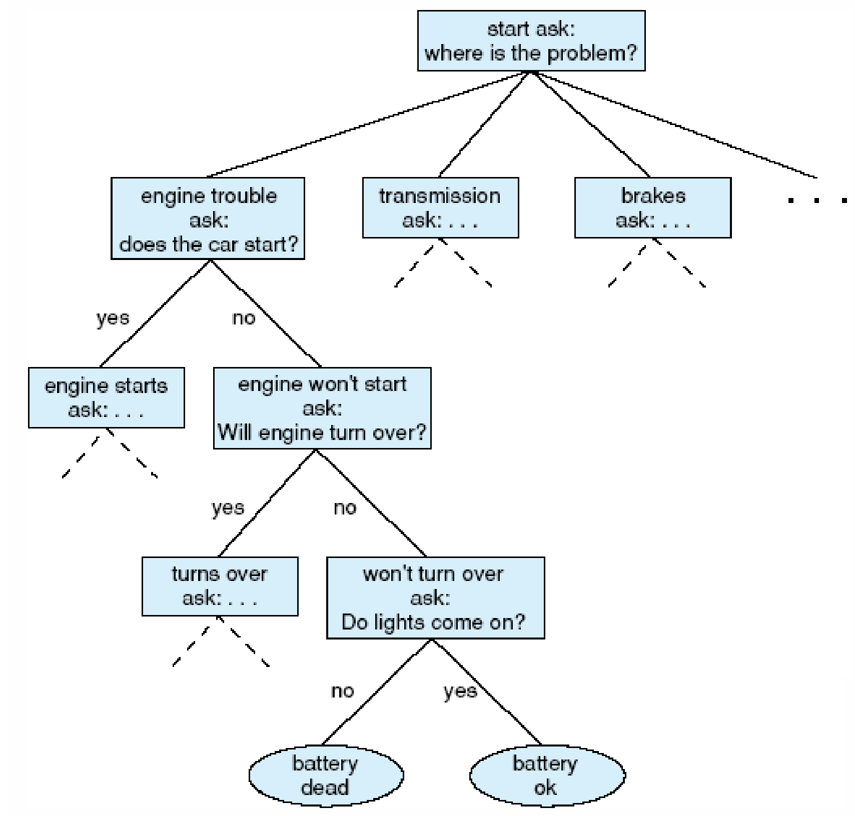
Search Spaces and Types of Search

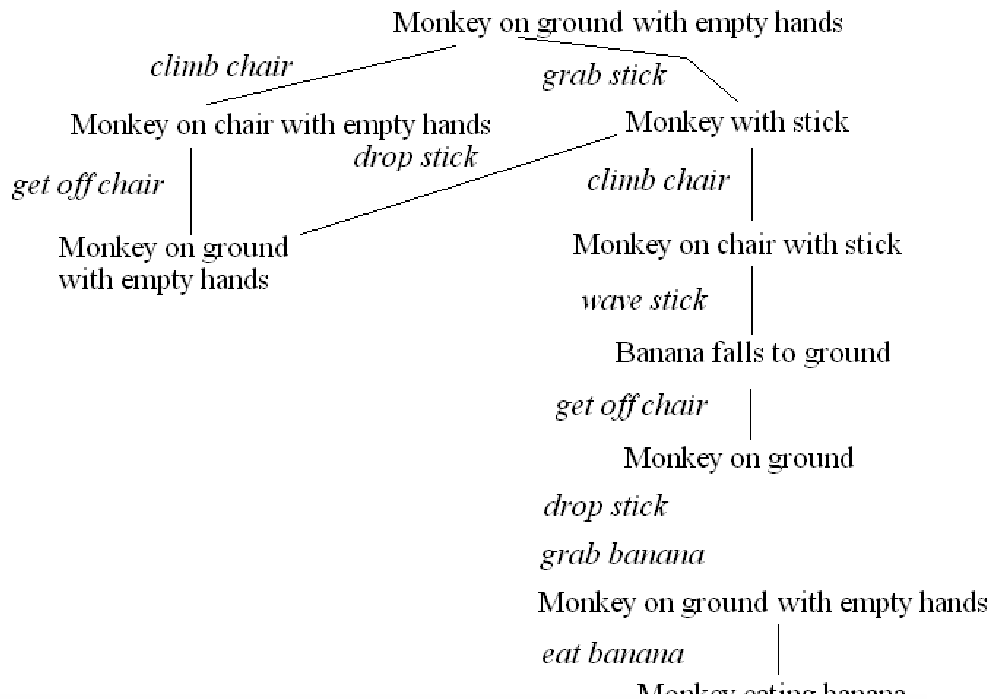
The search space consists of all possible states of the problem as it is being solved

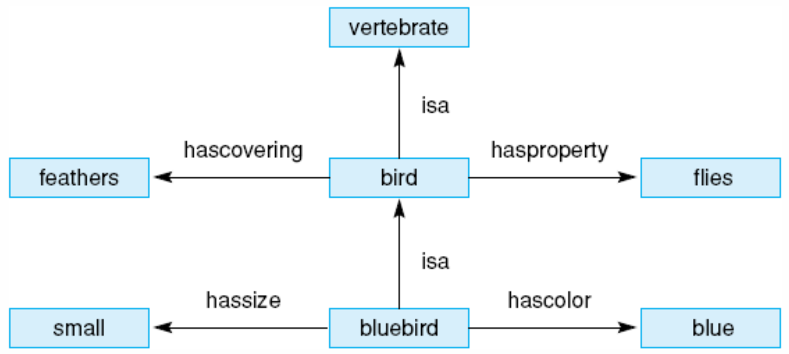
- A search space is often viewed as a tree and can very well consist of an exponential number of nodes making the search process intractable
- Search spaces might be pre-enumerated or generated during the search process
- Some search algorithms may search the entire space until a solution is found, others will only search parts of the space, possibly selecting where to search through a heuristic

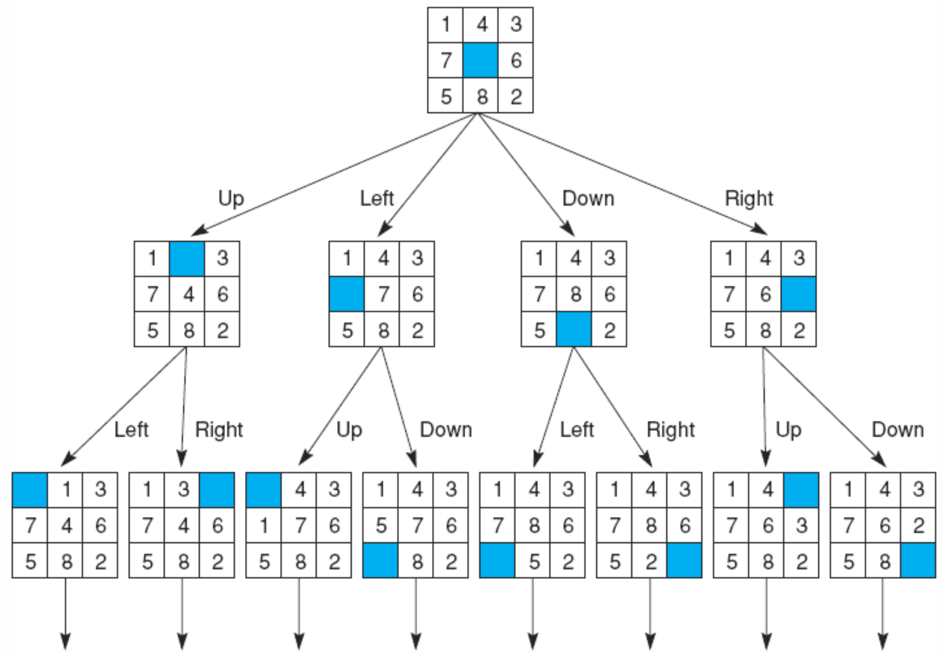
Search spaces include

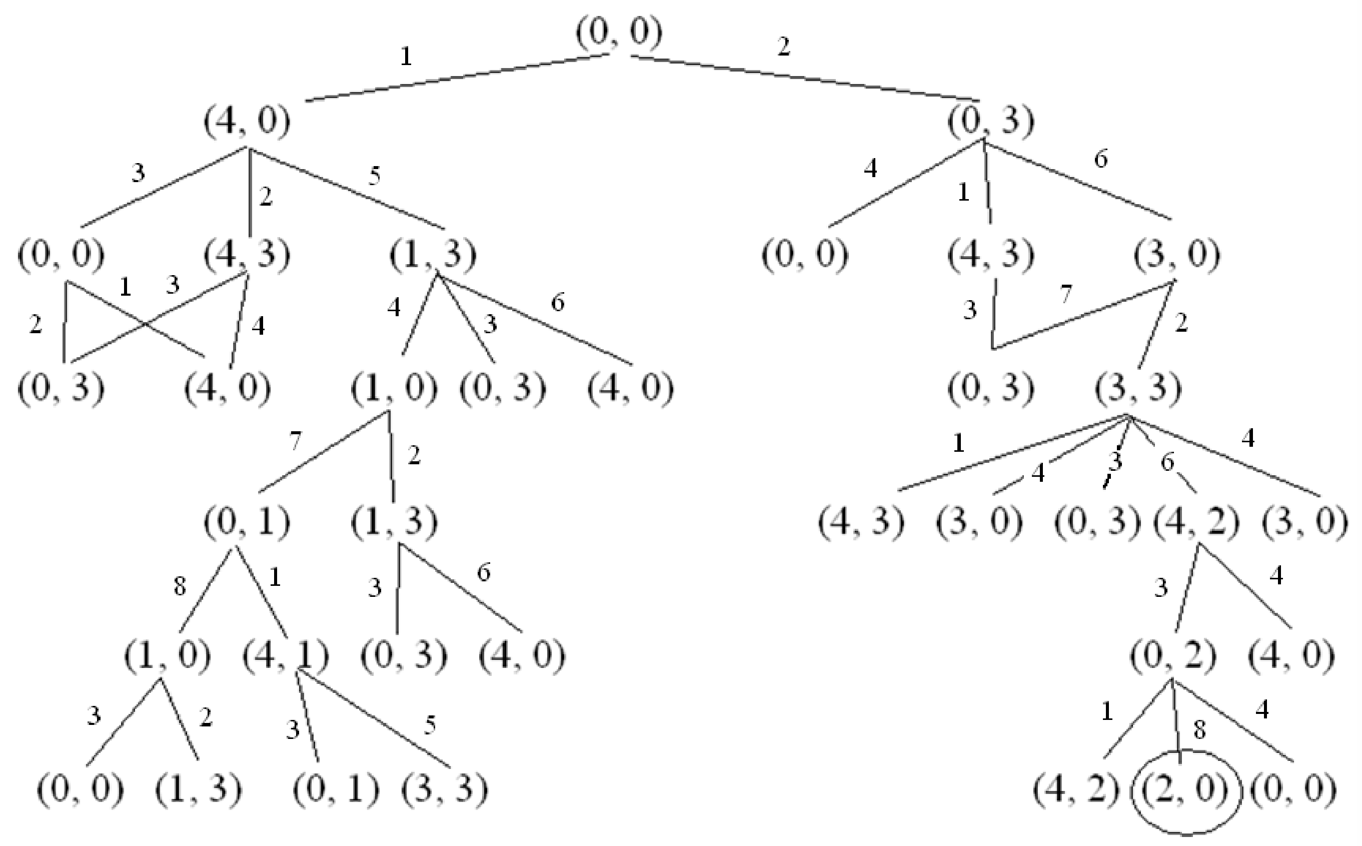
- Game trees like the tic-tac-toe game
- Decision trees (see next slides)
- Combinations of rules to select in a production system
- Networks of various forms (see next slides)
- Other types of spaces


start ask: where is the problem?
engine trouble ask: does the car start?
transmission ask: . . .
brakes ask: . . .
. . .
yes
no
engine starts ask: . . .
engine won't start ask: Will engine turn over?
yes
no
turns over ask: . . .
won't turn over ask: Do lights come on?
no
yes
battery dead
battery ok


Monkey on ground with empty hands
climb chair
grab stick
Monkey on chair with empty hands
Monkey with stick
drop stick
get off chair
climb chair
Monkey on ground with empty hands
Monkey on chair with stick
wave stick
Banana falls to ground
get off chair
Monkey on ground
drop stick
grab banana
Monkey on ground with empty hands
eat banana
Monkey eating banana


vertebrate
isa
feathers
hascovering
bird
hasproperty
flies
isa
small
hassize
bluebird
hascolor
blue


Up
Left
Down
Right
Left
Right
Up
Down
Left
Right
Up
Down



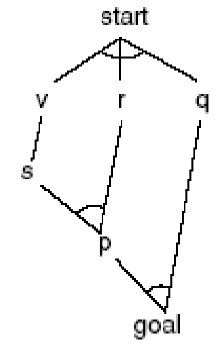
Production set:

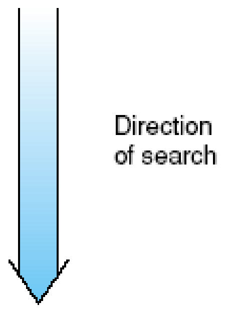
1. p ∧ q → goal
2. r ∧ s → p
3. w ∧ r → q
4. t ∧ u → q
5. v → s
6. start → v ∧ r ∧ q

Trace of execution:

Iteration #	Working memory	Conflict set	Rule fired
0	start	6	6
1	start, v, r, q	6, 5	5
2	start, v, r, q, s	6, 5, 2	2
3	start, v, r, q, s, p	6, 5, 2, 1	1
4	start, v, r, q, s, p, goal	6, 5, 2, 1	halt

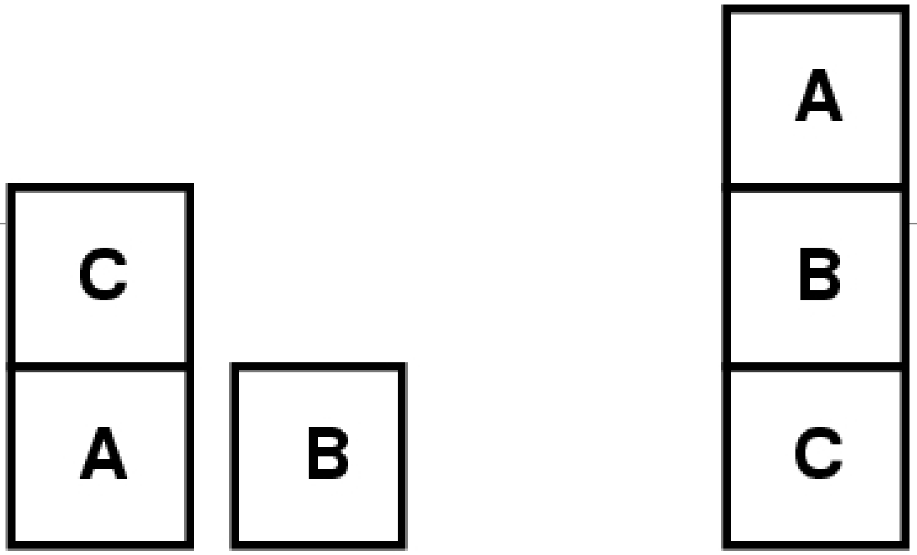
Space searched by execution:





Planning in the Blocks World

How can we get from the situation depicted on the left to the situation shown on the right?



Search and Optimisation Problems

All these problems have got a common structure:

•We are faced with an initial situation and we would like to achieve a certain goal.

•At any point in time we have different simple actions available to us (e.g. “turn left” vs. “turn right”). Executing a particular sequence of such actions may or may not achieve the goal.

•*Search* is the process of inspecting several such sequences and choosing one that achieves the goal.

•For some applications, each sequence of actions may be associated with a certain cost. A search problem where we aim not only at reaching our goal but also at doing so at minimal cost is an *optimisation* problem.

The State-Space Representation

- *State space:* What are the possible states? Examples:
  - Route planning: position on the map
  - Blocks World: configuration of blocks

A concrete problem must also specify the *initial state*.

- *Moves:* What are legal moves between states? Examples:
  - Turning 45° to the right could be a legal move for a robot.
  - Putting block *A* on top of block *B* is *not* a legal move if block *C* is currently on top of *A*.
- *Goal state:* When have we found a solution? Example:
  - Route planning: position = “Plantage Muidergracht 24”
- *Cost function:* How costly is a given move? Example:
  - Route planning: The cost of moving from position *X* to position *Y* could be the distance between the two.

Representation

For now, we are going to ignore the cost of moving from one node to the next; that is, we are going to deal with pure search problems.

A *problem specification* has to include the following:

The representation of states/nodes is problem-specific. In the simplest case, a state will simply be represented by its name .

move(+State,-NextState).

Given the current State, instantiate the variable NextState with a possible follow-up state (and all possible follow-up states through backtracking).

goal(+State).

Succeed if State represents a goal state.

Searching the State Space

The set of all possible sequences of legal moves form a *tree:*

The *nodes* of the tree are labelled with states (the same state could label many different nodes).

The initial state is the *root* of the tree.

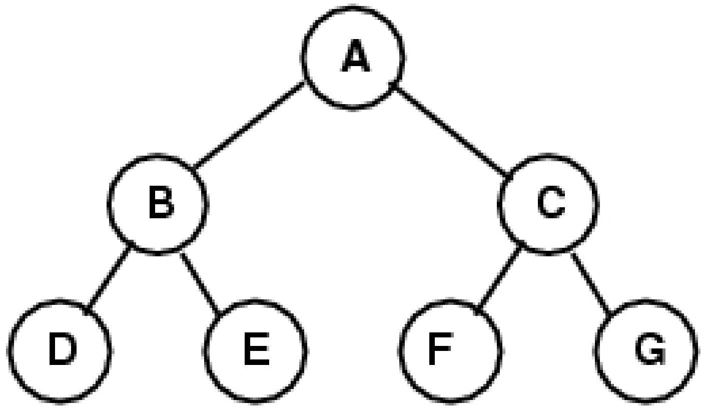
For each of the legal follow-up moves of a given state, any node labelled with that state will have a *child* labelled with the follow-up state.

Each *branch* corresponds to a sequence of states (and thereby also a sequence of moves).

There are, at least, two ways of moving through such a tree:
depth-first and *breadth-first* search . . .

Depth-first Search

In depth-first search, we start with the root node and completely explore the descendants of a node before exploring its siblings (and siblings are explored in a left-to-right fashion).



Depth-first traversal: A *!* B *!* D *!* E *!* C *!* F *!* G

Implementing depth-first search in Prolog is very easy, because Prolog itself uses depth-first search during backtracking.

Testing: Blocks World

It’s working pretty well for some problem instances . . .

```
?- solve_depthfirst([[c,b,a],[],[]], Plan).
Plan = [[[c,b,a],     [],          []],
        [[b,a],       [c],         []],
        [[a],         [b,c],       []],
        [[],          [a,b,c],     []]]
Yes
```

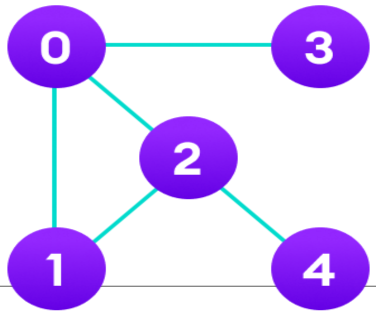
. . . but not for others . . .

```
?- solve_depthfirst([[c,a],[b],[]], Plan).  ERROR:
Out of local stack
```

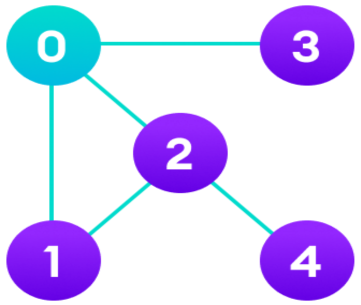
STEPS DFS

The step by step process to implement the DFS traversal is given as follows -

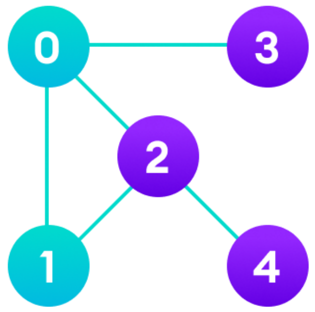
1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.



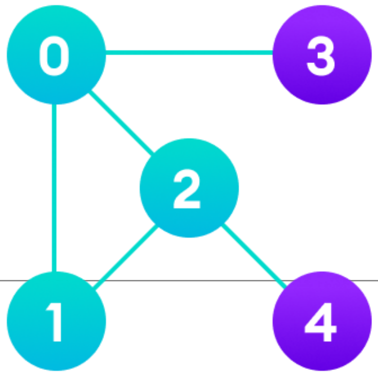
					Visited
					Stack



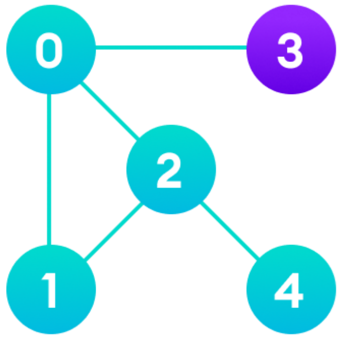
0					Visited
1	2	3			Stack



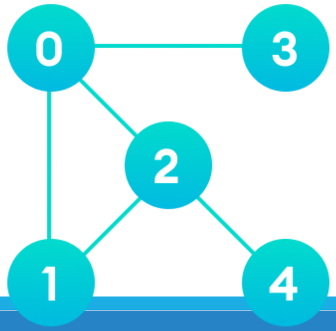
0	1				Visited
2	3				Stack



0	1	2			Visited
4	3				Stack

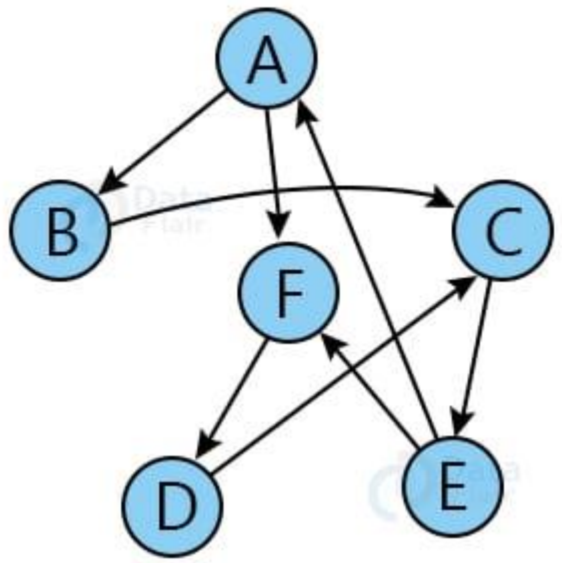


0	1	2	4		Visited
3					Stack

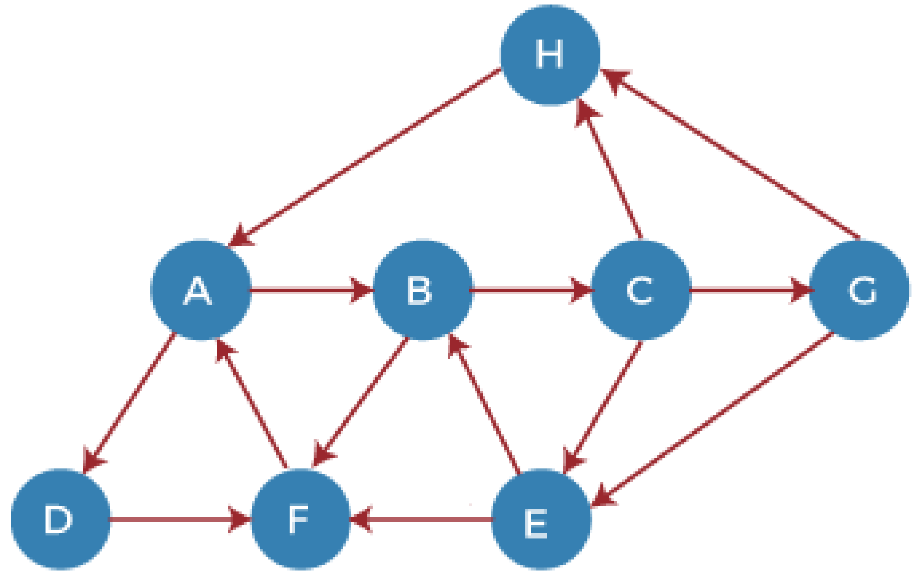


0	1	2	4	3	Visited
					Stack

EXAMPLE



A ⟶ B,F
B ⟶ C
C ⟶ E
D ⟶ C
E ⟶ A,F
F ⟶ D



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

```
#DFS algorithm in Python
# DFS algorithm
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)
    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited


graph = {'0': set(['1', '2']),
         '1': set(['0', '3', '4']),
         '2': set(['0']),
         '3': set(['1']),
         '4': set(['2', '3'])}
dfs(graph, '0')
```

- we run the DFS function on every node
- Because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.

The time complexity of the DFS algorithm is represented in the form of 0(V+E) where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is 0(V)

Application of DFS Algorithm

- For finding the path between two vertices.
- DFS algorithm can be used to implement the topological sorting.
- To test if the graph is bipartite
- For finding the strongly connected components of a graph
- For detecting cycles in a graph

Complexity Analysis

- It is important to understand the *complexity* of an algorithm.
  - *Time complexity:* How much time will it take to compute a solution to the problem?
  - *Space complexity:* How much memory do we need to do so?
- We may be interested in both a *worst-case* and an *average-case* complexity analysis.
  - *Worst-case analysis:* How much time/memory will the algorithm require in the worst case?
  - *Average-case analysis:* How much time/memory will the algorithm require on average?
- It is typically extremely difficult to give a formal average-case analysis that is theoretically sound. Experimental studies using real-world data are often the only way.

Time Complexity of Depth-first Search

- As there can be infinite loops, in the worst case, the simple depth-first algorithm will never stop. So we are going to analyse depth-bounded depth-first search instead.
- Let d_{max} be the maximal depth allowed. (If we happen to know that no branch in the tree can be longer than d_{max} , then our analysis will also apply to the other two depth-first algorithms.)
- For simplicity, assume that for every possible state there are exactly b possible follow-up states. That is, b is the branching factor of the search tree.

Time Complexity of Depth-first Search (cont.)

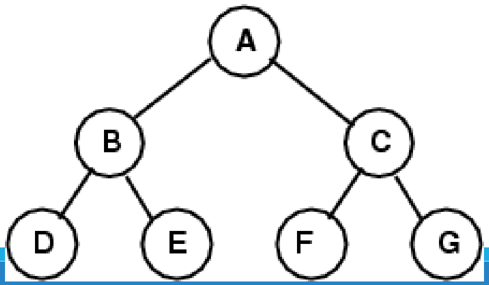
- *What is the worst case?*

In the worst case, every branch has length d_{max} (or more) and the only node labelled with a goal state is the last node on the rightmost branch. Hence, depth-first search will visit *all* the nodes in the tree (up to depth d_{max}) before finding a solution.

- So *how many nodes* are there in a tree of height d_{max} with branching factor b?

$$1 + b + b^2 + b^3 + \cdots + b^{d_{max}}$$

<u>Example:</u> $b = 2$ and $d_{max} = 2$



$1+2^1+2^2 = 2^{2+1} - 1 = 7$

Exponential Complexity

In general, in Computer Science, anything exponential is considered bad news. Indeed, our simple search techniques will usually not work very well (or at all) for larger problem instances.

Suppose the branching factor is *b* = 4 and suppose it takes us
1 millisecond to check one node.

What kind of depth bound would be feasible to use in depth-first search?

Depth	Nodes	Time
2	21	0.021 seconds
5	1365	1.365 seconds
10	1398101	23.3 minutes
15	1431655765	16.6 days
20	1466015503701	46.5 years

Space Complexity of Depth-first Search

The good news is that depth-first search is very efficient in view of its memory requirements:

•At any point in time, we only need to keep the path from the root to the current node in memory, and —depending on the exact implementation— possibly also all the sibling nodes for each of the nodes in that path.

•The length of the path is at most d_{max} +1 and each of the nodes on the path will have at most b—1 siblings left to consider.

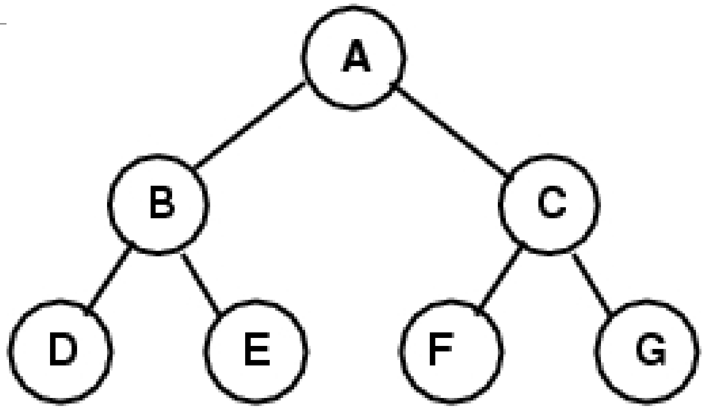
•Hence, the worst-case space complexity is $O(b \cdot d_{max})$. That is, the complexity is *linear* in d_{max} .

Breadth-first Search

The problem with (unbounded) depth-first search is that we may get lost in an infinite branch, while there could be another short branch leading to a solution.

The problem with depth-bounded depth-first search is that it can be difficult to correctly estimate a good value for the bound.

Such problems can be overcome by using *breadth-first* search, where we explore (right-hand) siblings before children.



Breadth-first Search: Implementation Difficulties

How do we keep track of which nodes we have already visited and how do we identify the next node to go to?

Recall that for depth-first search, in theory, we had to keep the current branch in memory, together with all the sibling nodes of the nodes on that branch.

Because of the way backtracking works, we actually only had to keep track of the current node

For breadth-first search, we are going to have to take care of the memory management by ourselves.

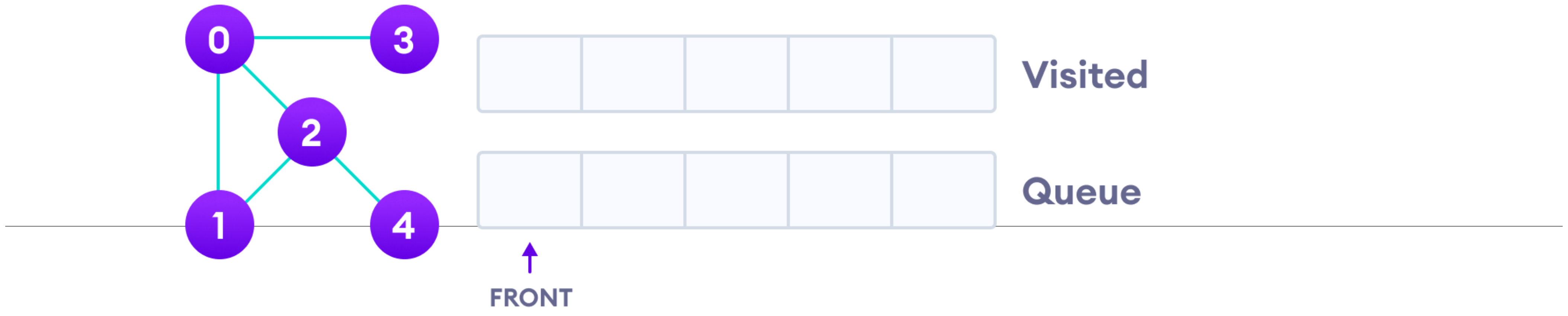
Implementation Idea

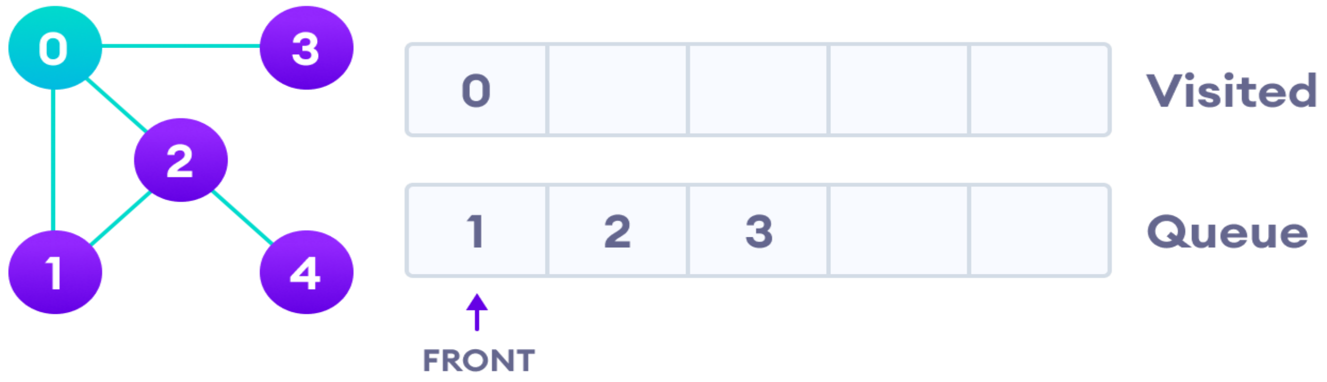
The algorithm will maintain a *list of the currently active paths*. Each round of the algorithm running consists of three steps:

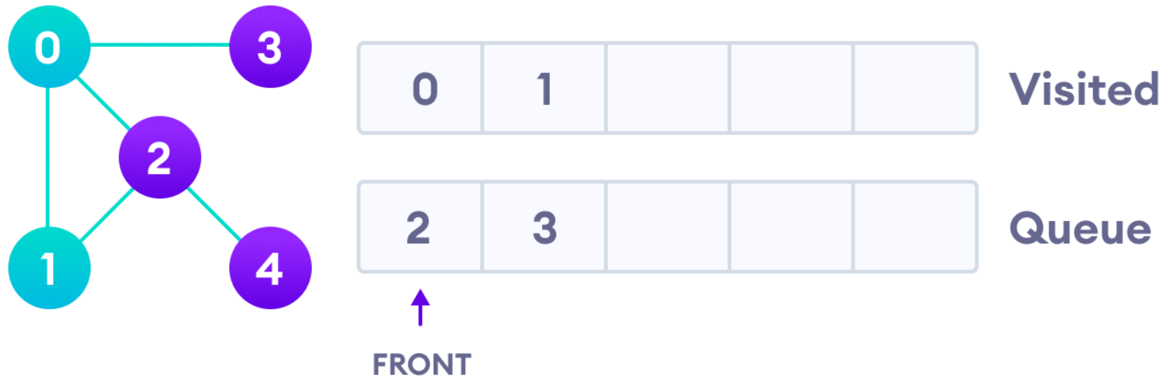
(1) Remove the first path from the list of paths.

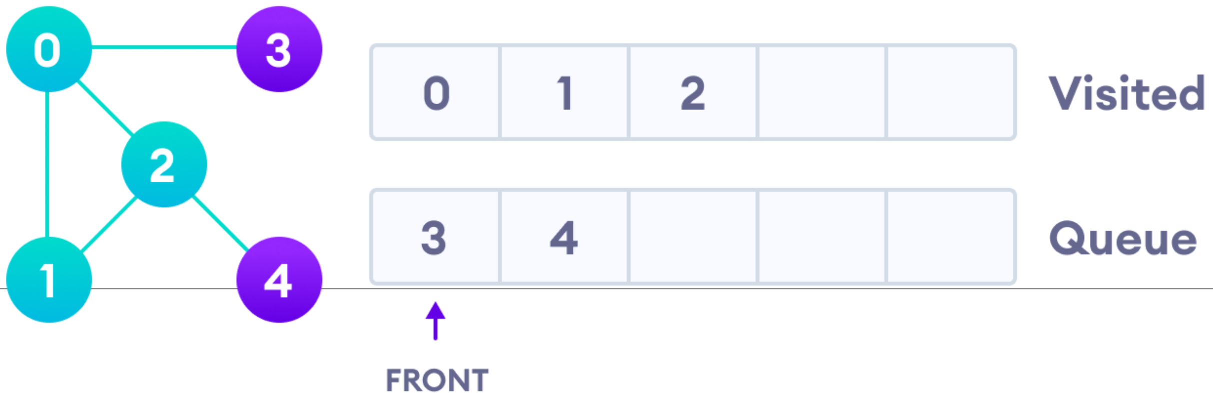
(2) Generate a new path for every possible follow-up state of the state labelling the last node in the selected path.

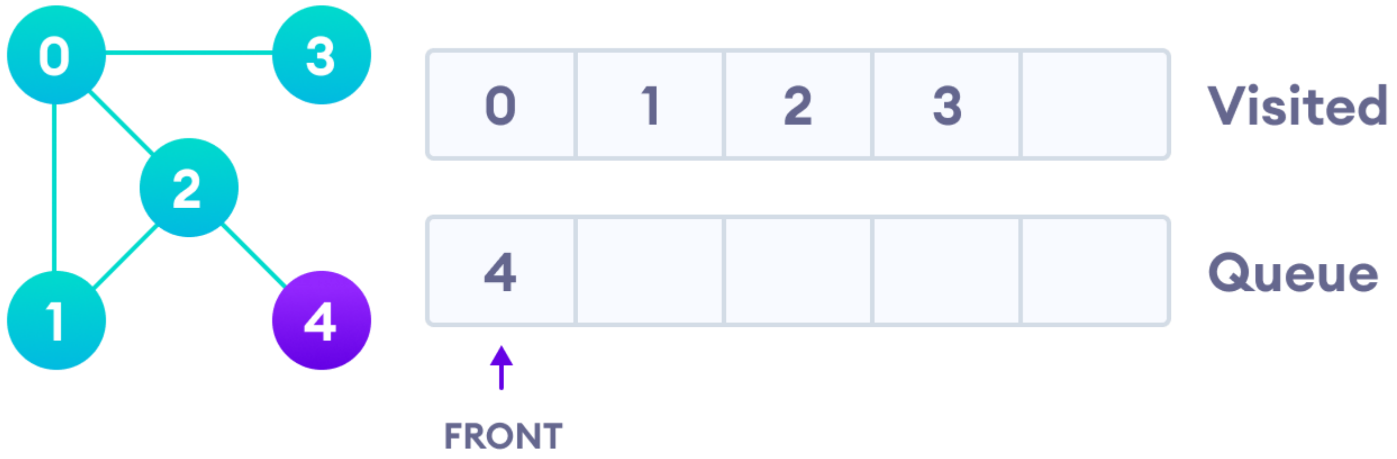
(3) Append the list of newly generated paths to the *end* of the list of paths (to ensure paths are really being visited in breadth-first order).

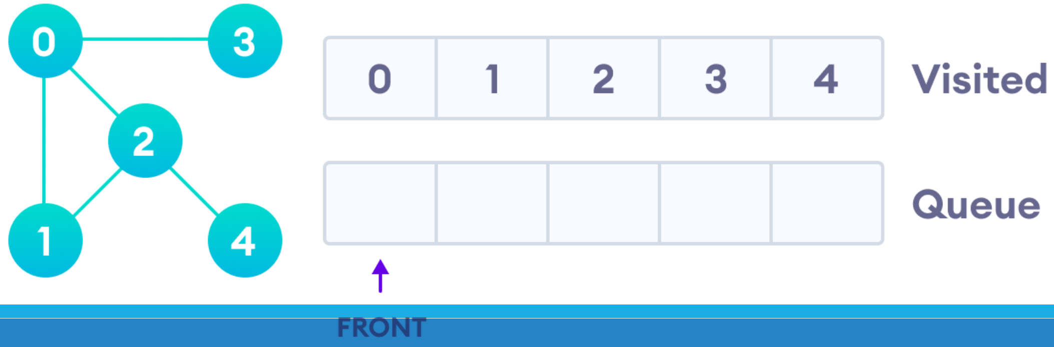

0
3
2
1
4
Visited
Queue
FRONT


0
3
2
1
4
0
Visited
1
2
3
Queue
FRONT


0
3
2
1
4
0
1
Visited
2
3
Queue
FRONT


0
3
2
1
4
0 1 2
Visited
3 4
Queue
FRONT


0
3
2
1
4
0 1 2 3
Visited
4
Queue
FRONT


0
3
2
1
4
0 1 2 3 4
Visited
Queue
FRONT

```
# BFS algorithm in Python

import collections

# BFS algorithm
def bfs(graph, root):

    visited, queue = set(), collections.deque([root])
    visited.add(root)

    while queue:

        # Dequeue a vertex from queue
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")

        # If not visited, mark it as visited, and
        # enqueue it
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)


if __name__ == '__main__':
    graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
    print("Following is Breadth First Traversal: ")
    bfs(graph, 0)
```

BFS Algorithm Applications

To build index by search index

For GPS navigation

Path finding algorithms

In Ford-Fulkerson algorithm to find maximum flow in a network

Cycle detection in an undirected graph

In minimum spanning tree

The time complexity of the DFS algorithm is represented in the form of 0(V+E) where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is 0(V)

Completeness and Optimality

Some good news about breadth-first search:

•Breadth-first search guarantees ***completeness:*** if there exists a solution it will be found eventually.

•Breadth-first search also guarantees ***optimality:*** the first solution returned will be as short as possible.

(Remark: This interpretation of optimality assumes that every move has got a cost of 1. With real cost functions it does become a little more involved.)

Recall that depth-first search does not ensure either completeness or optimality.

Complexity Analysis of Breadth-first Search

Time complexity: In the worst case, we have to search through the entire tree for any search algorithm. As both depth-first and breadth-first search visit each node exactly once, time complexity will be the same.

Let d be the the depth of the first solution and let b be the branching factor (again, assumed to be constant for simplicity). Then worst-case time complexity is $O(b^d)$.

Space complexity: Big difference; now we have to store every path visited before, while for depth-first we only had to keep a single branch in memory. Hence, space complexity is also $O(b^d)$.

So there is a *trade-o*↵ between memory-requirements on the one hand and completeness/optimality considerations on the other.

Best of Both Worlds

We would like an algorithm that, like breadth-first search, is guaranteed (1) to visit every node on the tree eventually and (2) to return the shortest possible solution, but with (3) the favourable memory requirements of a depth-first algorithm.

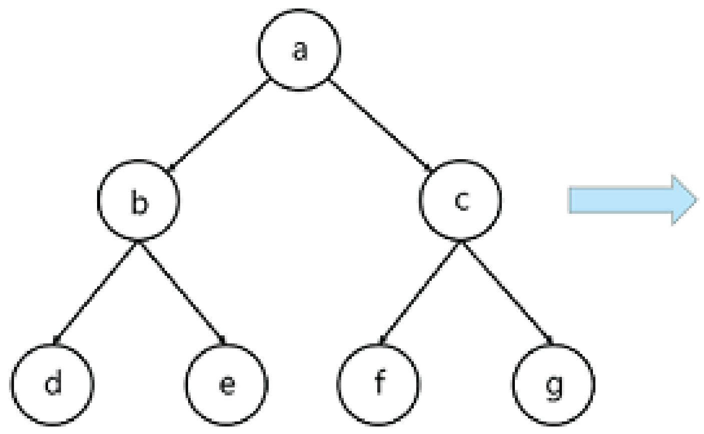
Observation: Depth-bounded depth-first search *almost* fits the bill. The only problem is that we may choose the bound either

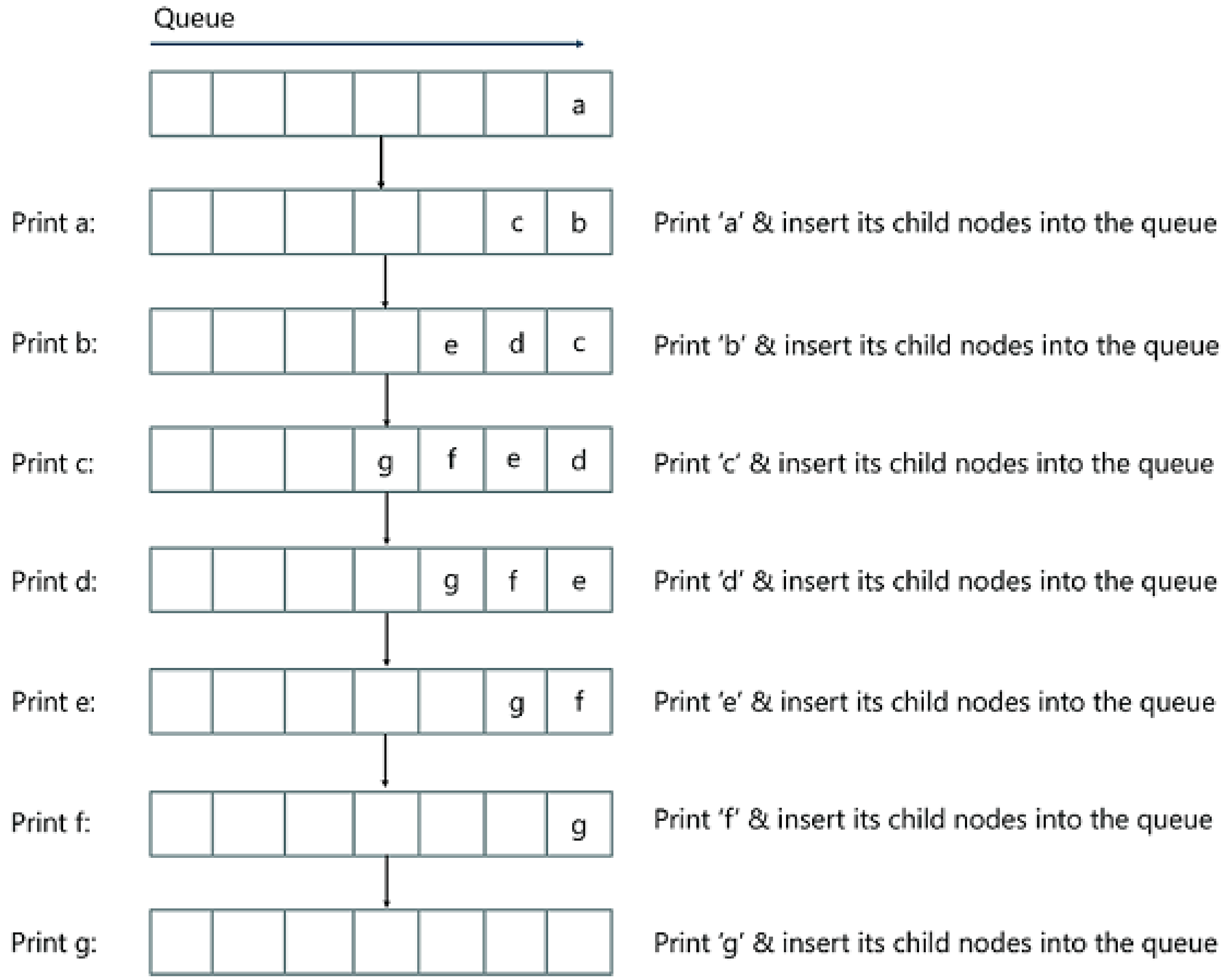
•*too low* (losing completeness by stopping early) or

•*too high* (becoming too similar to normal depth-first with the danger of getting lost in a single deep branch).

Idea: Run depth-bounded depth-first search again and again, with increasing values for the bound!

This approach is called *iterative deepening* .. .


a
b
c
d
e
f
g


Queue
a
Print a:
c b
Print 'a' & insert its child nodes into the queue
Print b:
e d c
Print 'b' & insert its child nodes into the queue
Print c:
g f e d
Print 'c' & insert its child nodes into the queue
Print d:
g f e
Print 'd' & insert its child nodes into the queue
Print e:
g f
Print 'e' & insert its child nodes into the queue
Print f:
g
Print 'f' & insert its child nodes into the queue
Print g:
Print 'g' & insert its child nodes into the queue

Iterative Deepening

We can specify the iterative deepening algorithm as follows:

(1)Set *n* to 0.

(2)Run depth-bounded depth-first search with bound *n*.

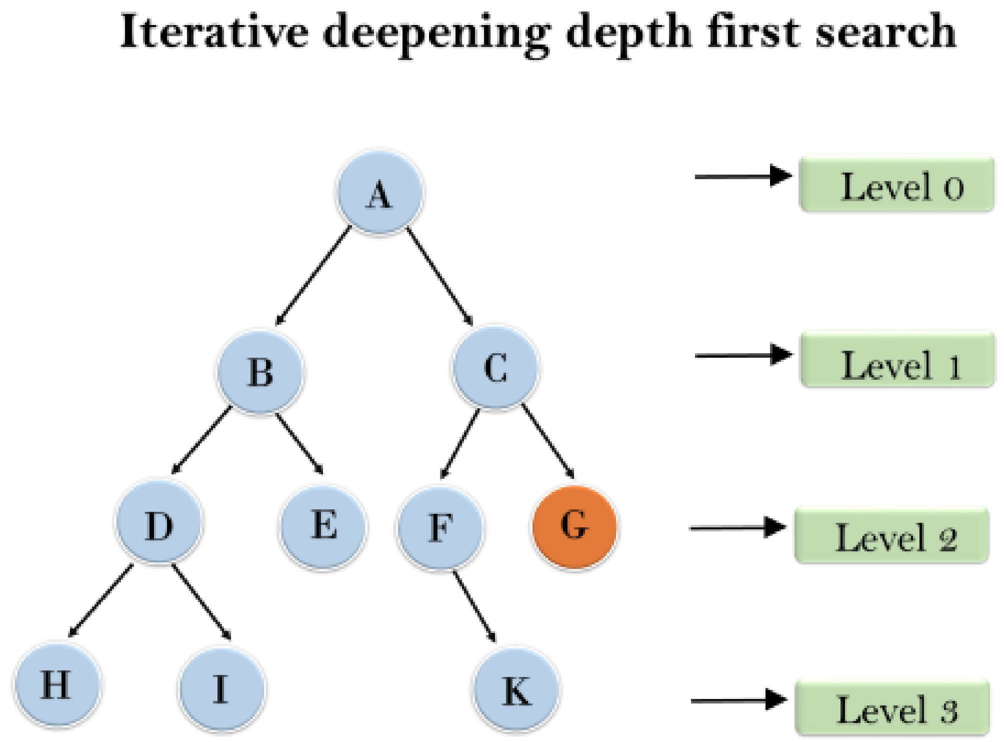
(3)Stop and return answer in case of success; increment *n* by 1 and go back to (2) otherwise.

However, in Prolog we can implement the same algorithm also in a more compact manner . . .

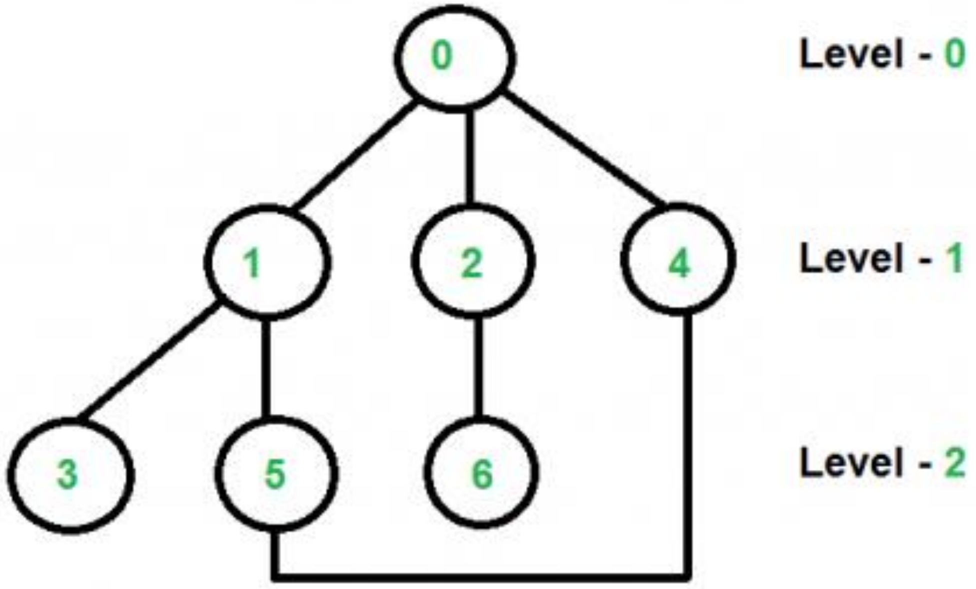
Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node.

1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.



Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



The explanation of the above pattern is left to the readers.

Completeness:

- This algorithm is complete is if the branching factor is finite.

Time Complexity:

- Let's suppose 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.

Some other approaches

Depth-Limited Search Algorithm:

- A depth-limited search algorithm is similar to depth-first search with a predetermined limit

Uniform-cost Search Algorithm:

- Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge

Bidirectional Search Algorithm:

- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node.

Depth-Limited Search Algorithm:

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

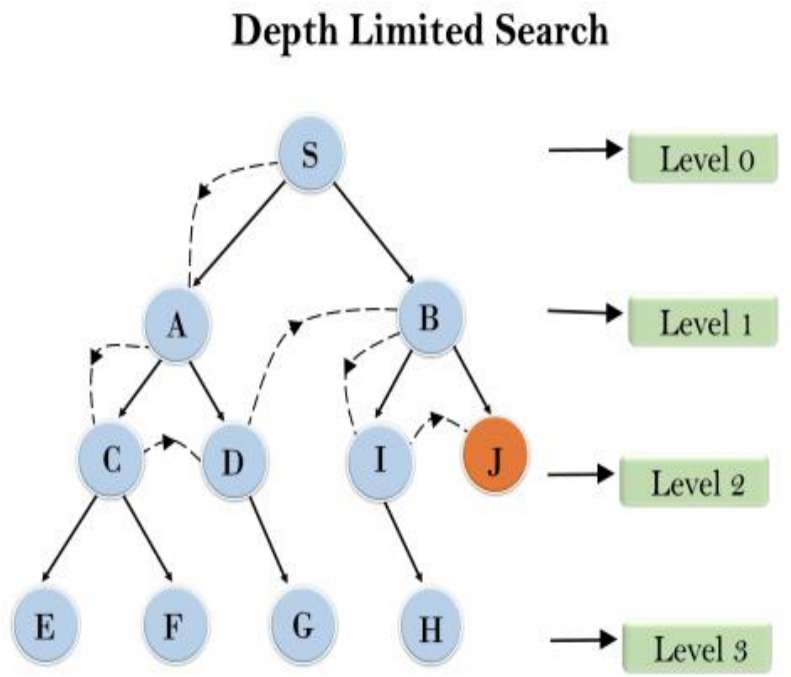
- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

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^{ℓ})**.
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 $\ell > d$.

Uniform-cost Search Algorithm:

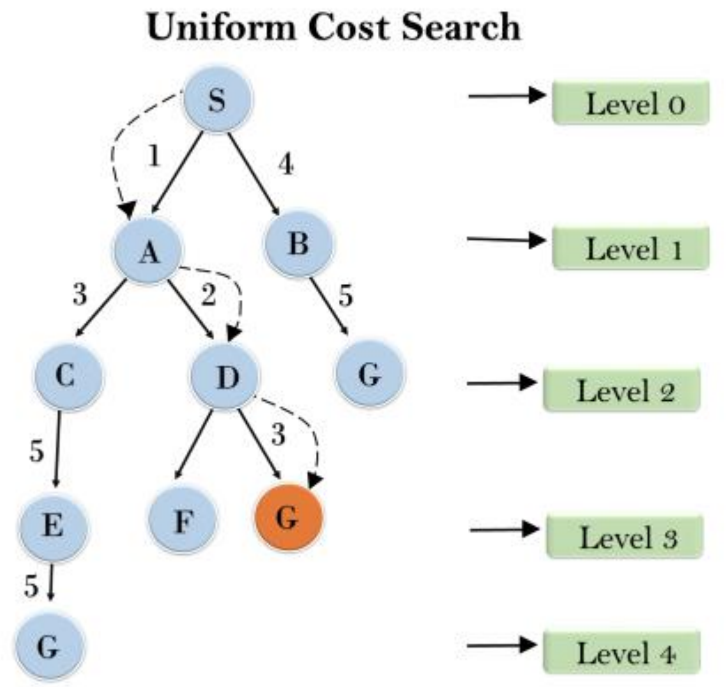
The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost.

Uniform-cost search expands nodes according to their path costs form the root node.

It can be used to solve any graph/tree where the optimal cost is in demand.

A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost.

Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.



Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Completeness:

- Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

- Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.
- Hence, the worst-case time complexity of Uniform-cost search is**$O(b^{1 + [C^*/\varepsilon]})$/**.

Space Complexity:

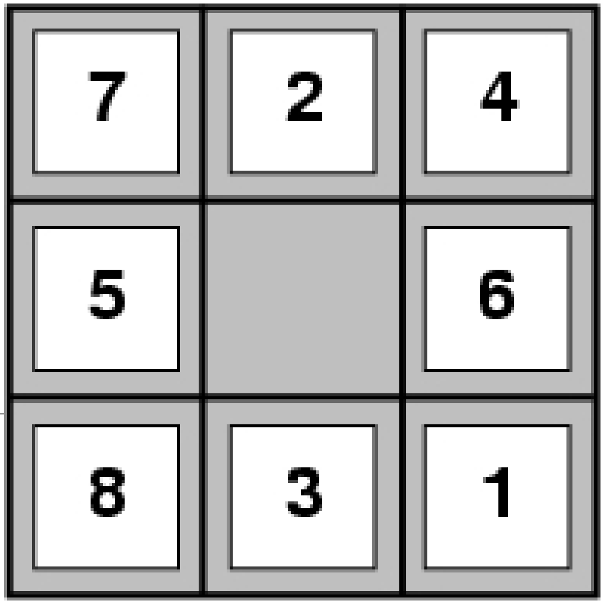
- The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is **$O(b^{1 + [C^*/\varepsilon]})$**.

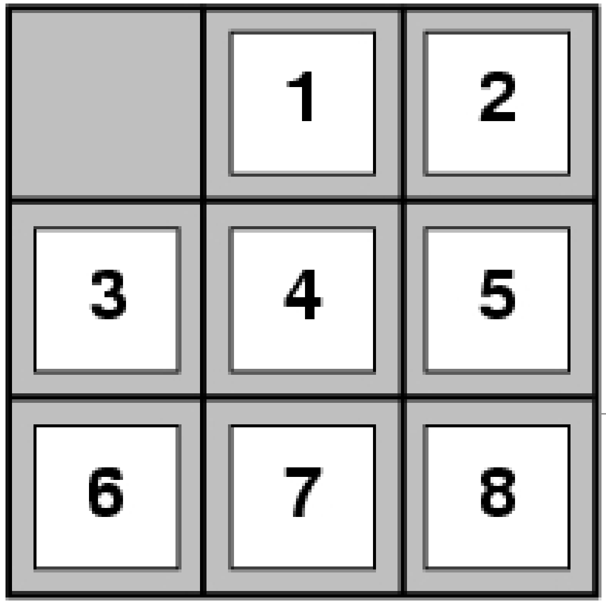
Optimal:

- Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

Eight-Puzzle

Yet another puzzle …





Source: Russell & Norvig, *Artificial Intelligence*

What are heuristics?

- Heuristics are know-hows obtained

through a lot of experiences.

- Heuristics often enable us (or animals) to act quickly without thinking deeply about the reasons or without thinking at all.
- In many cases, heuristics are “tacit knowledge”

that cannot be explained verbally.

- The more experiences we have, the more and the better the heuristics will be.

Definitions

Heuristics (Greek *heuriskein* = find, discover): "the study of the methods and rules of discovery and invention".

We use our knowledge of the problem to consider some (not all) successors of the current state (preferably just one, as with an oracle). This means pruning the state space, gaining speed, but perhaps missing the solution!

In chess: consider one (apparently best) move, maybe a few -- but not all possible legal moves.

In the travelling salesman problem: select one nearest city, give up complete search (the greedy technique). This gives us, in polynomial time, an approximate solution of the inherently exponential problem; it can be proven that the approximation error is bounded.

Definitions (2)

For heuristic search to work, we must be able to rank the children of a node. A heuristic function takes a state and returns a numeric value -- a composite assessment of this state. We then choose a child with the best score (this could be a maximum or minimum).

A heuristic function can help gain or lose a lot, but finding the right function is not always easy.

The 8-puzzle: how many misplaced tiles? how many slots away from the correct place? and so on.

Water jugs: ???

Chess: no simple counting of pieces is adequate.

Definitions (3)

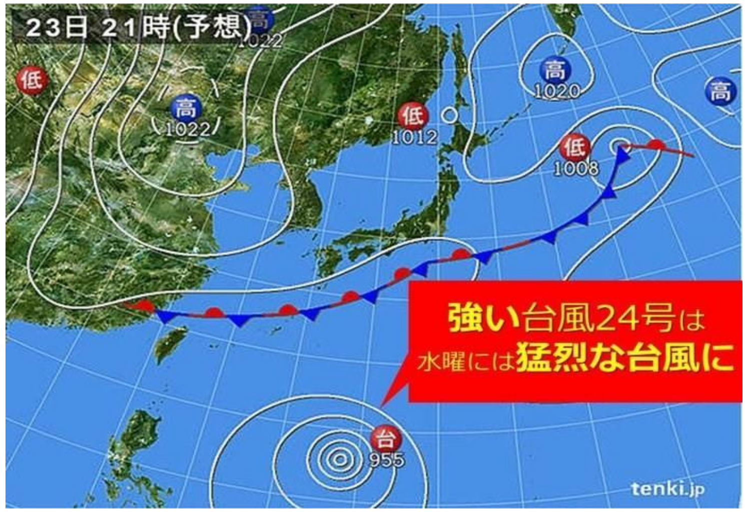
The principal gain -- often spectacular -- is the reduction of the state space. For example, the full tree for Tic-Tac-Toe has 9! leaves. If we consider symmetries, the tree becomes six times smaller, but it is still quite large.

With a fairly simple heuristic function we can get the tree down to 40 states. (More on this when we discuss games.)

Heuristics can also help speed up exhaustive, blind search, such as depth-first and breadth-first search.

Why heuristic search?

- Based on the heuristics, we can get good solutions without investigating all possible cases.
- In fact, a deep learner is used in Alpha-Go to learn heuristics for playing Go-game, and this learner can help the system to make decisions efficiently.
- Without using heuristics, many AI-related problems cannot be solved (may take many years to get a solution).



Patterns memorized so far can be used by a prediction model to make a hypothesis about the move of the typhon.

Hill climbing

This is a *greedy* algorithm: go as high up as possible as fast as possible, without looking around too much.

The algorithm

select a heuristic function;

```
set C, the current node, to the highest-valued initial node;
loop
  ◦select N, the highest-valued child of C;
  ◦return C if its value is better than the value of N;
  otherwise set C to N;
```

Hill climbing

- This is a *greedy* algorithm: go as high up as possible as fast as possible, without looking around too much.

  - The algorithm

- select a heuristic function;
- set C, the current node, to the highest-valued initial node;
- **loop**
  - select N, the highest-valued child of C;
  - return C if its value is better than the value of N;
  otherwise set C to N;

- Define the current state as an initial state
- Loop until the goal state is achieved or no more operators can be applied on the current state:
  - Apply an operation to current state and **get a new state**
  - **Compare** the new state with the goal
  - **Quit** if the goal state is achieved
  - Evaluate new state with heuristic function and **compare it with the current state**
  - **If the newer state is closer** to the goal compared to current state, **update the current state**

Hill-Climbing Features

A hill-climbing algorithm has four main features:

It employs a **greedy approach:** This means that it moves in a direction in which the cost function is optimized. The greedy approach enables the algorithm to establish local maxima or minima.

No Backtracking: A hill-climbing algorithm only works on the current state and succeeding states (future). It does not look at the previous states.

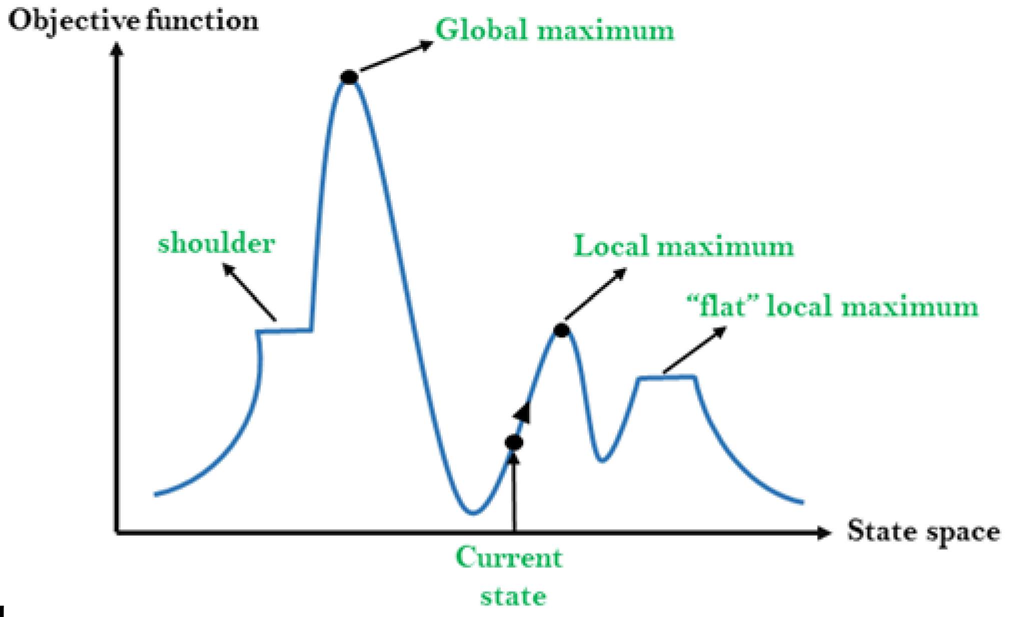
Feedback mechanism: The algorithm has a feedback mechanism that helps it decide on the direction of movement (whether up or down the hill). The feedback mechanism is enhanced through the generate-and-test technique.

Incremental change: The algorithm improves the current solution by incremental changes.

State-Space of Hill Climbing

A state-space diagram consists of various regions that can be explained as follows;

- **Local maximum:** A local maximum is a solution that surpasses other neighboring solutions or states but is not the best possible solution.
- **Global maximum:** This is the best possible solution achieved by the algorithm.
- **Current state:** This is the existing or present state.
- **Flat local maximum:** This is a flat region where the neighboring solutions attain the same value.
- **Shoulder:** This is a plateau whose edge is stretching upwards.



Variant of generate and test algorithm: It is a variant of generating and test algorithm. The generate and test algorithm is as follows :

- *Generate possible solutions.*
- *Test to see if this is the expected solution.*
- *If the solution has been found quit else go to step 1.*

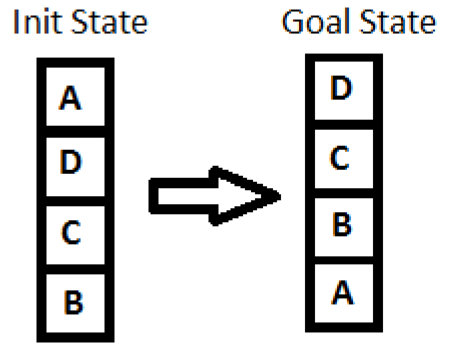
Hence we call Hill climbing a variant of generating and test algorithm as it takes the feedback from the test procedure. Then this feedback is utilized by the generator in deciding the next move in the search space.

Problems with hill climbing

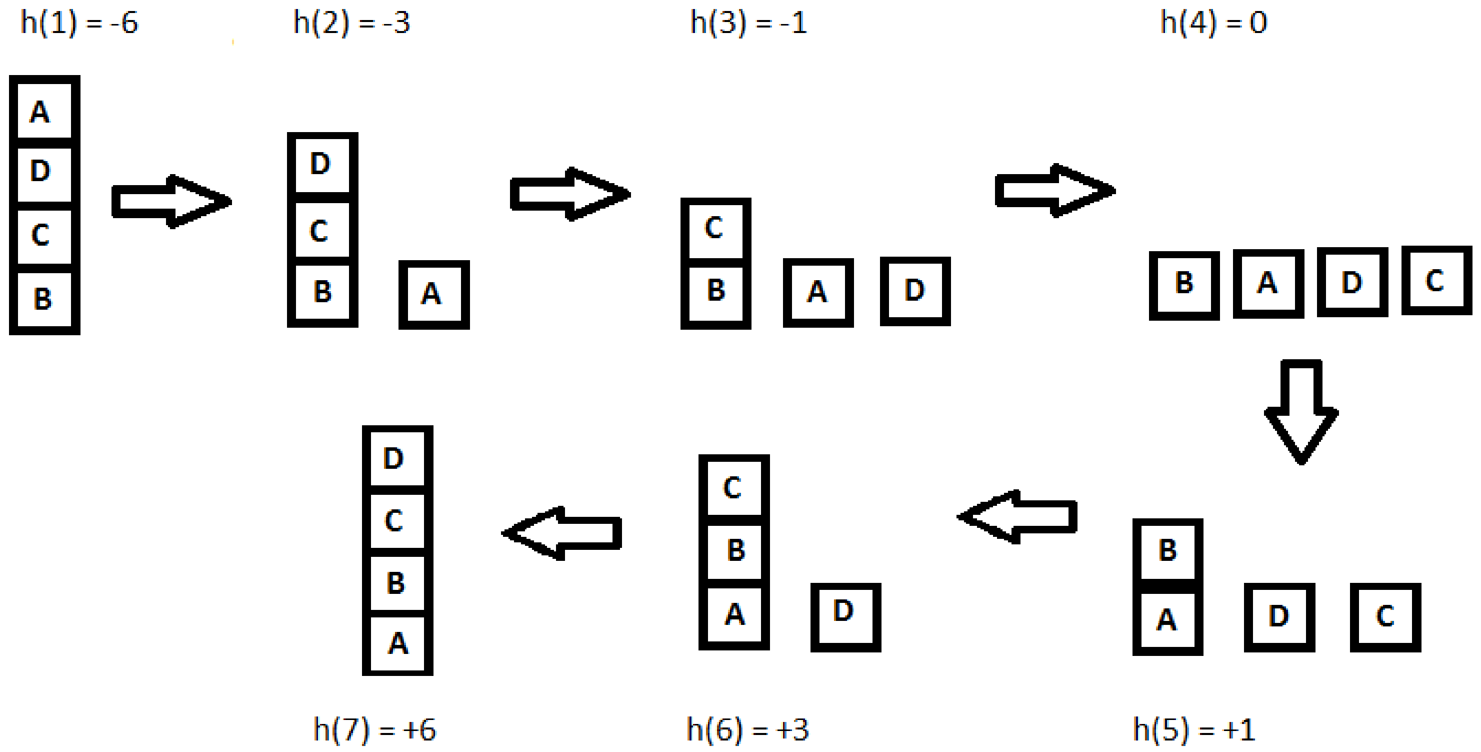
1. Local maximum, or the foothill problem: there is a peak, but it is lower than the highest peak in the whole space.
2. The plateau problem: all local moves are equally unpromising, and all peaks seem far away.
3. The ridge problem: almost every move takes us down.

- Random-restart hill climbing is a series of hill-climbing searches with a randomly selected start node whenever the current search gets stuck.
- See also simulated annealing -- in a moment.

Example



Let's define such function *h:*
h(x) **= +1 for all the blocks in the support structure if the block is correctly positioned otherwise -1 for all the blocks in the support structure.**



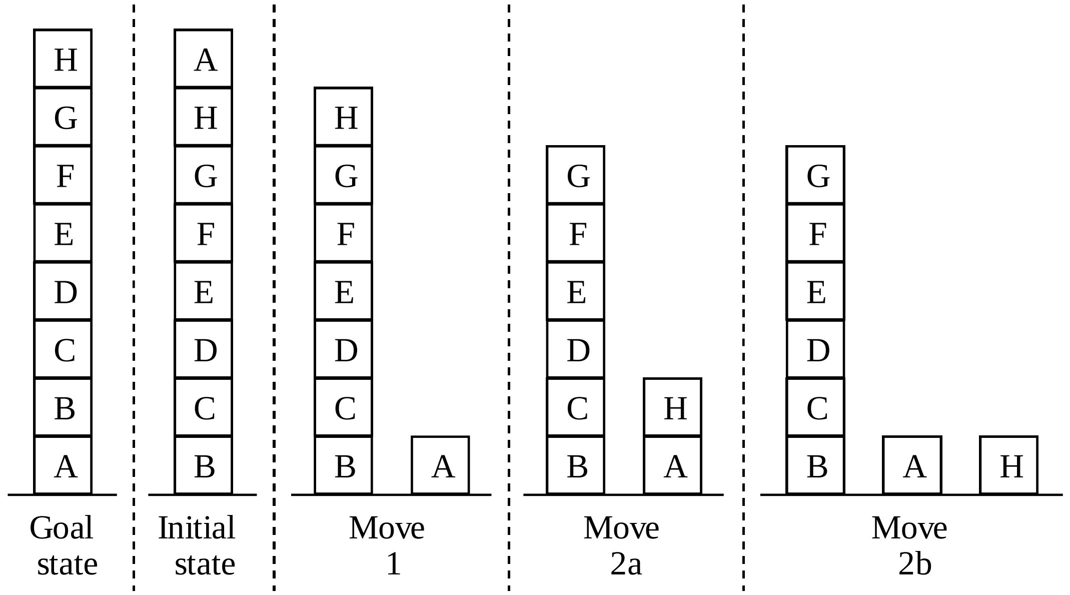
Problems with hill climbing

1. Local maximum, or the foothill problem: there is a peak, but it is lower than the highest peak in the whole space.
2. The plateau problem: all local moves are equally unpromising, and all peaks seem far away.
3. The ridge problem: almost every move takes us down.

Random-restart hill climbing is a series of hill-climbing searches with a randomly selected start node whenever the current search gets stuck.

See also simulated annealing -- in a moment.

A hill climbing example



A hill climbing example (2)

A local heuristic function

Count +1 for every block that sits on the correct thing. The goal state has the value +8.

Count -1 for every block that sits on an incorrect thing. In the initial state blocks C, D, E, F, G, H count +1 each. Blocks A, B count -1 each , for the total of +4.

Move 1 gives the value +6 (A is now on the correct support). Moves 2a and 2b both give +4 (B and H are wrongly situated). This means we have a local maximum of +6.

A hill climbing example (4)

Move 1 gives the value -21 (A is now on the correct support).

Move 2a gives -16, because C, D, E, F, G, H
count -1, -2, -3, -4, -5, -1.

Move 2b gives -15, because C, D, E, F, G
count -1, -2, -3, -4, -5.

There is no local maximum!

Moral: sometimes changing the heuristic function is all we need.

Types of Hill Climbing

A. Simple Hill climbing:

- It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as the next node.

B. Steepest-Ascent Hill climbing:

- It first examines all the neighboring nodes and then selects the node closest to the solution state as of the next node.

C. Stochastic hill climbing:

- It does not examine all the neighboring nodes before deciding which node to select. It just selects a neighboring node at random and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

Simulated Annealing

Simulated annealing is a stochastic global search algorithm for function optimization.

It makes use of randomness as part of the search process. This makes the algorithm appropriate for nonlinear objective functions where other local search algorithms do not operate well.

Like the stochastic hill climbing local search algorithm, it modifies a single solution and searches the relatively local area of the search space until the local optima is located.

Unlike the hill climbing algorithm, it may accept worse solutions as the current working solution.

The simulated annealing optimization algorithm can be thought of as a modified version of stochastic hill climbing.

Stochastic hill climbing maintains a single candidate solution and takes steps of a random but constrained size from the candidate in the search space. If the new point is better than the current point, then the current point is replaced with the new point. This process continues for a fixed number of iterations.

Simulated annealing executes the search in the same way. The main difference is that new points that are not as good as the current point (worse points) are accepted sometimes.

$$P(\Delta E) = e^{-\frac{\Delta E}{k * t}}$$

A worse point is accepted probabilistically where the likelihood of accepting a solution worse than the current solution is a function of the temperature of the search and how much worse the solution is than the current solution.

Differences with hill climbing

The annealing scheduling must be maintained

Moves to worse states may be accepted

It is good idea to maintain, in addition to the current state, the best state found so far. Then, if the final state is worse than the earlier state (because of bad luck in accepting moves to worse state) the earlier state is still available.

Advantages

Easy to implement and use

Provides optimal solutions to a wide range of problems

Disadvantages

Can take a long time to run if the annealing schedule is very long

There are a lot of tunable parameters in this algorithm

Let’s now try to draw parallel’s between Annealing in metallurgy and Simulated annealing for Feature selection:

In terms of feature selection,

1. **Set of features** represents the arrangement of molecules in the material(metal).
2. **No. of Iterations** represents time. Therefore, as the no. of iterations decreases temperature decreases.
3. Change in **Predictive performance** between the previous and the current iteration represents the change in material’s energy.

Algorithm:

Let’s go over the exact Simulated Annealing algorithm, step-by-step.

1. The initial step is to select a subset of features at random.
2. Then choose the no. of iterations. A ML model is then built and the predictive performance (otherwise called objective function) is calculated.
3. A small percentage of features are randomly included/excluded from the model. This is just to ‘perturb’ the features. Then the predictive performance is calculated once again for this new set of features.

Two things can happen here:

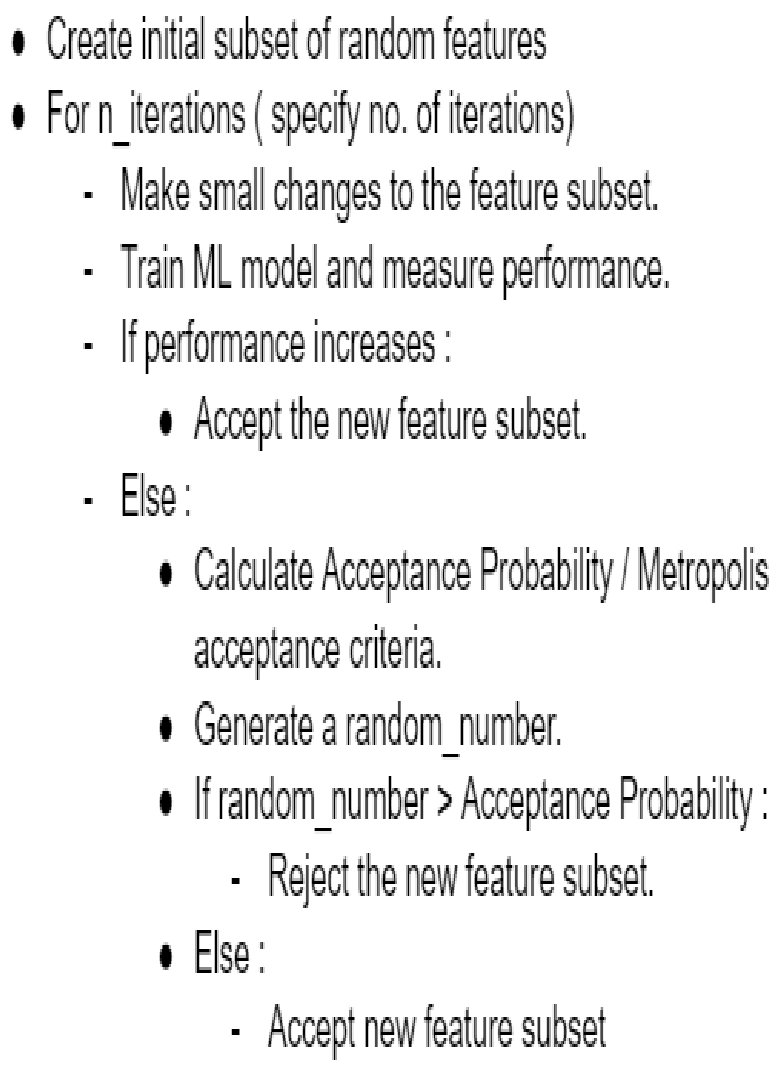
1. If performance **Increases in the new set** then the new feature set is **Accepted**.
2. If the performance of the new feature set has **worse performance**, then the **Acceptance Probability** (otherwise called **metropolis acceptance criterion**) is calculated. (You will see the formula and its significance shortly. Stay with the flow for now.)

Once the acceptance probability is calculate
generate a random number between 0 – 1

1. If the **Random Number** > **Acceptance Probability** then the new feature set is **Rejected** and the previous feature set will be continued to be used.
2. If the **Random Number** < **Acceptance Probability** then the new feature set is **Accepted**.

The impact of randomness by this process helps simulated annealing to not get stuck a
local optimums in search of a global optimu

Keep doing this for the chosen number of iterations.



```
• Create initial subset of random features
• For n_iterations ( specify no. of iterations)
    ▪ Make small changes to the feature subset.
    ▪ Train ML model and measure performance.
    ▪ If performance increases :
        • Accept the new feature subset.
    ▪ Else :
        • Calculate Acceptance Probability / Metropolis
          acceptance criteria.
        • Generate a random_number.
        • If random_number > Acceptance Probability :
            ▪ Reject the new feature subset.
        • Else :
            ▪ Accept new feature subset
```

Search Algorithms and Representations

Breadth-first

Depth-first

Best-first (Heuristic Search)

A*

Hill Climbing

Limiting the number of Plies

Minimax

Alpha-Beta Pruning

Adding Constraints

Genetic Algorithms

Forward vs Backward Chaining

We will study various forms of representation and uncertainty handling in the next class period

Knowledge needs to be represented

- Production systems of some form are very common
  - If-then rules
  - Predicate calculus rules
  - Operators
- Other general forms include semantic networks, frames, scripts
- Knowledge groups
- Models, cases
- Agents
- Ontologies