Table of Contents

Agent:

An agent is anything that can be viewed as :

- perceiving its environment through sensors and
- acting upon that environment through actuators

Simple Reflex Agent

A simple reflex agent is the simplest of all the agent programs. Decisions or actions are taken based on the current percept, which does not depend on the rest of the percept history. These agents react only to the predefined rules. It works best when the environment is fully observable.

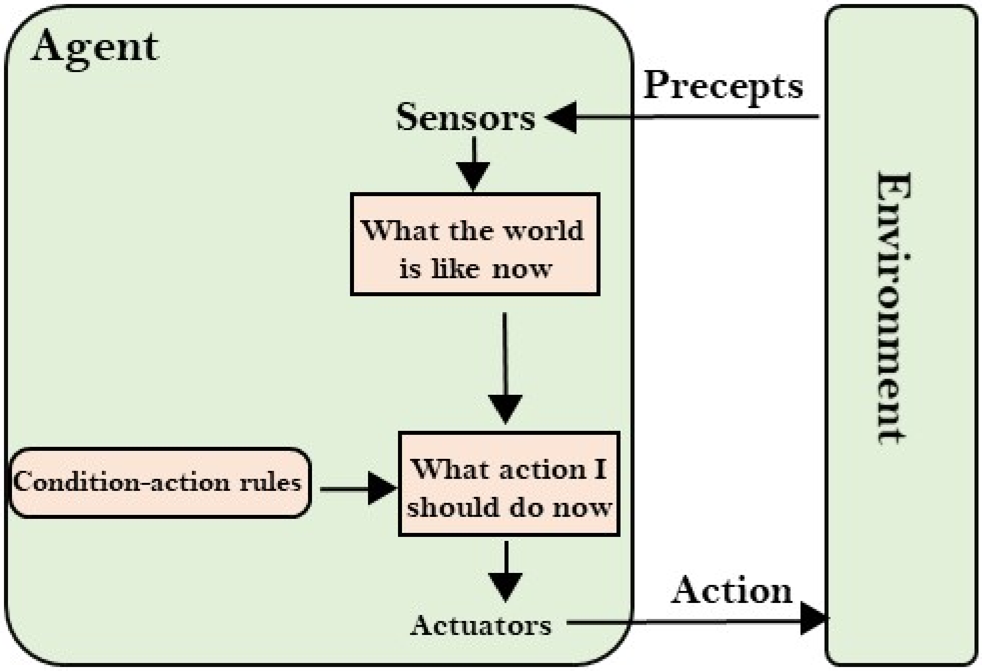
A simple reflex agent comprises the following parts:

Agent: The agent is the one who performs actions on the environment.

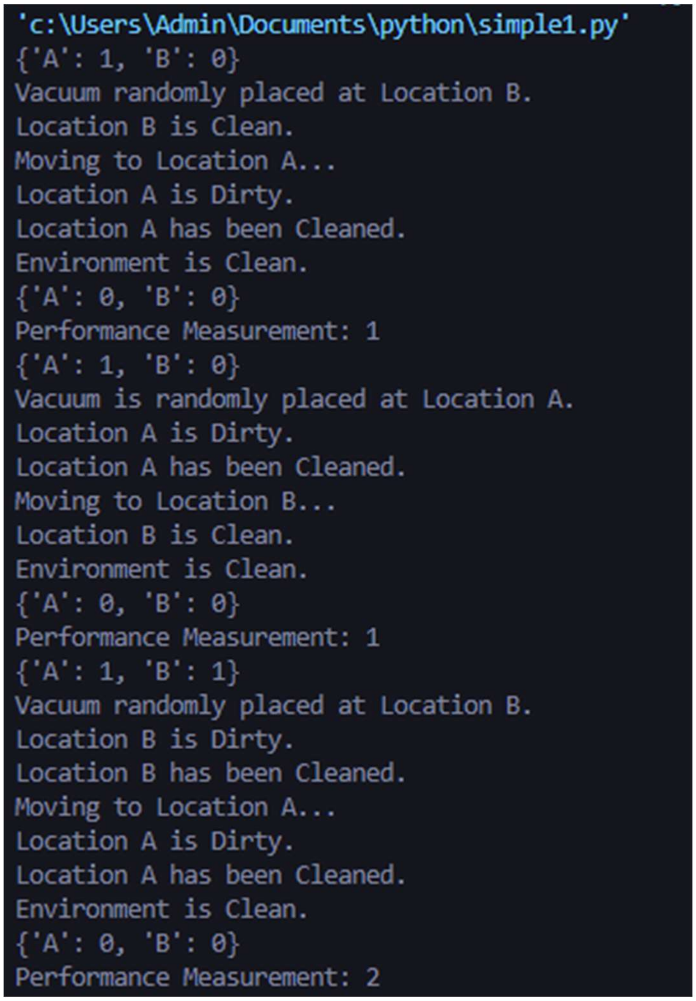
Sensors: Sensors are the things that sense the environment. They are devices that measure physical property.

Actuators: Actuators are devices that convert energy into motion.

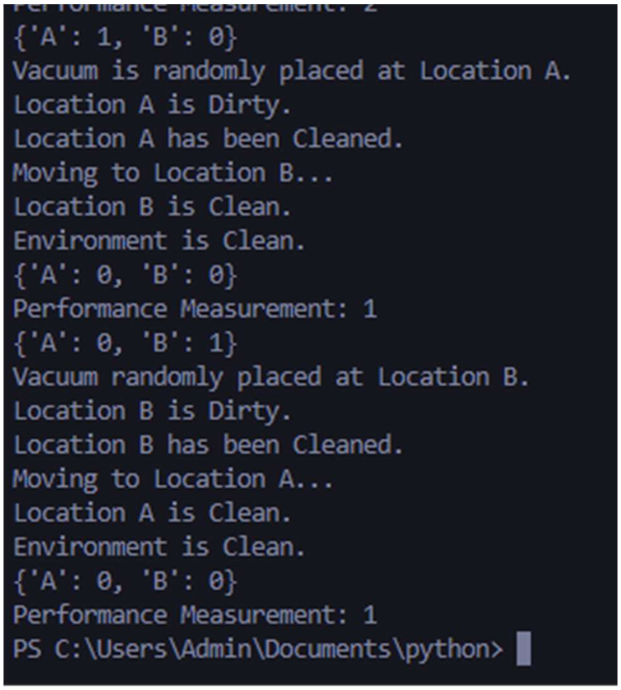
Environment: The environment includes the surroundings of the agent.



Output:



```
'c:\Users\Admin\Documents\python\simple1.py'
{'A': 1, 'B': 0}
Vacuum randomly placed at Location B.
Location B is Clean.
Moving to Location A...
Location A is Dirty.
Location A has been Cleaned.
Environment is Clean.
{'A': 0, 'B': 0}
Performance Measurement: 1
{'A': 1, 'B': 0}
Vacuum is randomly placed at Location A.
Location A is Dirty.
Location A has been Cleaned.
Moving to Location B...
Location B is Clean.
Environment is Clean.
{'A': 0, 'B': 0}
Performance Measurement: 1
{'A': 1, 'B': 1}
Vacuum randomly placed at Location B.
Location B is Dirty.
Location B has been Cleaned.
Moving to Location A...
Location A is Dirty.
Location A has been Cleaned.
Environment is Clean.
{'A': 0, 'B': 0}
Performance Measurement: 2
```



```
Performance Measurement: 2
{'A': 1, 'B': 0}
Vacuum is randomly placed at Location A.
Location A is Dirty.
Location A has been Cleaned.
Moving to Location B...
Location B is Clean.
Environment is Clean.
{'A': 0, 'B': 0}
Performance Measurement: 1
{'A': 0, 'B': 1}
Vacuum randomly placed at Location B.
Location B is Dirty.
Location B has been Cleaned.
Moving to Location A...
Location A is Clean.
Environment is Clean.
{'A': 0, 'B': 0}
Performance Measurement: 1
PS C:\Users\Admin\Documents\python>
```

Model Based Reflex Agent

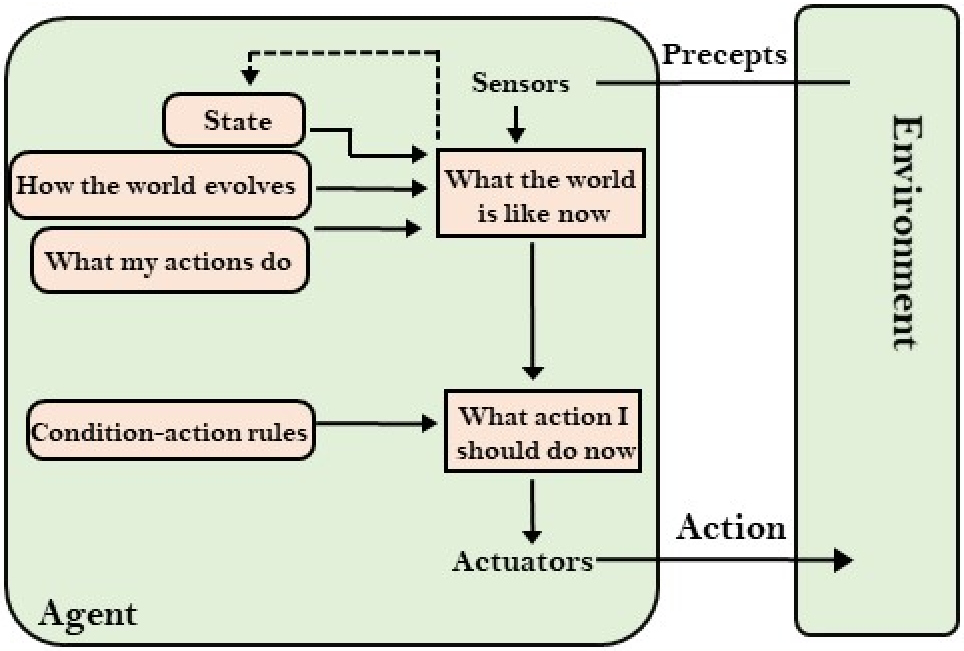
The Model-based agent can work in a partially observable environment, and track the situation.

A model-based agent has two important factors:

Model: It is knowledge about "how things happen in the world," so it is called a Model-based agent.

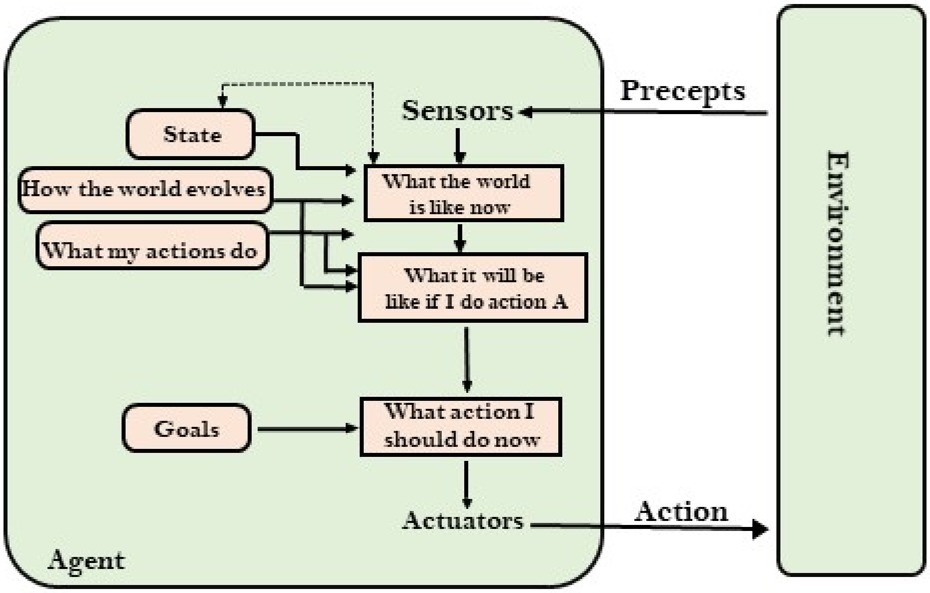
Internal State: It is a representation of the current state based on percept history.

These agents have the model, "which is knowledge of the world" and based on the model they perform actions.

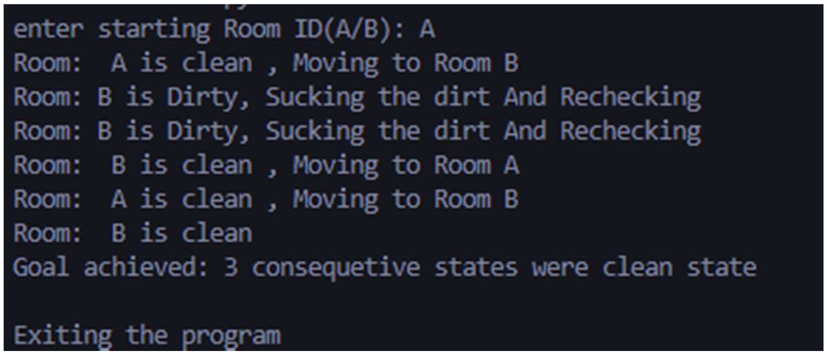


Goal Based Agent

The knowledge of the current state environment is not always sufficient to decide for an agent to what to do .The agent needs to know its goal which describes desirable situations .Goal-based agents expand the capabilities of the model-based agent by having the "goal" information .They choose an action, so that they can achieve the goal. These agents may have to consider a long sequence of possible actions before deciding whether the goal is achieved or not. Such considerations of different scenario are called searching and planning, which makes an agent proactive.



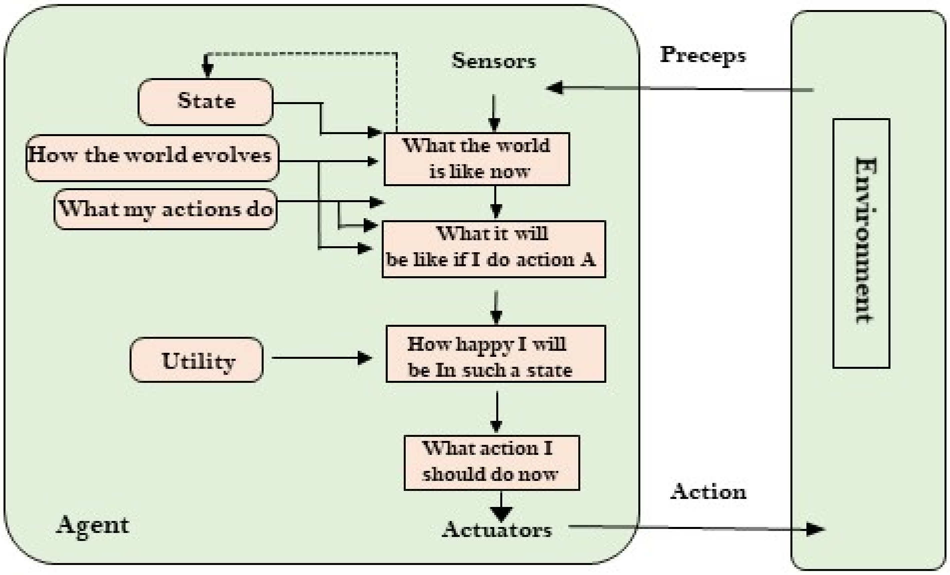
Output:



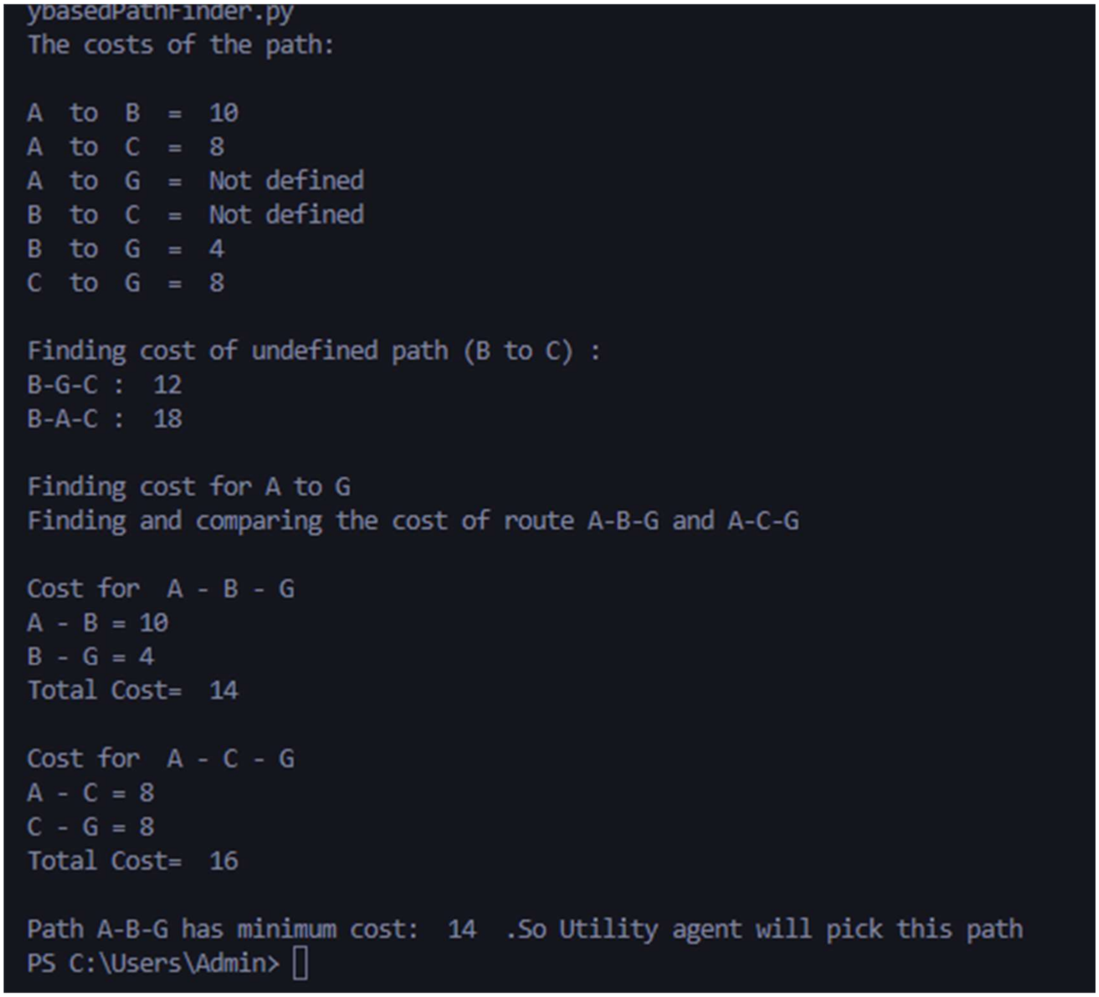
```
enter starting Room ID(A/B): A
Room:  A is clean , Moving to Room B
Room: B is Dirty, Sucking the dirt And Rechecking
Room: B is Dirty, Sucking the dirt And Rechecking
Room:  B is clean , Moving to Room A
Room:  A is clean , Moving to Room B
Room:  B is clean
Goal achieved: 3 consequetive states were clean state

Exiting the program
```

Utility Based Agent

These agents are similar to the goal-based agent but provide an extra component of utility measurement which makes them different by providing a measure of success at a given state .Utility-based agent act based not only goals but also the best way to achieve the goal .The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action .The utility function maps each state to a real number to check how efficiently each action achieves the goals.


Sensors
Preceps
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Utility
How happy I will be In such a state
What action I should do now
Environment
Action
Agent
Actuators

Output:



```
ybasedPathFinder.py
The costs of the path:

A  to  B  =  10
A  to  C  =  8
A  to  G  =  Not defined
B  to  C  =  Not defined
B  to  G  =  4
C  to  G  =  8

Finding cost of undefined path (B to C) :
B-G-C :  12
B-A-C :  18

Finding cost for A to G
Finding and comparing the cost of route A-B-G and A-C-G

Cost for  A - B - G
A - B = 10
B - G = 4
Total Cost=  14

Cost for  A - C - G
A - C = 8
C - G = 8
Total Cost=  16

Path A-B-G has minimum cost:  14  .So Utility agent will pick this path
PS C:\Users\Admin> 
```

Breadth First Search

Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search .BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level .The breadth-first search algorithm is an example of a general-graph search algorithm .Breadth-first search implemented using FIFO queue data structure.

Advantages:

BFS will provide a solution if any solution exists.

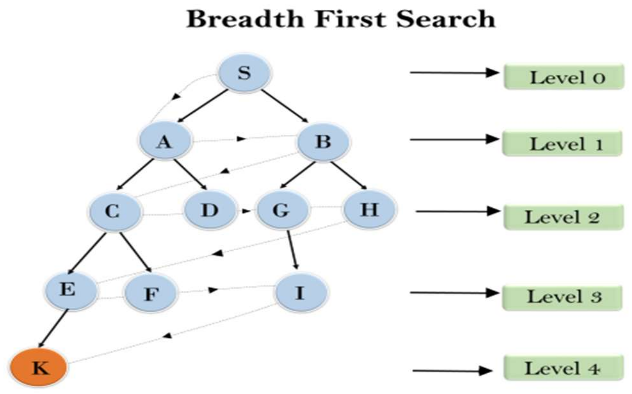
If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantages

It requires lots of memory since each level of the tree must be saved into memory to expand the next level .BFS needs lots of time if the solution is far away from the root node.

Example

S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

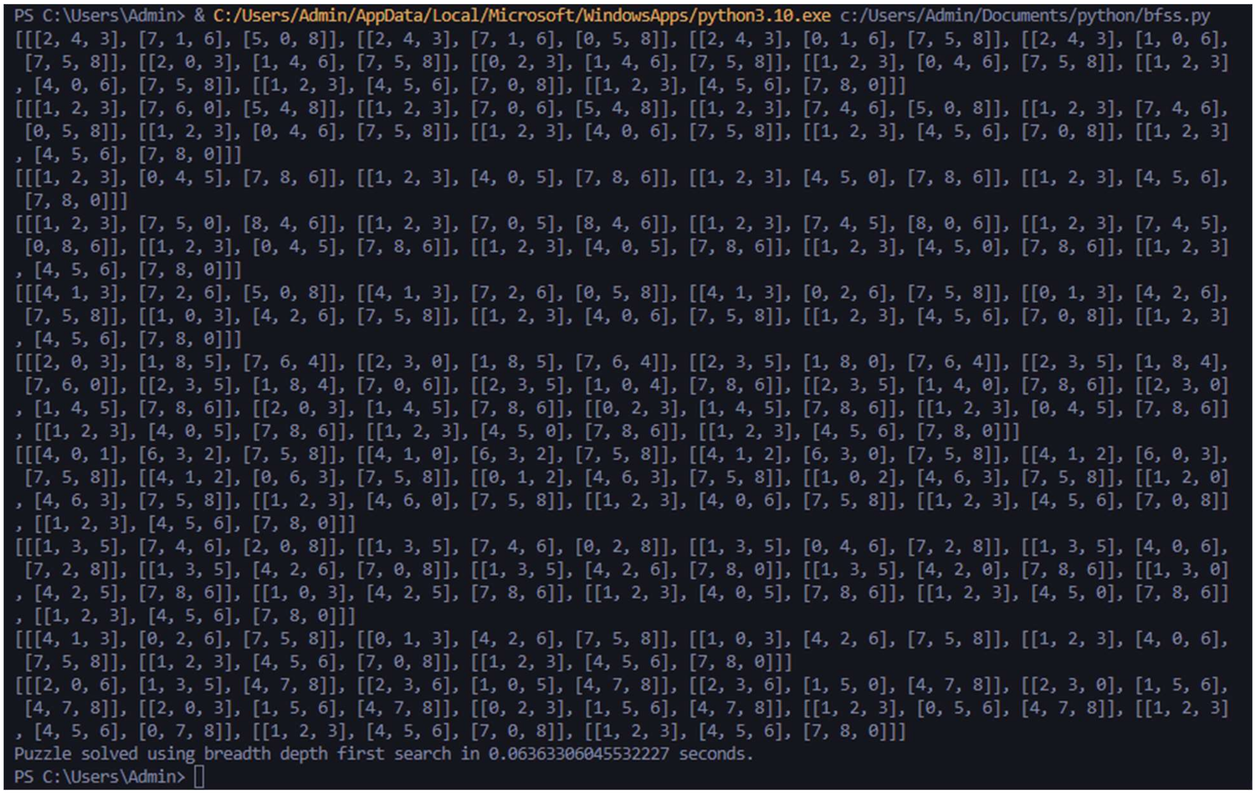
T (b) = 1+b2+b3+.......+ bd= O (bd)

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is O(bd).

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

Output:



```
PS C:\Users\Admin> & C:/Users/Admin/AppData/Local/Microsoft/WindowsApps/python3.10.exe c:/Users/Admin/Documents/python/bfss.py
[[[2, 4, 3], [7, 1, 6], [5, 0, 8]], [[2, 4, 3], [7, 1, 6], [0, 5, 8]], [[2, 4, 3], [0, 1, 6], [7, 5, 8]], [[2, 4, 3], [1, 0, 6],
 [7, 5, 8]], [[2, 0, 3], [1, 4, 6], [7, 5, 8]], [[0, 2, 3], [1, 4, 6], [7, 5, 8]], [[1, 2, 3], [0, 4, 6], [7, 5, 8]], [[1, 2, 3]
, [4, 0, 6], [7, 5, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[1, 2, 3], [7, 6, 0], [5, 4, 8]], [[1, 2, 3], [7, 0, 6], [5, 4, 8]], [[1, 2, 3], [7, 4, 6], [5, 0, 8]], [[1, 2, 3], [7, 4, 6],
 [0, 5, 8]], [[1, 2, 3], [0, 4, 6], [7, 5, 8]], [[1, 2, 3], [4, 0, 6], [7, 5, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3]
, [4, 5, 6], [7, 8, 0]]]
[[[1, 2, 3], [0, 4, 5], [7, 8, 6]], [[1, 2, 3], [4, 0, 5], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]], [[1, 2, 3], [4, 5, 6],
 [7, 8, 0]]]
[[[1, 2, 3], [7, 5, 0], [8, 4, 6]], [[1, 2, 3], [7, 0, 5], [8, 4, 6]], [[1, 2, 3], [7, 4, 5], [8, 0, 6]], [[1, 2, 3], [7, 4, 5],
 [0, 8, 6]], [[1, 2, 3], [0, 4, 5], [7, 8, 6]], [[1, 2, 3], [4, 0, 5], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]], [[1, 2, 3]
, [4, 5, 6], [7, 8, 0]]]
[[[4, 1, 3], [7, 2, 6], [5, 0, 8]], [[4, 1, 3], [7, 2, 6], [0, 5, 8]], [[4, 1, 3], [0, 2, 6], [7, 5, 8]], [[0, 1, 3], [4, 2, 6],
 [7, 5, 8]], [[1, 0, 3], [4, 2, 6], [7, 5, 8]], [[1, 2, 3], [4, 0, 6], [7, 5, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3]
, [4, 5, 6], [7, 8, 0]]]
[[[2, 0, 3], [1, 8, 5], [7, 6, 4]], [[2, 3, 0], [1, 8, 5], [7, 6, 4]], [[2, 3, 5], [1, 8, 0], [7, 6, 4]], [[2, 3, 5], [1, 8, 4],
 [7, 6, 0]], [[2, 3, 5], [1, 8, 4], [7, 0, 6]], [[2, 3, 5], [1, 0, 4], [7, 8, 6]], [[2, 3, 5], [1, 4, 0], [7, 8, 6]], [[2, 3, 0]
, [1, 4, 5], [7, 8, 6]], [[2, 0, 3], [1, 4, 5], [7, 8, 6]], [[0, 2, 3], [1, 4, 5], [7, 8, 6]], [[1, 2, 3], [0, 4, 5], [7, 8, 6]]
, [[1, 2, 3], [4, 0, 5], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[4, 0, 1], [6, 3, 2], [7, 5, 8]], [[4, 1, 0], [6, 3, 2], [7, 5, 8]], [[4, 1, 2], [6, 3, 0], [7, 5, 8]], [[4, 1, 2], [6, 0, 3],
 [7, 5, 8]], [[4, 1, 2], [0, 6, 3], [7, 5, 8]], [[0, 1, 2], [4, 6, 3], [7, 5, 8]], [[1, 0, 2], [4, 6, 3], [7, 5, 8]], [[1, 2, 0]
, [4, 6, 3], [7, 5, 8]], [[1, 2, 3], [4, 6, 0], [7, 5, 8]], [[1, 2, 3], [4, 0, 6], [7, 5, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]]
, [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[1, 3, 5], [7, 4, 6], [2, 0, 8]], [[1, 3, 5], [7, 4, 6], [0, 2, 8]], [[1, 3, 5], [0, 4, 6], [7, 2, 8]], [[1, 3, 5], [4, 0, 6],
 [7, 2, 8]], [[1, 3, 5], [4, 2, 6], [7, 0, 8]], [[1, 3, 5], [4, 2, 6], [7, 8, 0]], [[1, 3, 5], [4, 2, 0], [7, 8, 6]], [[1, 3, 0]
, [4, 2, 5], [7, 8, 6]], [[1, 0, 3], [4, 2, 5], [7, 8, 6]], [[1, 2, 3], [4, 0, 5], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]]
, [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[4, 1, 3], [0, 2, 6], [7, 5, 8]], [[0, 1, 3], [4, 2, 6], [7, 5, 8]], [[1, 0, 3], [4, 2, 6], [7, 5, 8]], [[1, 2, 3], [4, 0, 6],
 [7, 5, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[2, 0, 6], [1, 3, 5], [4, 7, 8]], [[2, 3, 6], [1, 0, 5], [4, 7, 8]], [[2, 3, 6], [1, 5, 0], [4, 7, 8]], [[2, 3, 0], [1, 5, 6],
 [4, 7, 8]], [[2, 0, 3], [1, 5, 6], [4, 7, 8]], [[0, 2, 3], [1, 5, 6], [4, 7, 8]], [[1, 2, 3], [0, 5, 6], [4, 7, 8]], [[1, 2, 3]
, [4, 5, 6], [0, 7, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
Puzzle solved using breadth depth first search in 0.06363306045532227 seconds.
PS C:\Users\Admin> 
```

Depth-first Search

Depth-first search is a recursive algorithm for traversing a tree or graph data structure .It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path .DFS uses a stack data structure for its implementation .The process of the DFS algorithm is similar to the BFS algorithm .

Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

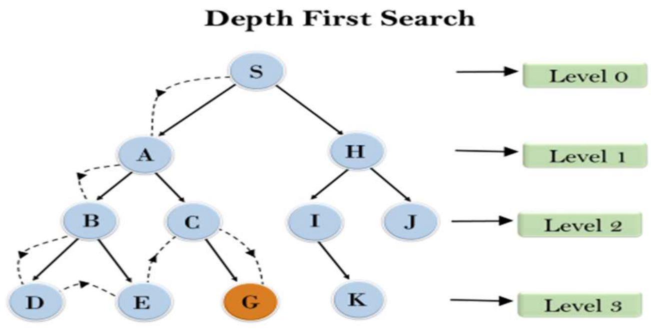
Advantage:

DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node .It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution .DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:



Root node--->Left node ----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.

Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

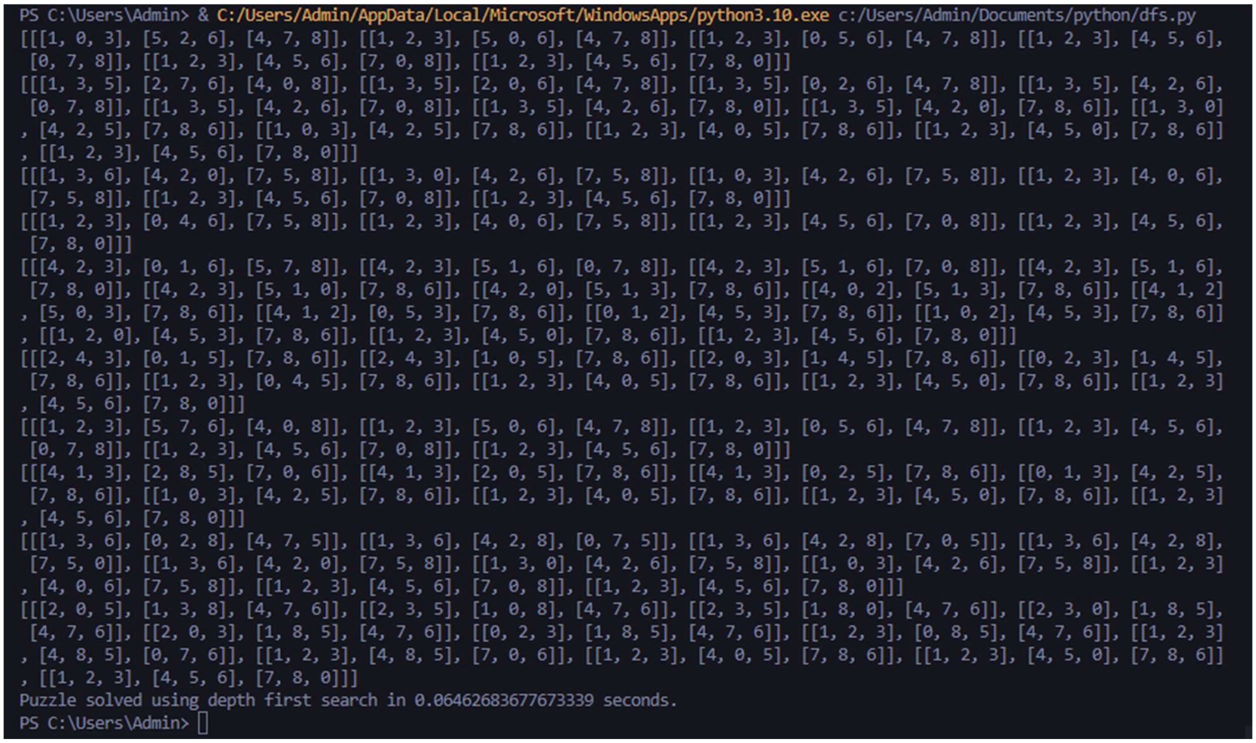
Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:T(n)= 1+ n2+ n3 +.........+ nm=O(nm)

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is O(bm).

Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Output:



```
PS C:\Users\Admin> & C:/Users/Admin/AppData/Local/Microsoft/WindowsApps/python3.10.exe c:/Users/Admin/Documents/python/dfs.py
[[[1, 0, 3], [5, 2, 6], [4, 7, 8]], [[1, 2, 3], [5, 0, 6], [4, 7, 8]], [[1, 2, 3], [0, 5, 6], [4, 7, 8]], [[1, 2, 3], [4, 5, 6],
 [0, 7, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[1, 3, 5], [2, 7, 6], [4, 0, 8]], [[1, 3, 5], [2, 0, 6], [4, 7, 8]], [[1, 3, 5], [0, 2, 6], [4, 7, 8]], [[1, 3, 5], [4, 2, 6],
 [0, 7, 8]], [[1, 3, 5], [4, 2, 6], [7, 0, 8]], [[1, 3, 5], [4, 2, 6], [7, 8, 0]], [[1, 3, 5], [4, 2, 0], [7, 8, 6]], [[1, 3, 0]
, [4, 2, 5], [7, 8, 6]], [[1, 0, 3], [4, 2, 5], [7, 8, 6]], [[1, 2, 3], [4, 0, 5], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]]
, [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[1, 3, 6], [4, 2, 0], [7, 5, 8]], [[1, 3, 0], [4, 2, 6], [7, 5, 8]], [[1, 0, 3], [4, 2, 6], [7, 5, 8]], [[1, 2, 3], [4, 0, 6],
 [7, 5, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[1, 2, 3], [0, 4, 6], [7, 5, 8]], [[1, 2, 3], [4, 0, 6], [7, 5, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6],
 [7, 8, 0]]]
[[[4, 2, 3], [0, 1, 6], [5, 7, 8]], [[4, 2, 3], [5, 1, 6], [0, 7, 8]], [[4, 2, 3], [5, 1, 6], [7, 0, 8]], [[4, 2, 3], [5, 1, 6],
 [7, 8, 0]], [[4, 2, 3], [5, 1, 0], [7, 8, 6]], [[4, 2, 0], [5, 1, 3], [7, 8, 6]], [[4, 0, 2], [5, 1, 3], [7, 8, 6]], [[4, 1, 2]
, [5, 0, 3], [7, 8, 6]], [[4, 1, 2], [0, 5, 3], [7, 8, 6]], [[0, 1, 2], [4, 5, 3], [7, 8, 6]], [[1, 0, 2], [4, 5, 3], [7, 8, 6]]
, [[1, 2, 0], [4, 5, 3], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[2, 4, 3], [0, 1, 5], [7, 8, 6]], [[2, 4, 3], [1, 0, 5], [7, 8, 6]], [[2, 0, 3], [1, 4, 5], [7, 8, 6]], [[0, 2, 3], [1, 4, 5],
 [7, 8, 6]], [[1, 2, 3], [0, 4, 5], [7, 8, 6]], [[1, 2, 3], [4, 0, 5], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]], [[1, 2, 3]
, [4, 5, 6], [7, 8, 0]]]
[[[1, 2, 3], [5, 7, 6], [4, 0, 8]], [[1, 2, 3], [5, 0, 6], [4, 7, 8]], [[1, 2, 3], [0, 5, 6], [4, 7, 8]], [[1, 2, 3], [4, 5, 6],
 [0, 7, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[4, 1, 3], [2, 8, 5], [7, 0, 6]], [[4, 1, 3], [2, 0, 5], [7, 8, 6]], [[4, 1, 3], [0, 2, 5], [7, 8, 6]], [[0, 1, 3], [4, 2, 5],
 [7, 8, 6]], [[1, 0, 3], [4, 2, 5], [7, 8, 6]], [[1, 2, 3], [4, 0, 5], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]], [[1, 2, 3]
, [4, 5, 6], [7, 8, 0]]]
[[[1, 3, 6], [0, 2, 8], [4, 7, 5]], [[1, 3, 6], [4, 2, 8], [0, 7, 5]], [[1, 3, 6], [4, 2, 8], [7, 0, 5]], [[1, 3, 6], [4, 2, 8],
 [7, 5, 0]], [[1, 3, 6], [4, 2, 0], [7, 5, 8]], [[1, 3, 0], [4, 2, 6], [7, 5, 8]], [[1, 0, 3], [4, 2, 6], [7, 5, 8]], [[1, 2, 3]
, [4, 0, 6], [7, 5, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[2, 0, 5], [1, 3, 8], [4, 7, 6]], [[2, 3, 5], [1, 0, 8], [4, 7, 6]], [[2, 3, 5], [1, 8, 0], [4, 7, 6]], [[2, 3, 0], [1, 8, 5],
 [4, 7, 6]], [[2, 0, 3], [1, 8, 5], [4, 7, 6]], [[0, 2, 3], [1, 8, 5], [4, 7, 6]], [[1, 2, 3], [0, 8, 5], [4, 7, 6]], [[1, 2, 3]
, [4, 8, 5], [0, 7, 6]], [[1, 2, 3], [4, 8, 5], [7, 0, 6]], [[1, 2, 3], [4, 0, 5], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]]
, [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
Puzzle solved using depth first search in 0.06462683677673339 seconds.
PS C:\Users\Admin> 
```

Iterative Deepening Depth-First Search

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found .This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found .This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency .The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

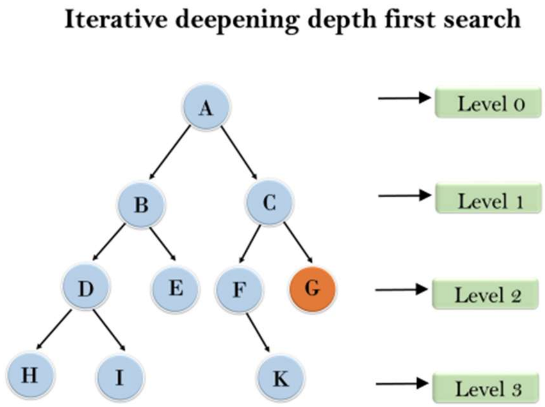
It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

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

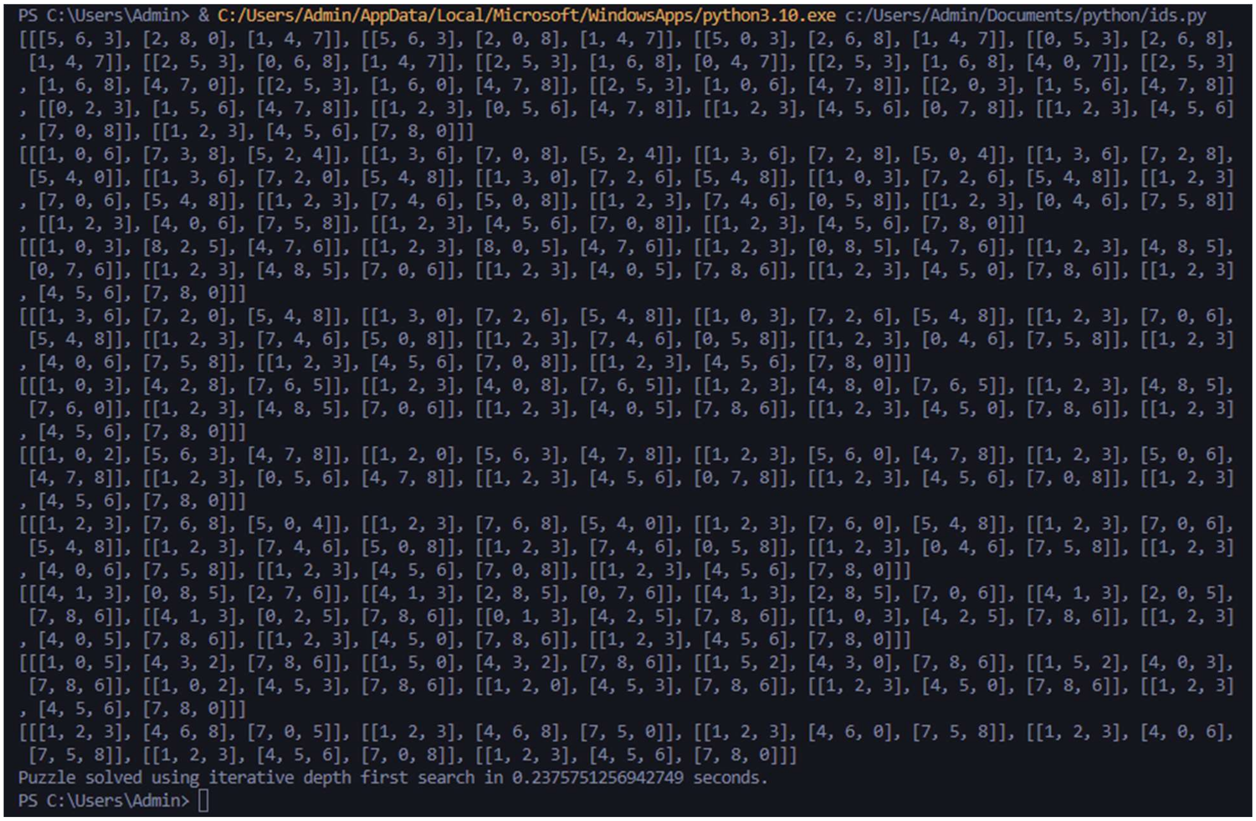
Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:



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.

Output:



```
PS C:\Users\Admin> & C:/Users/Admin/AppData/Local/Microsoft/WindowsApps/python3.10.exe c:/Users/Admin/Documents/python/ids.py
[[[5, 6, 3], [2, 8, 0], [1, 4, 7]], [[5, 6, 3], [2, 0, 8], [1, 4, 7]], [[5, 0, 3], [2, 6, 8], [1, 4, 7]], [[0, 5, 3], [2, 6, 8],
 [1, 4, 7]], [[2, 5, 3], [0, 6, 8], [1, 4, 7]], [[2, 5, 3], [1, 6, 8], [0, 4, 7]], [[2, 5, 3], [1, 6, 8], [4, 0, 7]], [[2, 5, 3]
, [1, 6, 8], [4, 7, 0]], [[2, 5, 3], [1, 6, 0], [4, 7, 8]], [[2, 5, 3], [1, 0, 6], [4, 7, 8]], [[2, 0, 3], [1, 5, 6], [4, 7, 8]]
, [[0, 2, 3], [1, 5, 6], [4, 7, 8]], [[1, 2, 3], [0, 5, 6], [4, 7, 8]], [[1, 2, 3], [4, 5, 6], [0, 7, 8]], [[1, 2, 3], [4, 5, 6]
, [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[1, 0, 6], [7, 3, 8], [5, 2, 4]], [[1, 3, 6], [7, 0, 8], [5, 2, 4]], [[1, 3, 6], [7, 2, 8], [5, 0, 4]], [[1, 3, 6], [7, 2, 8],
 [5, 4, 0]], [[1, 3, 6], [7, 2, 0], [5, 4, 8]], [[1, 3, 0], [7, 2, 6], [5, 4, 8]], [[1, 0, 3], [7, 2, 6], [5, 4, 8]], [[1, 2, 3]
, [7, 0, 6], [5, 4, 8]], [[1, 2, 3], [7, 4, 6], [5, 0, 8]], [[1, 2, 3], [7, 4, 6], [0, 5, 8]], [[1, 2, 3], [0, 4, 6], [7, 5, 8]]
, [[1, 2, 3], [4, 0, 6], [7, 5, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[1, 0, 3], [8, 2, 5], [4, 7, 6]], [[1, 2, 3], [8, 0, 5], [4, 7, 6]], [[1, 2, 3], [0, 8, 5], [4, 7, 6]], [[1, 2, 3], [4, 8, 5],
 [0, 7, 6]], [[1, 2, 3], [4, 8, 5], [7, 0, 6]], [[1, 2, 3], [4, 0, 5], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]], [[1, 2, 3]
, [4, 5, 6], [7, 8, 0]]]
[[[1, 3, 6], [7, 2, 0], [5, 4, 8]], [[1, 3, 0], [7, 2, 6], [5, 4, 8]], [[1, 0, 3], [7, 2, 6], [5, 4, 8]], [[1, 2, 3], [7, 0, 6],
 [5, 4, 8]], [[1, 2, 3], [7, 4, 6], [5, 0, 8]], [[1, 2, 3], [7, 4, 6], [0, 5, 8]], [[1, 2, 3], [0, 4, 6], [7, 5, 8]], [[1, 2, 3]
, [4, 0, 6], [7, 5, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[1, 0, 3], [4, 2, 8], [7, 6, 5]], [[1, 2, 3], [4, 0, 8], [7, 6, 5]], [[1, 2, 3], [4, 8, 0], [7, 6, 5]], [[1, 2, 3], [4, 8, 5],
 [7, 6, 0]], [[1, 2, 3], [4, 8, 5], [7, 0, 6]], [[1, 2, 3], [4, 0, 5], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]], [[1, 2, 3]
, [4, 5, 6], [7, 8, 0]]]
[[[1, 0, 2], [5, 6, 3], [4, 7, 8]], [[1, 2, 0], [5, 6, 3], [4, 7, 8]], [[1, 2, 3], [5, 6, 0], [4, 7, 8]], [[1, 2, 3], [5, 0, 6],
 [4, 7, 8]], [[1, 2, 3], [0, 5, 6], [4, 7, 8]], [[1, 2, 3], [4, 5, 6], [0, 7, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3]
, [4, 5, 6], [7, 8, 0]]]
[[[1, 2, 3], [7, 6, 8], [5, 0, 4]], [[1, 2, 3], [7, 6, 8], [5, 4, 0]], [[1, 2, 3], [7, 6, 0], [5, 4, 8]], [[1, 2, 3], [7, 0, 6],
 [5, 4, 8]], [[1, 2, 3], [7, 4, 6], [5, 0, 8]], [[1, 2, 3], [7, 4, 6], [0, 5, 8]], [[1, 2, 3], [0, 4, 6], [7, 5, 8]], [[1, 2, 3]
, [4, 0, 6], [7, 5, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[4, 1, 3], [0, 8, 5], [2, 7, 6]], [[4, 1, 3], [2, 8, 5], [0, 7, 6]], [[4, 1, 3], [2, 8, 5], [7, 0, 6]], [[4, 1, 3], [2, 0, 5],
 [7, 8, 6]], [[4, 1, 3], [0, 2, 5], [7, 8, 6]], [[0, 1, 3], [4, 2, 5], [7, 8, 6]], [[1, 0, 3], [4, 2, 5], [7, 8, 6]], [[1, 2, 3]
, [4, 0, 5], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
[[[1, 0, 5], [4, 3, 2], [7, 8, 6]], [[1, 5, 0], [4, 3, 2], [7, 8, 6]], [[1, 5, 2], [4, 3, 0], [7, 8, 6]], [[1, 5, 2], [4, 0, 3],
 [7, 8, 6]], [[1, 0, 2], [4, 5, 3], [7, 8, 6]], [[1, 2, 0], [4, 5, 3], [7, 8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]], [[1, 2, 3]
, [4, 5, 6], [7, 8, 0]]]
[[[1, 2, 3], [4, 6, 8], [7, 0, 5]], [[1, 2, 3], [4, 6, 8], [7, 5, 0]], [[1, 2, 3], [4, 6, 0], [7, 5, 8]], [[1, 2, 3], [4, 0, 6],
 [7, 5, 8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
Puzzle solved using iterative depth first search in 0.2375751256942749 seconds.
PS C:\Users\Admin> 
```

Best-first Search Algorithm (Greedy Search):

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.f(n)= g(n).

Where, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

Algorithm:

Step 1: Place the starting node into the OPEN list.

Step 2: If the OPEN list is empty, Stop and return failure.

Step 3: Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.

Step 4: Expand the node n, and generate the successors of node n.

Step 5: Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.

Step 6: For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.

Step 7: Return to Step 2.

Advantages:

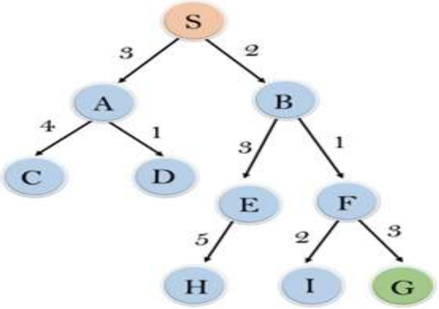
Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms .This algorithm is more efficient than BFS and DFS algorithms.

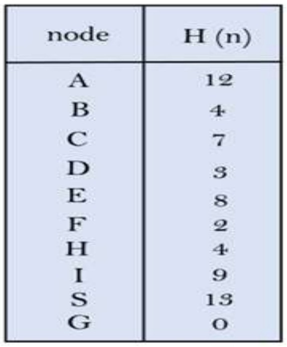
Disadvantages:

It can behave as an unguided depth-first search in the worst case scenario. It can get stuck in a loop as DFS .This algorithm is not optimal.

Example:

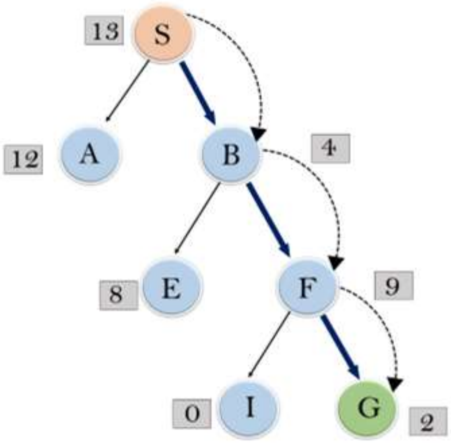
Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.





node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are OPEN and CLOSED Lists. Following are the iteration for traversing the above example.



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]

: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]

: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: S----> B----->F----> G

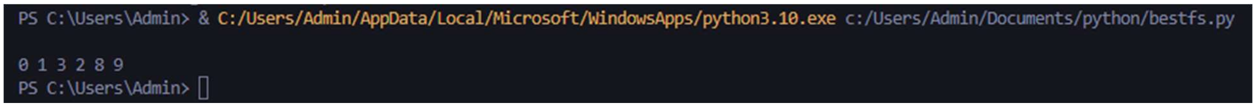
Time Complexity: The worst case time complexity of Greedy best first search is O(bm).

Space Complexity: The worst case space complexity of Greedy best first search is O(bm). Where, m is the maximum depth of the search space.

Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

Output:

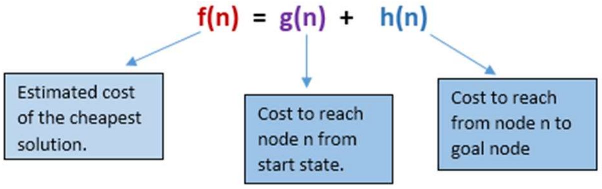


```
PS C:\Users\Admin> & C:/Users/Admin/AppData/Local/Microsoft/WindowsApps/python3.10.exe c:/Users/Admin/Documents/python/bestfs.py

0 1 3 2 8 9
PS C:\Users\Admin> 
```

A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a fitness number.


f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to Step 2.

Advantages:

A* search algorithm is the best algorithm than other search algorithms .A* search algorithm is optimal and complete .This algorithm can solve very complex problems.

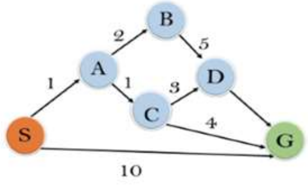
Disadvantages:

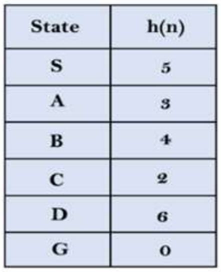
It does not always produce the shortest path as it mostly based on heuristics and approximation .A* search algorithm has some complexity issues .The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Example:

In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

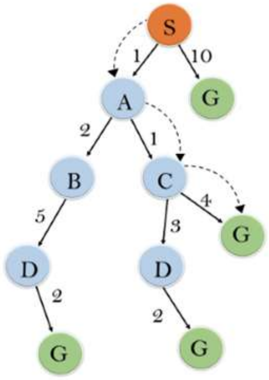
Here we will use OPEN and CLOSED list.





State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution:



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as S--->A--->C--->G it provides the optimal path with cost 6.

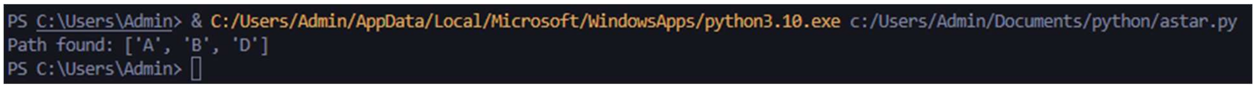
Points to remember:

A* algorithm returns the path which occurred first, and it does not search for all remaining paths.

The efficiency of A* algorithm depends on the quality of heuristic.

A* algorithm expands all nodes which satisfy the condition f(n)

Output:



```
PS C:\Users\Admin> & C:/Users/Admin/AppData/Local/Microsoft/WindowsApps/python3.10.exe c:/Users/Admin/Documents/python/astar.py
Path found: ['A', 'B', 'D']
PS C:\Users\Admin> 
```

Hill Climbing Search Algorithm

Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value .Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman .It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that .A node of hill climbing algorithm has two components which are state and value .Hill Climbing is mostly used when a good heuristic is available .In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing Algorithm:

Generate and Test variant: Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.

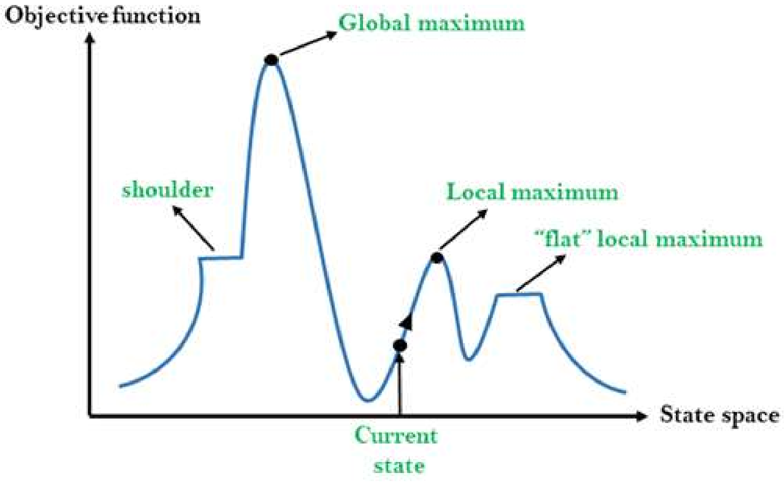
Greedy approach: Hill-climbing algorithm search moves in the direction which optimizes the cost.

No backtracking: It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

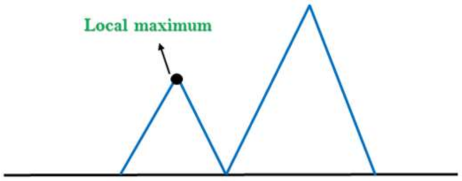
The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.

On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.



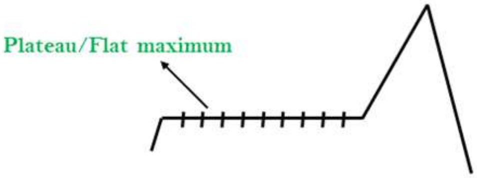
Problems in Hill Climbing Algorithm:

1. **Local Maximum**: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.



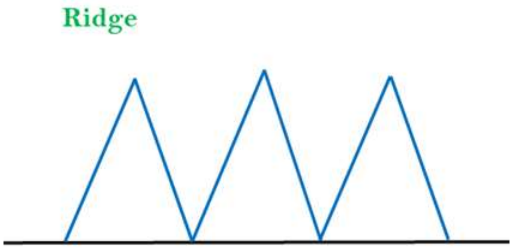
Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.

2. **Plateau**: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.



Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.

3. Ridges: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.



Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.

Hebbian Learning Algorithm

Hebb Network was stated by Donald Hebb in 1949. According to Hebb's rule, the weights are found to increase proportionately to the product of input and output. It means that in a Hebb network if two neurons are interconnected then the weights associated with these neurons can be increased by changes in the synaptic gap.

This network is suitable for bipolar data. The Hebbian learning rule is generally applied to logic gates.

The weights are updated as:

W (new) = w (old) + x*y

Training Algorithm For Hebbian Learning Rule

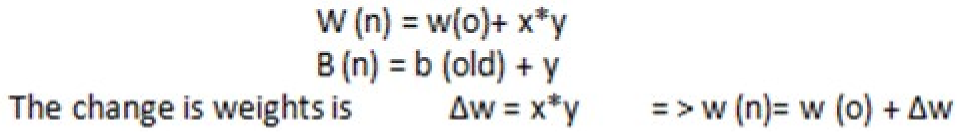
The training steps of the algorithm are as follows:

1. Initially, the weights are set to zero, i.e. w =0 for all inputs i =1 to n and n is the total number of input neurons.

2. Let s be the output. The activation function for inputs is generally set as an identity function.

3. The activation function for output is also set to y= t.

4. The weight adjustments and bias are adjusted to:



W (n) = w(o)+ x*y
B (n) = b (old) + y
The change is weights is Δw = x*y => w (n)= w (o) + Δw

5. The steps 2 to 4 are repeated for each input vector and output.

Example Of Hebbian Learning Rule

Let us implement logical AND function with bipolar inputs using Hebbian Learning

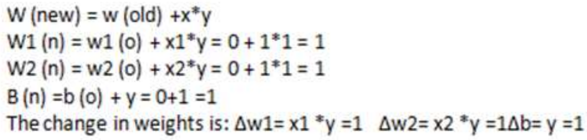
X1 and X2 are inputs, b is the bias taken as 1, the target value is the output of logical AND operation over inputs.

Input	Input	Bias	Target
x1	**x2**	**b**	**Y**
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1

1. Initially, the weights are set to zero and bias is also set as zero.

W1=w2=b=0

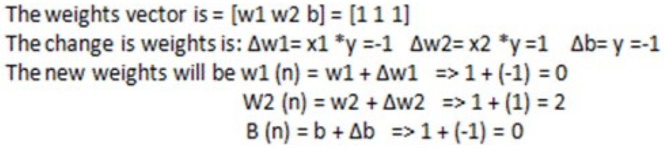
2. First input vector is taken as [x1 x2 b] = [1 1 1] and target value is 1.



W (new) = w (old) +x*y
W1 (n) = w1 (o) + x1*y = 0 + 1*1 = 1
W2 (n) = w2 (o) + x2*y = 0 + 1*1 = 1
B (n) =b (o) + y = 0+1 =1
The change in weights is: Δw1= x1 *y =1   Δw2= x2 *y =1Δb= y =1

3. The above weights are the final new weights. When the second input is passed, these become the initial weights.

4. Take the second input = [1 -1 1]. The target is -1.

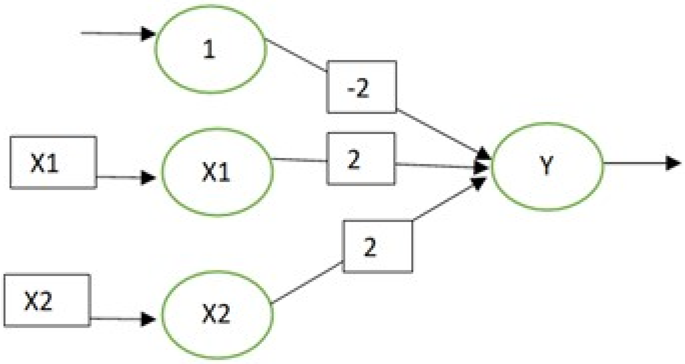


The weights vector is = [w1 w2 b] = [1 1 1]
The change is weights is: Δw1= x1 *y =-1   Δw2= x2 *y =1   Δb= y =-1
The new weights will be w1 (n) = w1 + Δw1   => 1 + (-1) = 0
W2 (n) = w2 + Δw2   => 1 + (1) = 2
B (n) = b + Δb   => 1 + (-1) = 0

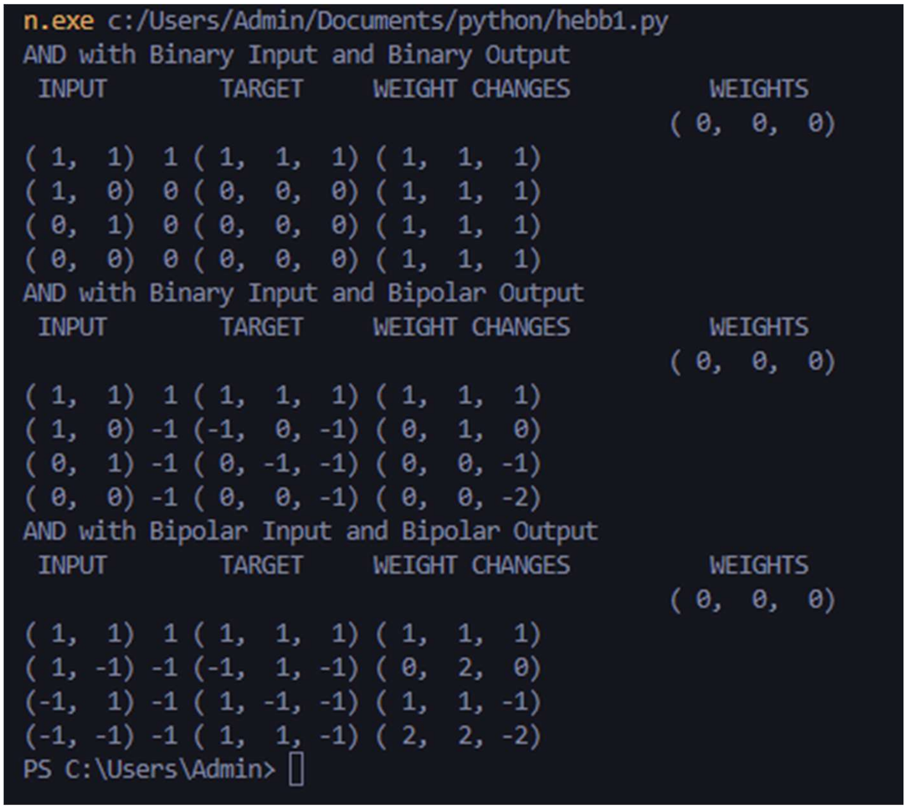
5. Similarly, the other inputs and weights are calculated.

Inputs	Bias	Target output	Weight	Bias changes	New weights	b
x1 x2	b	y	changes	b	w1 w2	
			w1 w2			
1 1	1	1	1 1	1	1 1	1
1 -1	1	-1	-1 1	-1	0 2	0
-1 1	1	-1	1 -1	-1	1 1	-1
-1 -1	1	-1	1 1	-1	2 2	-2

Hebb Net for AND Function



Output:



```
n.exe c:/Users/Admin/Documents/python/hebb1.py
AND with Binary Input and Binary Output
 INPUT        TARGET     WEIGHT CHANGES          WEIGHTS
                                              ( 0,  0,  0)

( 1,  1)  1 ( 1,  1,  1) ( 1,  1,  1)
( 1,  0)  0 ( 0,  0,  0) ( 1,  1,  1)
( 0,  1)  0 ( 0,  0,  0) ( 1,  1,  1)
( 0,  0)  0 ( 0,  0,  0) ( 1,  1,  1)
AND with Binary Input and Bipolar Output
 INPUT        TARGET     WEIGHT CHANGES          WEIGHTS
                                              ( 0,  0,  0)

( 1,  1)  1 ( 1,  1,  1) ( 1,  1,  1)
( 1,  0) -1 (-1,  0, -1) ( 0,  1,  0)
( 0,  1) -1 ( 0, -1, -1) ( 0,  0, -1)
( 0,  0) -1 ( 0,  0, -1) ( 0,  0, -2)
AND with Bipolar Input and Bipolar Output
 INPUT        TARGET     WEIGHT CHANGES          WEIGHTS
                                              ( 0,  0,  0)

( 1,  1)  1 ( 1,  1,  1) ( 1,  1,  1)
( 1, -1) -1 (-1,  1, -1) ( 0,  2,  0)
(-1,  1) -1 ( 1, -1, -1) ( 1,  1, -1)
(-1, -1) -1 ( 1,  1, -1) ( 2,  2, -2)
PS C:\Users\Admin>
```

Perceptron Learning Algorithm

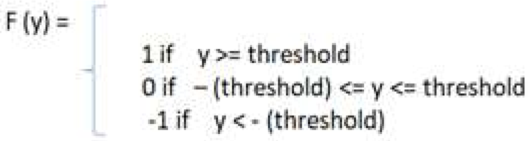
Perceptron Networks are single-layer feed-forward networks. These are also called Single Perceptron Networks. The Perceptron consists of an input layer, a hidden layer, and output layer.

The input layer is connected to the hidden layer through weights which may be inhibitory or excitery or zero (-1, +1 or 0). The activation function used is a binary step function for the input layer and the hidden layer.

The output is

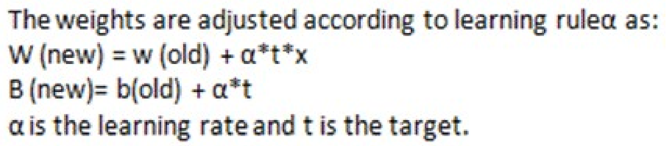
Y= f (y)

The activation function is:



F (y) =
1 if y >= threshold
0 if – (threshold) <= y <= threshold
-1 if y < - (threshold)

The weight updation takes place between the hidden layer and the output layer to match the target output. The error is calculated based on the actual output and the desired output.



The weights are adjusted according to learning ruleα as:
W (new) = w (old) + α*t*x
B (new)= b(old) + α*t
α is the learning rate and t is the target.

If the output matches the target then no weight updation takes place. The weights are initially set to 0 or 1 and adjusted successively till an optimal solution is found .The weights in the network can be set to any values initially. The Perceptron learning will converge to weight vector that gives correct output for all input training pattern and this learning happens in a finite number of steps .The Perceptron rule can be used for both binary and bipolar inputs.

Learning Rule for Single Output Perceptron

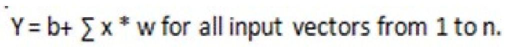
1) Let there be "n" training input vectors and x (n) and t (n) are associated with the target values.

2) Initialize the weights and bias. Set them to zero for easy calculation.

3) Let the learning rate be 1.

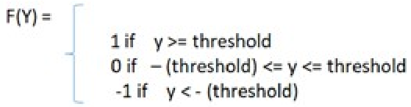
4) The input layer has identity activation function so x (i)= s (i).

5) To calculate the output of the network:



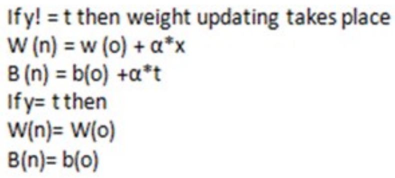
Y = b+ ∑ x * w for all input vectors from 1 to n.

6) The activation function is applied over the net input to obtain an output.



F(Y) =
1 if y >= threshold
0 if – (threshold) <= y <= threshold
-1 if y < - (threshold)

7) Now based on the output, compare the desired target value (t) and the actual output.



If y! = t then weight updating takes place
W (n) = w (o) + α*x
B (n) = b(o) +α*t
If y= t then
W(n)= W(o)
B(n)= b(o)

8) Continue the iteration until there is no weight change. Stop once this condition is achieved.

Learning Rule for Multiple Output Perceptron

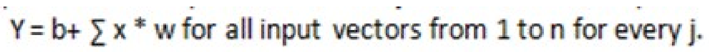
1) Let there be "n" training input vectors and x (n) and t (n) are associated with the target values.

2) Initialize the weights and bias. Set them to zero for easy calculation.

3) Let the learning rate be 1.

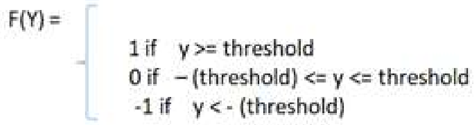
4) The input layer has identity activation function so x (i)= s (i).

5) To calculate the output of each output vector from j= 1 to m, the net input is:



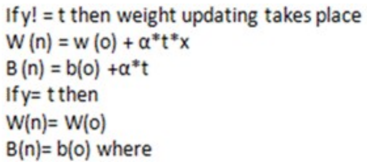
Y = b+ ∑ x * w for all input vectors from 1 to n for every j.

6) The activation function is applied over the net input to obtain an output.



F(Y) =
1 if y >= threshold
0 if – (threshold) <= y <= threshold
-1 if y < - (threshold)

7) Now based on the output, compare the desired target value (t) and the actual output and make weight adjustments.



If y! = t then weight updating takes place
W (n) = w (o) + α*t*x
B (n) = b(o) +α*t
If y= t then
W(n)= W(o)
B(n)= b(o) where

w is the weight vector of the connection links between ith input and jth output neuron and t is the target output for the output unit j.

8) Continue the iteration until there is no weight change. Stop once this condition is achieved.

Example Of Perceptron Learning Rule

Implementation of AND function using a Perceptron network for bipolar inputs and output.

The input pattern will be x1, x2 and bias b. Let the initial weights be 0 and bias be 0. The threshold is set to zero and the learning rate is 1.

AND Gate

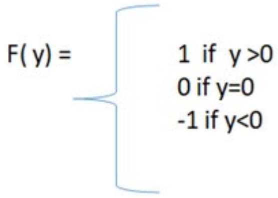
X1	X2	Target
1	1	1
1	-1	-1
-1	1	-1
-1	-1	-1

1) X1=1 , X2= 1 and target output = 1

W1=w2=wb=0 and x1=x2=b=1, t=1

Net input= y =b + x1*w1+x2*w2 = 0+1*0 +1*0 =0

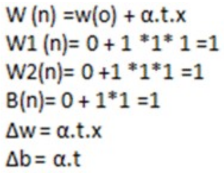
As threshold is zero therefore:



$$F(y) = \begin{cases} 1 \text{ if } y>0 \\ 0 \text{ if } y=0 \\ -1 \text{ if } y<0 \end{cases}$$

From here we get, output = 0. Now check if output (y) = target (t).

y = 0 but t= 1 which means that these are not same, hence weight updation takes place.



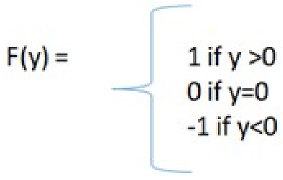
W (n) =w(o) + α.t.x
W1 (n)= 0 + 1 *1* 1 =1
W2(n)= 0 +1 *1*1 =1
B(n)= 0 + 1*1 =1
Δw = α.t.x
Δb = α.t

The new weights are 1, 1, and 1 after the first input vector is presented.

2) X1= 1 X2= -1 , b= 1 and target = -1, W1=1 ,W2=2, Wb=1

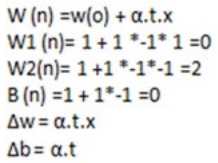
Net input= y =b + x1*w1+x2*w2 = 1+1*1 + (-1)*1 =1

The net output for input= 1 will be 1 from:



$$F(y) = \begin{cases} 1 \text{ if } y>0 \\ 0 \text{ if } y=0 \\ -1 \text{ if } y<0 \end{cases}$$

Therefore again, target = -1 does not match with the actual output =1. Weight updates take place.



$W (n) = w(o) + \alpha.t.x$
$W1 (n) = 1 + 1 * -1 * 1 = 0$
$W2(n) = 1 + 1 * -1 * -1 = 2$
$B (n) = 1 + 1 * -1 = 0$
$\Delta w = \alpha.t.x$
$\Delta b = \alpha.t$

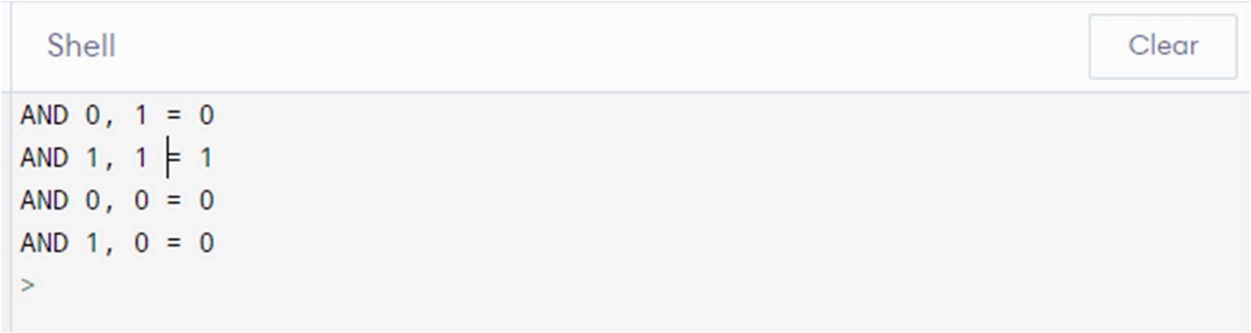
Now new weights are w1 = 0 w2 =2 and wb =0

Similarly, by continuing with the next set of inputs, we get the following table:

X1	X2	b	t	yin	Y	w1	w2	b	W1	W2	wb
EPOCH 1											
1	1	1	1	0	0	1	1	1	1	1	1
1	-1	1	-1	1	1	-1	1	-1	0	2	0
-1	1	1	-1	2	1	1	-1	-1	1	1	-1
-1	-1	1	-1	-3	-1	0	0	0	1	1	-1
EPOCH 2											
1	1	1	1	1	1	0	0	0	1	1	-1
1	-1	1	-1	-1	-1	0	0	0	1	1	-1
-1	1	1	-1	-1	-1	0	0	0	1	1	-1
-1	-1	1	-1	-3	-1	0	0	0	1	1	-1

The EPOCHS are the cycle of input patterns fed to the system until there is no weight change required and the iteration stops.

Output:



```
AND 0, 1 = 0
AND 1, 1 = 1
AND 0, 0 = 0
AND 1, 0 = 0
>
```

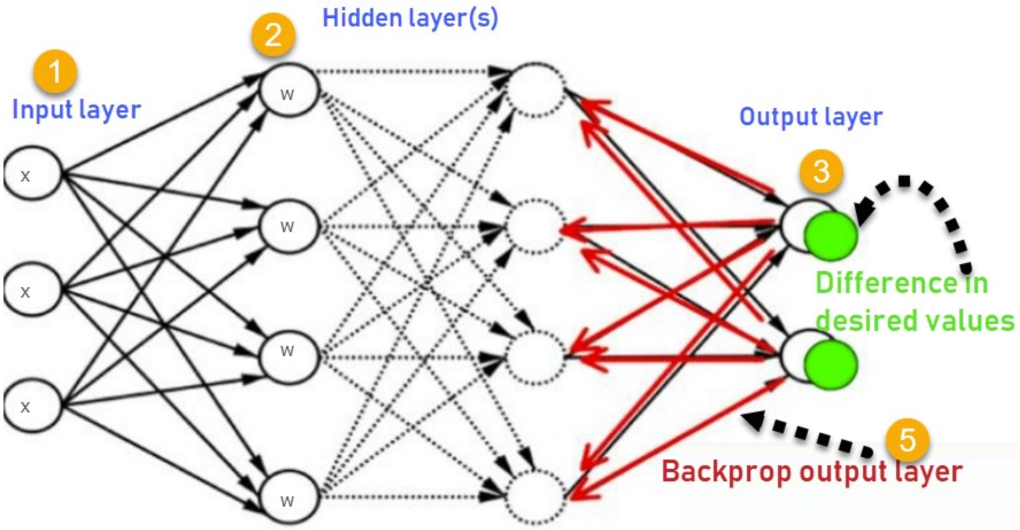
Backpropagation in Neural Network

Backpropagation is the essence of neural network training. It is the method of fine-tuning the weights of a neural network based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and make the model reliable by increasing its generalization .Backpropagation in neural network is a short form for “backward propagation of errors.” It is a standard method of training artificial neural networks. This method helps calculate the gradient of a loss function with respect to all the weights in the network.

How Backpropagation Algorithm Works

The Back propagation algorithm in neural network computes the gradient of the loss function for a single weight by the chain rule. It efficiently computes one layer at a time, unlike a native direct computation. It computes the gradient, but it does not define how the gradient is used. It generalizes the computation in the delta rule.

Consider the following Back propagation neural network example diagram to understand:



Inputs X, arrive through the preconnected path

Input is modeled using real weights W. The weights are usually randomly selected.

Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.

Calculate the error in the outputs

ErrorB= Actual Output – Desired Output

Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.

Keep repeating the process until the desired output is achieved

Advantages:

Backpropagation is fast, simple and easy to program

It has no parameters to tune apart from the numbers of input

It is a flexible method as it does not require prior knowledge about the network

It is a standard method that generally works well

It does not need any special mention of the features of the function to be learned.

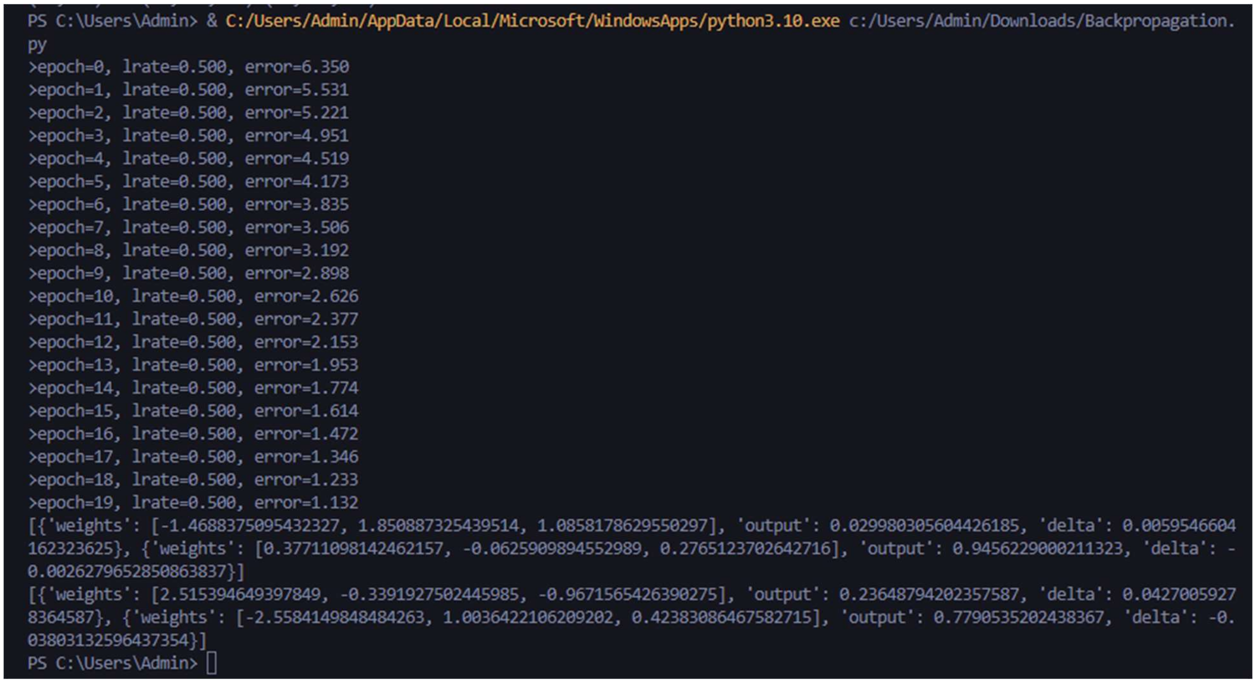
Disadvantages:

The actual performance of backpropagation on a specific problem is dependent on the input data.

Back propagation algorithm in data mining can be quite sensitive to noisy data

You need to use the matrix-based approach for backpropagation instead of mini-batch.

Output



```
PS C:\Users\Admin> & C:/Users/Admin/AppData/Local/Microsoft/WindowsApps/python3.10.exe c:/Users/Admin/Downloads/Backpropagation.
py
>epoch=0, lrate=0.500, error=6.350
>epoch=1, lrate=0.500, error=5.531
>epoch=2, lrate=0.500, error=5.221
>epoch=3, lrate=0.500, error=4.951
>epoch=4, lrate=0.500, error=4.519
>epoch=5, lrate=0.500, error=4.173
>epoch=6, lrate=0.500, error=3.835
>epoch=7, lrate=0.500, error=3.506
>epoch=8, lrate=0.500, error=3.192
>epoch=9, lrate=0.500, error=2.898
>epoch=10, lrate=0.500, error=2.626
>epoch=11, lrate=0.500, error=2.377
>epoch=12, lrate=0.500, error=2.153
>epoch=13, lrate=0.500, error=1.953
>epoch=14, lrate=0.500, error=1.774
>epoch=15, lrate=0.500, error=1.614
>epoch=16, lrate=0.500, error=1.472
>epoch=17, lrate=0.500, error=1.346
>epoch=18, lrate=0.500, error=1.233
>epoch=19, lrate=0.500, error=1.132
[{'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297], 'output': 0.029980305604426185, 'delta': 0.0059546604
162323625}, {'weights': [0.37711098142462157, -0.0625909894552989, 0.2765123702642716], 'output': 0.9456229000211323, 'delta': -
0.0026279652850863837}]
[{'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275], 'output': 0.23648794202357587, 'delta': 0.0427005927
8364587}, {'weights': [-2.5584149848484263, 1.0036422106209202, 0.42383086467582715], 'output': 0.7790535202438367, 'delta': -0.
03803132596437354}]
PS C:\Users\Admin> 
```

Genetic Algorithm

A genetic algorithm is used to solve complicated problems with a greater number of variables & possible outcomes/solutions. The combinations of different solutions are passed through the Darwinian based algorithm to find the best solutions. The poorer solutions are then replaced with the offspring of good solutions.

It all works on the Darwinian theory, where only the fittest individuals are chosen for reproduction. The various solutions are considered the elements of the population, and only the fittest solutions are allowed to reproduce (to create better solutions). Genetic algorithms help in optimizing the solutions to any particular problem.

The whole process of genetic algorithms is a computer program simulation in which the attributes of the problem & solution are treated as the attributes of the Darwinian theory. The basic processes which are involved in genetic algorithms are as follows:

A population of solutions is built to any particular problem. The elements of the population compete with each other to find out the fittest one.

The elements of the population that are fit are only allowed to create offspring (better solutions).

The genes from the fittest parents (solutions) create a better offspring. Thus, future solutions will be better and sustainable.

The working of a genetic algorithm in AI is as follows:

The components of the population, i.e., elements, are termed as genes in genetic algorithms in AI. These genes form an individual in the population (also termed as a chromosome).

A search space is created in which all the individuals are accumulated. All the individuals are coded within a finite length in the search space.

Each individual in the search space (population) is given a fitness score, which tells its ability to compete with other individuals.

All the individuals with their respective fitness scores are sought & maintained by the genetic algorithm & the individuals with high fitness scores are given a chance to reproduce.

The new offspring are having better ‘partial solutions’ as compared to their parents. Genetic algorithms also keep the space of the search space dynamic for accumulating the new solutions (offspring).

This process is repeated until the offsprings do not have any new attributes/features than their parents (convergence). The population converges at the end, and only the fittest solutions remain along with their offspring (better solutions). The fitness score of new individuals in the population (offspring) are also calculated.

Benefits and Uses of Genetic Algorithms

The solutions created through genetic algorithms are strong & reliable as compared to other solutions.

They increase the size of solutions as solutions can be optimized over a large search scale. This algorithm also can manage a large population.

The solutions produced by genetic algorithms do not deviate much on slightly changing the input. They can handle a little bit of noise.

Genetic algorithms have a stochastic distribution that follows probabilistic transition rules, making them hard to predict but easy to analyze.

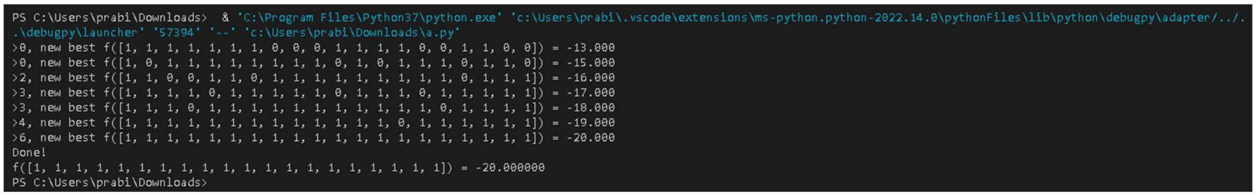
Genetic algorithms can also perform in noisy environments. It can also work in case of complex & discrete problems.

Due to their effectiveness, genetic algorithms have many applications like neural networks, fuzzy logic, code-breaking, filtering & signal processing.

Limitations:

The first limitation is that these algorithms are computationally expensive because the evaluation of each individual necessitates the training of a model. The second problem faced with these algorithms is that they are ineffective in tackling minor problems. Another issue with these algorithms is that their stochastic nature can take a long time to converge, and improper implementation may cause the algorithm to converge to an unsatisfactory result. Also, in a genetic algorithm, the quality of the final answer is not guaranteed. In these algorithms, the repetitive calculation of the fittest values may cause specific issues with computing hurdles.

Output:



```
PS C:\Users\prabi\Downloads>  & 'C:\Program Files\Python37\python.exe' 'c:\Users\prabi\.vscode\extensions\ms-python.python-2022.14.0\pythonFiles\lib\python\debugpy\adapter/../.
.\debugpy\launcher' '57394' '--' 'c:\Users\prabi\Downloads\a.py'
>0, new best f([1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0]) = -13.000
>0, new best f([1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0]) = -15.000
>2, new best f([1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]) = -16.000
>3, new best f([1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1]) = -17.000
>3, new best f([1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1]) = -18.000
>4, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]) = -19.000
>6, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000
Done!
f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000000
PS C:\Users\prabi\Downloads>
```

Naive Bayes Classifier Algorithm

Naive Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems .It is mainly used in text classification that includes a high-dimensional training dataset .Naive Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions .It is a probabilistic classifier, which means it predicts on the basis of the probability of an object .Some popular examples of Naive Bayes Algorithm are spam filtration, Sentimental analysis, and classifying articles.

Bayes' Theorem:

Bayes' theorem is also known as Bayes' Rule or Bayes' law, which is used to determine the probability of a hypothesis with prior knowledge. It depends on the conditional probability.

The formula for Bayes' theorem is given as:

$$\mathrm{P(A|B)}= \frac{P(B|A)P(A)}{P(B)}$$

Where,

P(A|B) is Posterior probability: Probability of hypothesis A on the observed event B.

P(B|A) is Likelihood probability: Probability of the evidence given that the probability of a hypothesis is true.

P(A) is Prior Probability: Probability of hypothesis before observing the evidence.

P(B) is Marginal Probability: Probability of Evidence.

Advantages :

Naive Bayes is one of the fast and easy ML algorithms to predict a class of datasets.

It can be used for Binary as well as Multi-class Classifications.

It performs well in Multi-class predictions as compared to the other Algorithms.

It is the most popular choice for text classification problems.

Disadvantages :

Naive Bayes assumes that all features are independent or unrelated, so it cannot learn the relationship between features.

Applications of Naive Bayes Classifier:

It is used for Credit Scoring .It is used in medical data classification.

It can be used in real-time predictions because Naïve Bayes Classifier is an eager learner.

It is used in Text classification such as Spam filtering and Sentiment analysis.

Parse Tree

Theory:

A parse tree is a really just a “diagrammed” form of a sentence; that sentence could be written in any language, which means that it could adhere to any set of grammatical rules.

Sentence diagramming involves breaking up a single sentence into its smallest, most distinct parts. If we think about parse trees from the persepctive of diagramming sentences we’ll begin to quickly realize that, depending on the grammar and language of a sentence, a parse tree could really be constructed in a multitude of different ways!

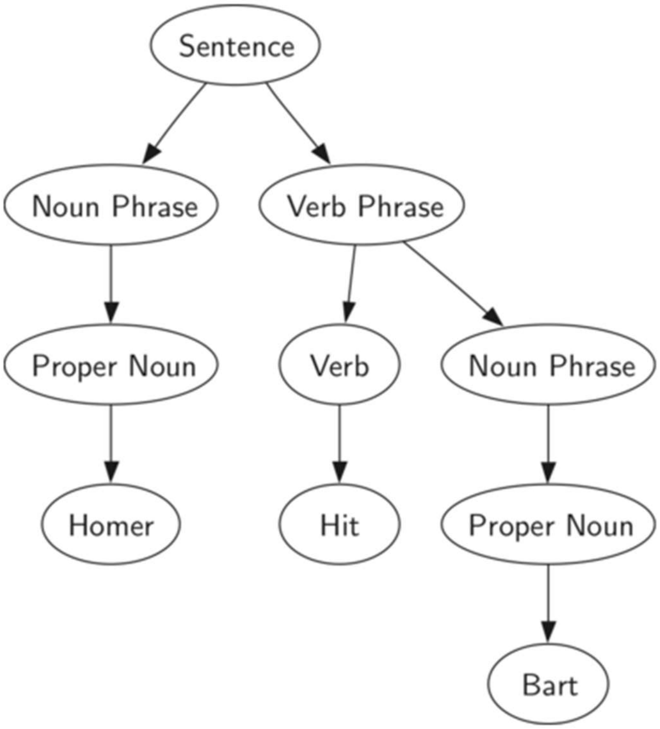


Figure 1: A Parse Tree for a Simple Sentence

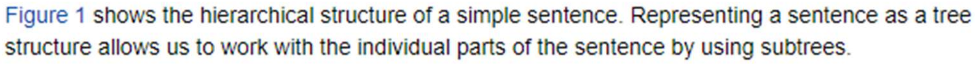
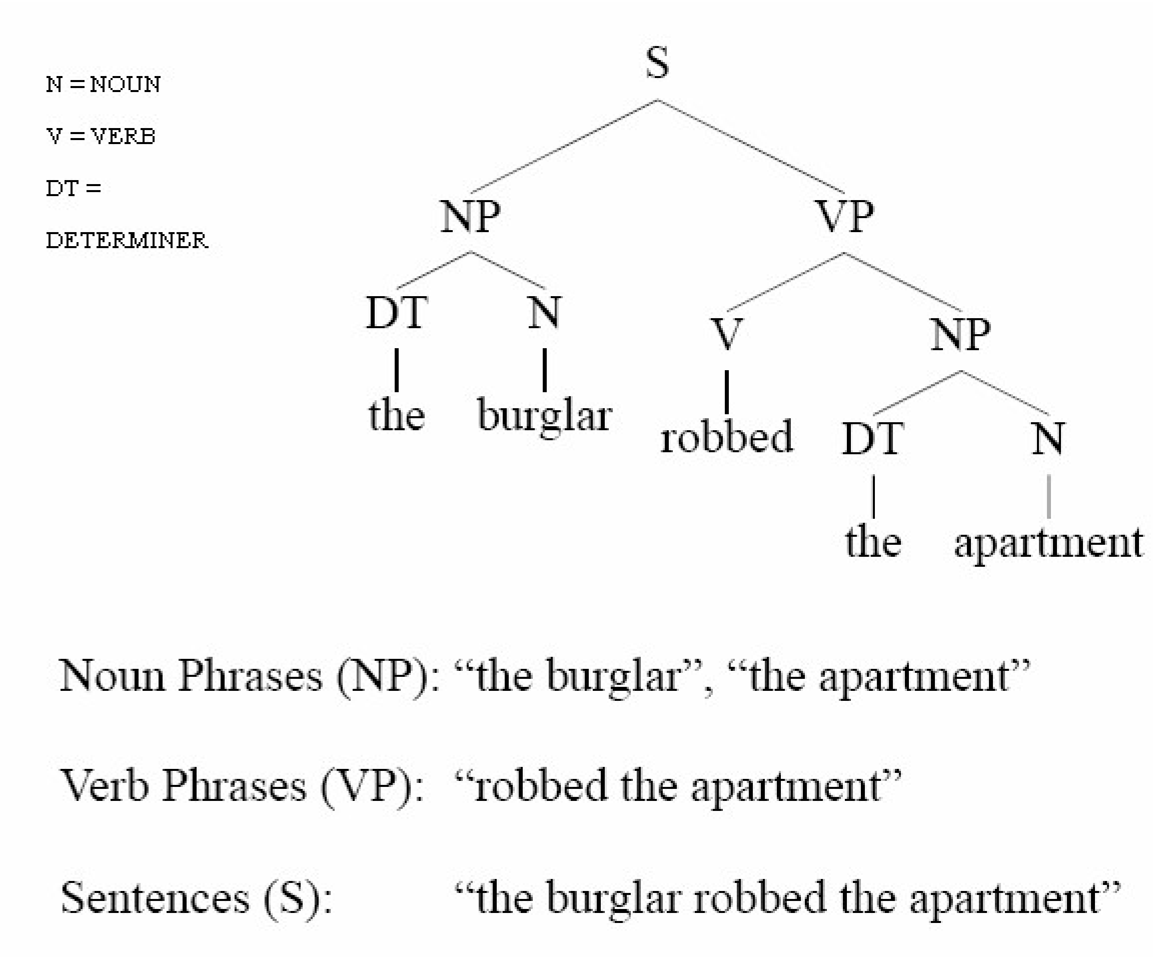


Figure 1 shows the hierarchical structure of a simple sentence. Representing a sentence as a tree structure allows us to work with the individual parts of the sentence by using subtrees.

Parsing:

- Parsing is the process of analyzing a sentence by taking it apart word – by – word and determining its structure from its constituent parts and sub parts.
- The structure of a sentence can be represented with a syntactic tree.
- When given an input string, the lexical parts or terms (root words), must first be identified by type and then the role they play in a sentence must be determined.
- These parts can be combined successively into larger units until a complete tree has been computed.

Example:



N = NOUN

V = VERB

DT = DETERMINER

Noun Phrases (NP): “the burglar”, “the apartment”

Verb Phrases (VP): “robbed the apartment”

Sentences (S): “the burglar robbed the apartment”