# Depth First Search

## Theory

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

### Algorithm

- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

### Analysis:

- $V$ - number of vertices
- $E$ - number of edges
- **Time Complexity:** $O(V + E)$
- **Space Complexity:** $O(V)$
- Complete search

## Code

```python
def DFS(graph, start, goal, visited):
    visited.add(start)
    print(start, end=" ")
    if start == goal:
        return
    for neighbor in graph[start]:
        if neighbor not in visited:
            DFS(graph, neighbor, goal, visited)


graph = {
    'S': {'A': 3, 'B': 2},
    'A': {'C': 4, 'D': 1, 'S': 3},
    'B': {'E': 3, 'F': 1, 'S': 2},
    'C': {'A': 4},
    'D': {'A': 1},
    'E': {'B': 3, 'H': 5},
    'F': {'B': 1, 'I': 2, 'G': 3},
    'G': {'F': 3},
    'I': {'F': 2},
    'H': {'E': 5}
}

DFS(graph, 'S', 'G', set())
```

## Output

```
S A C D B E H F I G
```

# Depth Limited Search

## Theory

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

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.

### Algorithm

- Determine the start node and the search depth.
- Check if the current node is the goal node.
  If not: Do nothing
  If yes:return
- Check if the current node is within the specified search depth
  If not: Do nothing
  If yes:Expand the node and save all of its successors in a stack.
- Call DLS recursively for all nodes of the stack and go back to step 2.

### Analysis

- $b$ - branching factor
- $d$ - depth of the graph
- $l$ - is the restricted depth
- **Time Complexity:** $O(b * l)$
- **Time Complexity:** $O(b * l)$
- Incomplete search

## Code

```python
def depth_limited_search(graph, start, end, visited, level, limit):
    if level == limit:
        return
    visited.add(start)
    print(start, end=" ")
    if start == end:
        return
    for neighbor in graph[start]:
        if neighbor not in visited:
            depth_limited_search(graph, neighbor, end,
                                 visited, level + 1, limit)


graph = {
    'S': {'A': 3, 'B': 2},
    'A': {'C': 4, 'D': 1, 'S': 3},
    'B': {'E': 3, 'F': 1, 'S': 2},
    'C': {'A': 4},
    'D': {'A': 1},
    'E': {'B': 3, 'H': 5},
    'F': {'B': 1, 'I': 2, 'G': 3},
    'G': {'F': 3},
    'I': {'F': 2},
    'H': {'E': 5}
}

depth_limited_search(graph, 'S', 'G', set(), 0, 3)
```

## Output

```
S A C D B E F
```

## Breadth First Search

### Theory

Breadth-first search (BFS) is an algorithm for searching a graph data structure for a node that satisfies a given property. It starts at the graph root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

#### Algorithm

- Create an empty queue q
- Append the source node to q
- Loop while q is not empty
  - tempNode <- q.deque()
  - Enqueue tempNode's children (first left then right children) to q

#### Analysis:

- $V$ - number of vertices
- $E$ - number of edges
- **Time Complexity:** $O(V + E)$
- Complete search

## Code

```
from queue import Queue


def BFS(graph, source, destination, visited):
    q = Queue()
    q.put(source)
    visited.add(source)
    while not q.empty():
        current = q.get()
        print(current, end=" ")
        if(current == destination):
            return
        for neighbor in graph[current]:
            if neighbor not in visited:
                visited.add(neighbor)
                q.put(neighbor)


graph = {
    'S': {'A': 3, 'B': 2},
    'A': {'C': 4, 'D': 1, 'S': 3},
    'B': {'E': 3, 'F': 1, 'S': 2},
    'C': {'A': 4},
    'D': {'A': 1},
    'E': {'B': 3, 'H': 5},
    'F': {'B': 1, 'I': 2, 'G': 3},
    'G': {'F': 3},
    'I': {'F': 2},
    'H': {'E': 5}
}

BFS(graph, 'S', 'G', set())
```

## Output

```
S A B C D E F H I G
```

# Best First Seach

## Theory

Best first search is a traversal technique that decides which node is to be visited next by checking which node is the most promising one and then check it. For this it uses an evaluation function to decide the traversal.

This best first search technique of tree traversal comes under the category of heuristic search or informed search technique.

The cost of nodes is stored in a priority queue. This makes implementation of best-first search is same as that of breadth First search. We will use the priority queue just like we use a queue for BFS.

### Algorithm

- Create a priorityQueue pqueue.
- insert start in pqueue : pqueue.insert(start)
- delete all elements of pqueue one by one.
  - if, the element is goal. Exit.
  - else, traverse neighbors and mark the node examined.
- End.

### Analysis:

- $b$ - branching factor
- $d$ - depth of the graph
- **Time Complexity:** $O(b^d)$
- **Time Complexity:** $O(b^d)$
- Complete search

## Code

```python
from queue import PriorityQueue


def Best_First_Search(grid, heuristic, start, end):
    visited = set([start])
    pq = PriorityQueue()
    pq.put([heuristic[start], start])
    while(not pq.empty()):
        current = pq.get()
        print(current[1], end=" ")
        if current[1] == end:
            break
        for neighbor in grid[current[1]]:
            if neighbor not in visited:
                pq.put([heuristic[neighbor], neighbor])
                visited.add(neighbor)


graph = {
    'S': {'A': 3, 'B': 2},
    'A': {'C': 4, 'D': 1, 'S': 3},
    'B': {'E': 3, 'F': 1, 'S': 2},
    'C': {'A': 4},
    'D': {'A': 1},
    'E': {'B': 3, 'H': 5},
    'F': {'B': 1, 'I': 2, 'G': 3},
    'G': {'F': 3},
    'I': {'F': 2},
    'H': {'E': 5},
}

heuristic = {
    'S': 13,
    'A': 12,
    'B': 4,
    'C': 7,
    'D': 3,
    'E': 8,
    'F': 2,
    'G': 0,
    'H': 4,
    'I': 9
}

source = 'S'
destination = 'G'

Best_First_Search(graph, heuristic, source, destination)
```

## Output

```
S B F G
```

# Beam Search

## Theory

Beam search is a heuristic search algorithm that explores a graph by expanding the most promising node in a limited set. Beam search is an optimization of best-first search that reduces its memory requirements. Best-first search is a graph search which orders all partial solutions (states) according to some heuristic. But in beam search, only a predetermined number of best partial solutions are kept as candidates. It is thus a greedy algorithm.

### Algorithm

- set Node = rootNode and found = False
- if node is goalNode then found = True
- else find the successors of node with their estimated cost and store it in the open list.
- while found is False and open list is not empty
  - sort the open list
  - select top w candidate from the open list and put it in wopen list. Clean the open list.
  - for each node in wopen list
    - if node is the goal node then found = True
    - else find the successors of the node with their estimated cost and store it in the open list.

### Analysis:

- $b$ - branching factor
- $d$ - depth of the graph
- $w$ - is the beam width
- **Time Complexity:** $O(b^d)$
- **Time Complexity:** $O(w)$
- Incomplete search

## Code

```
from queue import PriorityQueue


def Beam_Search(grid, heuristic, start, end, beamWidth):
    visited = set([start])
    pq = PriorityQueue()
    pq.put([heuristic[start], start])

    while(not pq.empty()):
        current = pq.get()

        print(current[1], end=" ")

        if current[1] == end:
            break

        npq = pq
        pq = PriorityQueue()

        for neighbor in grid[current[1]]:
            if neighbor not in visited:
                pq.put([heuristic[neighbor], neighbor])
                visited.add(neighbor)

        for i in range(beamWidth):
            if not npq.empty():
                pq.put(npq.get())


graph = {
    'S': {'A': 3, 'B': 2},
    'A': {'C': 4, 'D': 1, 'S': 3},
    'B': {'E': 3, 'F': 1, 'S': 2},
    'C': {'A': 4},
    'D': {'A': 1},
    'E': {'B': 3, 'H': 5},
    'F': {'B': 1, 'I': 2, 'G': 3},
    'G': {'F': 3},
    'I': {'F': 2},
    'H': {'E': 5},
}

heuristic = {
    'S': 13,
    'A': 12,
    'B': 4,
    'C': 7,
    'D': 3,
    'E': 8,
    'F': 2,
    'G': 0,
    'H': 4,
    'I': 9
}
```

```
source = 'S'
destination = 'G'
beamWidth = 2

Beam_Search(graph, heuristic, source, destination, beamWidth)
```

### Output

```
S B F G
```

# A*

## Theory

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals. Many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

### Algorithm

- Initialize the open list
- Initialize the closed list put the starting node on the open list (you can leave its f at zero)
- while the open list is not empty
  - find the node with the least $f$ on the open list, call it q
  - pop q off the open list
  - generate q's successors and set their parents to q
  - for each successor
    - if successor is the goal, stop search
    - else, compute both $g$ and $h$ for successor
      **successor.g** = q.g + distance between successor and q
      **successor.h** = distance from goal to successor
      (This can be done using many ways, we will discuss three heuristics- Manhattan, Diagonal and Euclidean Heuristics)
      **successor.f** = successor.g + successor.h
    - if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
    - if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor otherwise, add the node to the open list end (for loop)
  - push q on the closed list end (while loop)

### Analysis:

- $b$ - branching factor
- $d$ - depth of the graph
- **Time Complexity:** $O(b^d)$
- **Time Complexity:** $O(b^d)$
- Complete search, informed search

## Code

```
from collections import deque


class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but who's
   neighbors
        # haven't all been inspected, starts off with the start node
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])

        # g contains current distances from start_node to all other nodes
        # the default value (if it's not found in the map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() - evaluation function
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            # then we begin reconstructing the path from it to the start_node
            if n == stop_node:
                reconst_path = []
                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]
                reconst_path.append(start_node)
                reconst_path.reverse()
                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both open_list and closed_list
                # add it to open_list and note n as it's parent
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it to open_list
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None


adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

**Output**

```
Path found: ['A', 'B', 'D']
```