Cover Page

(Lab work of AI using Python)

1. Write programs for Hangman (implementation of simple intelligent agent)

import random

# List of words to choose from

words = ['apple', 'banana', 'cherry', 'durian', 'elderberry', 'fig', 'grape']

# Select a random word from the list

word = random.choice(words)

# Create a list of underscores with the same length as the word

word\_progress = ['\_' for i in range(len(word))]

# Set the number of incorrect guesses allowed

max\_guesses = 6

# Keep track of the incorrect guesses made

incorrect\_guesses = []

# Loop until the game is over

while True:

# Print the current progress of the word

print(' '.join(word\_progress))

# Get a letter from the user

guess = input("Guess a letter: ").lower()

# Check if the letter has already been guessed

if guess in incorrect\_guesses or guess in word\_progress:

print("You've already guessed that letter!")

continue

# Check if the letter is in the word

if guess in word:

# Update the progress of the word

for i in range(len(word)):

if word[i] == guess:

word\_progress[i] = guess

print("Correct!")

else:

# Add the letter to the list of incorrect guesses

incorrect\_guesses.append(guess)

print("Incorrect!")

# Check if the player has won

if '\_' not in word\_progress:

print("Congratulations, you've won!")

break

# Check if the player has run out of guesses

if len(incorrect\_guesses) >= max\_guesses:

print("Sorry, you've lost. The word was '{}'.".format(word))

break

2. Write programs for BFS from queue import Queue

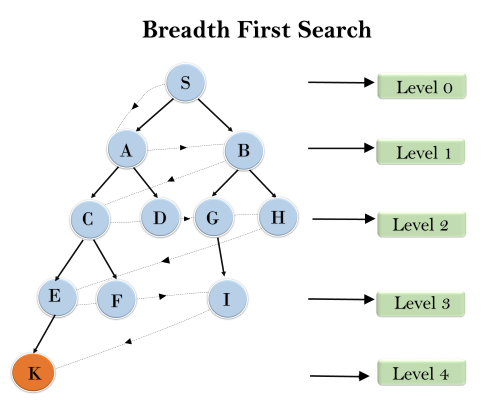
* Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
* BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before 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.



The given tree structure show the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

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.

# Define the graph as a dictionary of lists

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

# Define the BFS function

def bfs(graph, start):

visited = set() # Set to keep track of visited nodes

queue = Queue() # Initialize a queue to store nodes to visit

queue.put(start) # Add start node to queue

while not queue.empty():

node = queue.get() # Get the next node from the queue

if node not in visited:

print(node) # Visit the node

visited.add(node)

for neighbor in graph[node]: # Add unvisited neighbors to queue

if neighbor not in visited:

queue.put(neighbor)

# Test the BFS function with graph starting at 'A'

bfs(graph, 'A')

3. Write programs for DFS.

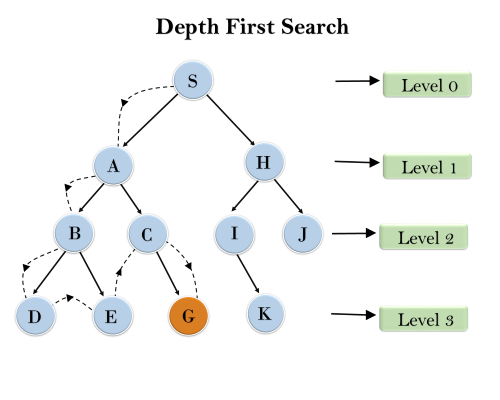
* Depth-first search isa recursive algorithm for traversing a tree or graph data structure.
* It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
* DFS uses a stack data structure for its implementation.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

**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.



* In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:
* 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.

# Define the graph as an adjacency list

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

# Define the DFS function

def dfs(graph, start, visited=None):

# Initialize visited if it is None

if visited is None:

visited = set()

# Mark the starting node as visited

visited.add(start)

print(start, end=' ')

# For each adjacent node that has not been visited, call DFS recursively

for neighbor in graph[start]:

if neighbor not in visited:

dfs(graph, neighbor, visited)

# Call the DFS function with the starting node

dfs(graph, 'A')

4. Write programs for A\*.

* 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**.
* **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.

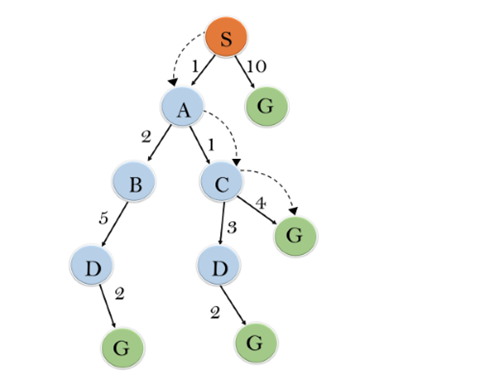
This algorithm can solve very complex problems.

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.



**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) <="" li="">

**Complete:** A\* algorithm is complete as long as:

Branching factor is finite.

Cost at every action is fixed.

**Optimal:** A\* search algorithm is optimal if it follows below two conditions:

**Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A\* tree search. An admissible heuristic is optimistic in nature.

**Consistency:** Second required condition is consistency for only A\* graph-search.

If the heuristic function is admissible, then A\* tree search will always find the least cost path.

**Time Complexity:** The time complexity of A\* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

**Space Complexity:** The space complexity of A\* search algorithm is **O(b^d)**

inputGraph = {

'A': [('B', 7, 6), ('D', 8, 5)],

'B': [('A', 7, 9), ('C', 9, 3)],

'C': [('D', 5, 5), ('B', 9, 4)],

'D': [('A', 5, 11), ('C', 5, 3)],

}

goal = "D"

def astar(graph, start):

# Set initial root-to-parent cost to 0 and add start node to the priority queue

rootToParentCost = 0

queue = [start + (rootToParentCost, )]

visitedNodes = []

while queue:

# Sort the queue based on the combined cost of the current node and the heuristic value

queue = sorted(queue, key=lambda x: x[1] + x[2] + x[3])

# Pop the node with the lowest combined cost and update the root-to-parent cost

node = queue.pop(0)

rootToParentCost = node[1] + node[3]

if node not in visitedNodes:

visitedNodes.append(node)

if node[0] == goal:

break

neighbours = graph[node[0]]

for neighbour in neighbours:

# Calculate the heuristic value for the neighbour

heuristic = neighbour[2]

# Add the neighbour to the queue with its combined cost and root-to-parent cost

queue.append(neighbour + (rootToParentCost, ))

return visitedNodes

visitedNodes = astar(inputGraph, ('A', 5, 10))

print(visitedNodes)

5. Write programs for Greedy Best First

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.

1. f(n)= g(n).

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

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

Best first search 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.

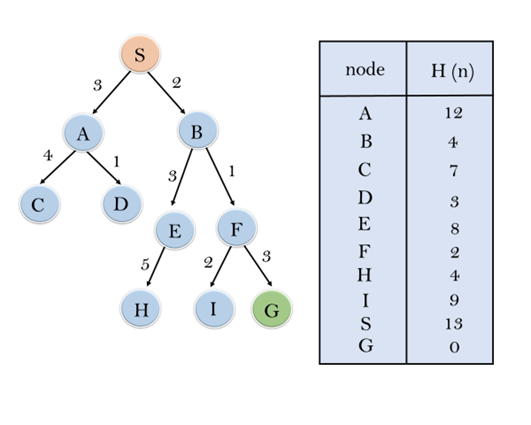
It can get stuck in a loop as DFS.

This algorithm is not optimal.

Example:

Consider the given 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 table.

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists.



**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]

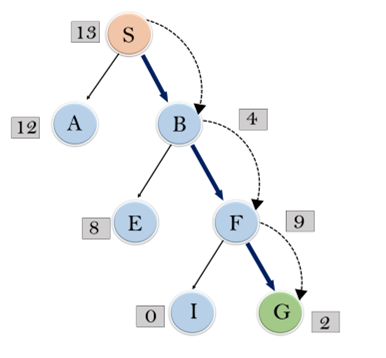
Hence the final solution path will be: **S----> B----->F----> G**

**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.



# Define the input graph and the goal node

inputGraph = {

'A': [('B', 4), ('D', 12)],

'B': [('A', 9), ('C', 4)],

'C': [('D', 4), ('B', 7)],

'D': [('A', 9), ('C', 5)]

}

goal = "C"

# Define the function for Best First Search algorithm

def gbfs(graph, start):

# Create a queue with the start node and a list to store visited nodes

queue = [start]

visitedNode = []

# Loop through the queue while it's not empty

while queue:

# Sort the queue based on the heuristic function

queue = sorted(queue, key=lambda x: x[1])

# Pop the first node from the queue

node = queue.pop(0)

# Check if the node has not already been visited

if node not in visitedNode:

visitedNode.append(node)

# Check if the current node is the goal node

if node[0] == goal:

break

# Get the neighbours of the current node and add them to the queue

neighbours = graph[node[0]]

for neighbour in neighbours:

queue.append(neighbour)

# Return the list of visited nodes

return visitedNode

# Call the function with the input graph and the start node

print(gbfs(inputGraph, ("A", 13)))