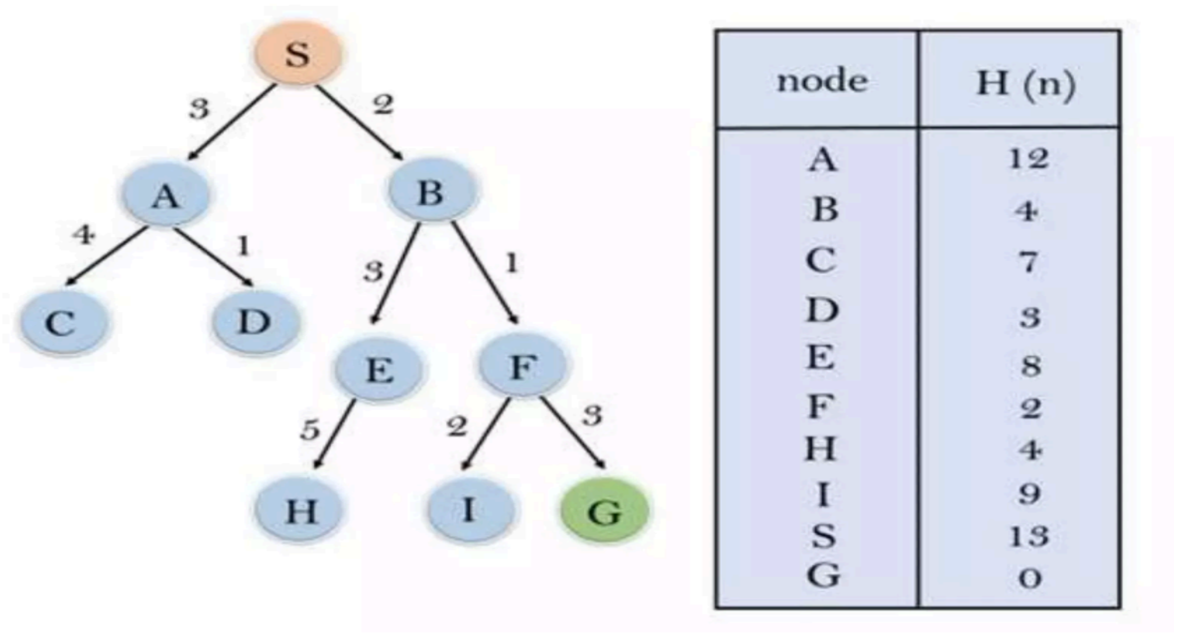
BFS best first search



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

1-Overview

Best-First Search (BFS) is a graph traversal and search algorithm that uses heuristic functions to guide the search toward the most promising path. It combines Depth-First Search (DFS) and Breadth-First Search (BFS), using a priority queue to evaluate and prioritize nodes based on heuristic estimates.

2-Algorithm steps

1. Initialization: Start from the initial node and place it in an open list (priority queue).

2. Node Selection: The algorithm selects the node with the lowest heuristic value from the open list.

3. Expansion: Expand the selected node, generating its successors.

4. Heuristic Evaluation: Evaluate successors using the heuristic function, which estimates the cost to reach the goal from a node.

5. Termination: Repeat until the goal is reached, or no nodes remain in the open list.

3-Advantages

1. Efficient for problems with good heuristic definitions.

2. Finds solutions faster than uninformed search algorithms like DFS and BFS in heuristic-driven problems.

4-Disadvantages

1. Performance depends heavily on the quality of the heuristic function.

2. Not guaranteed to find the optimal solution unless combined with cost-awareness (e.g., in A* Search).

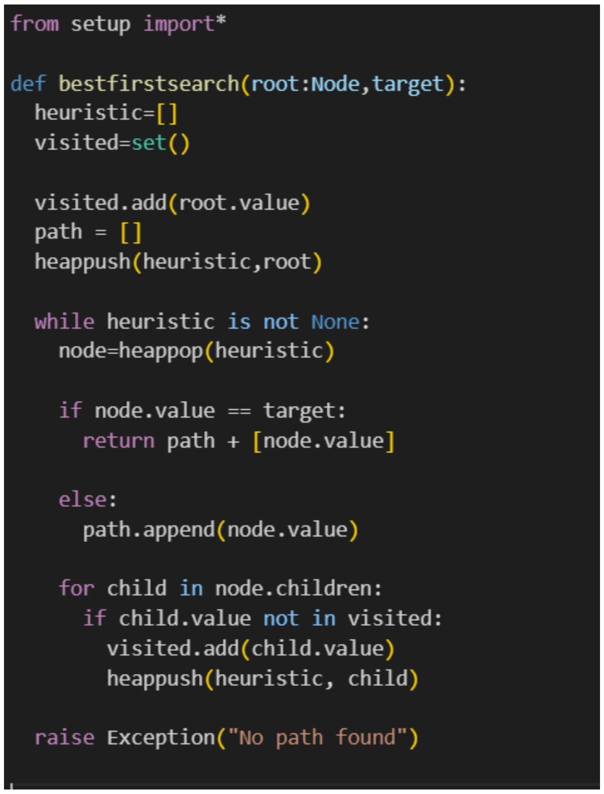
5-Applications

1. Game AI: Pathfinding in games like Mario or Contra.

2. Robotics: Navigation through obstacles.

3. Optimization problems: identifying the shortest path or efficient solution.

6- code explanation



```python
from setup import*

def bestfirstsearch(root:Node,target):
  heuristic=[]
  visited=set()

  visited.add(root.value)
  path = []
  heappush(heuristic,root)

  while heuristic is not None:
    node=heappop(heuristic)

    if node.value == target:
      return path + [node.value]

    else:
      path.append(node.value)

    for child in node.children:
      if child.value not in visited:
        visited.add(child.value)
        heappush(heuristic, child)

  raise Exception("No path found")
```

1- Initialize:

- heuristic: A priority queue for nodes.
- visited: A set to track visited nodes.
- Add the root to both visited and the heuristic.

2- Search Loop:

- While there are nodes in heuristic, pop the node with the lowest heuristic.
- If it matches the target, return the path to the target.
- Otherwise, add its value to the path and explore its unvisited children, adding them to heuristic and visited.

3- **No Path Found**:

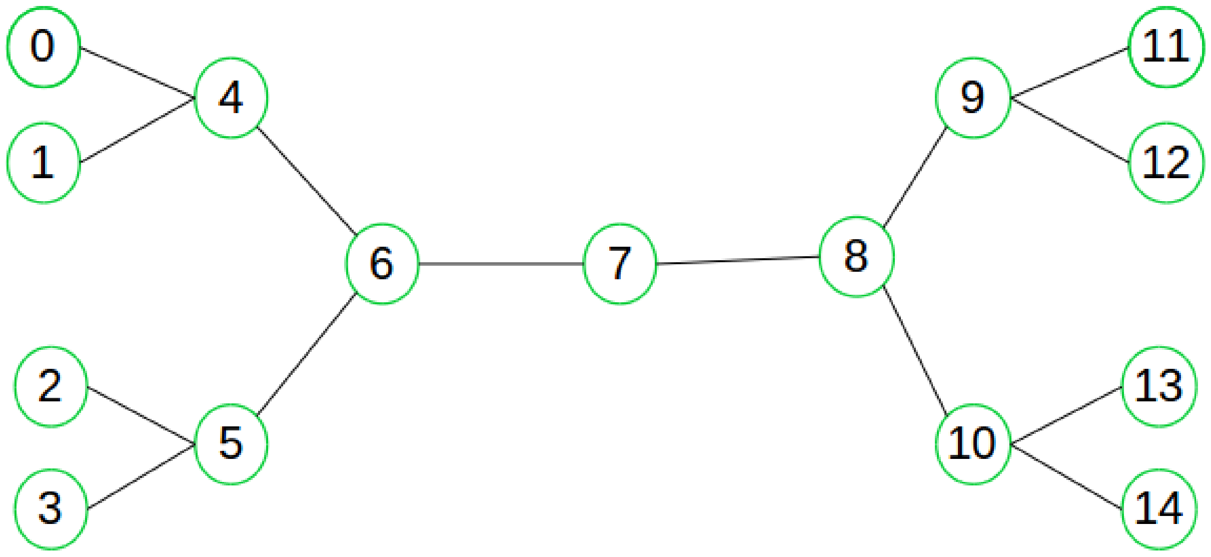
- If the target isn't found after exploring all nodes, raise an exception.

Resources

https://www.javatpoint.com/best-first-search-in-artificial-intelligence

https://iq.opengenus.org/best-first-search/

Bidirectional search



1-Overview

Bidirectional search is an efficient graph traversal algorithm that searches simultaneously from two directions: the start node and the goal node. It meets in the middle to significantly reduce the search space compared to unidirectional search algorithms like Breadth-First Search (BFS). This method is commonly applied in problems requiring the shortest path in large graphs, such as navigation systems, game development, and artificial intelligence.

2-Algorithm steps

Bidirectional search uses two BFS processes:

- **One starts from the source node and expands outward.**
- **The other starts from the goal node and expands backward.**

The algorithm halts when the two searches meet, and the shortest path is reconstructed by combining the paths from the start node to the meeting point and from the goal node to the meeting point.

Key advantages of this approach include reduced memory and time requirements because the search depth is halved compared to single-directional searches.

3-Advantages

1. Faster for large graphs as it reduces the search space.

2. Memory-efficient compared to BFS, which explores a much larger node set.

4-Limitations

1. Requires knowledge of the goal state.

2. Inefficient for graphs where backward exploration is not feasible (e.g., one-way streets).

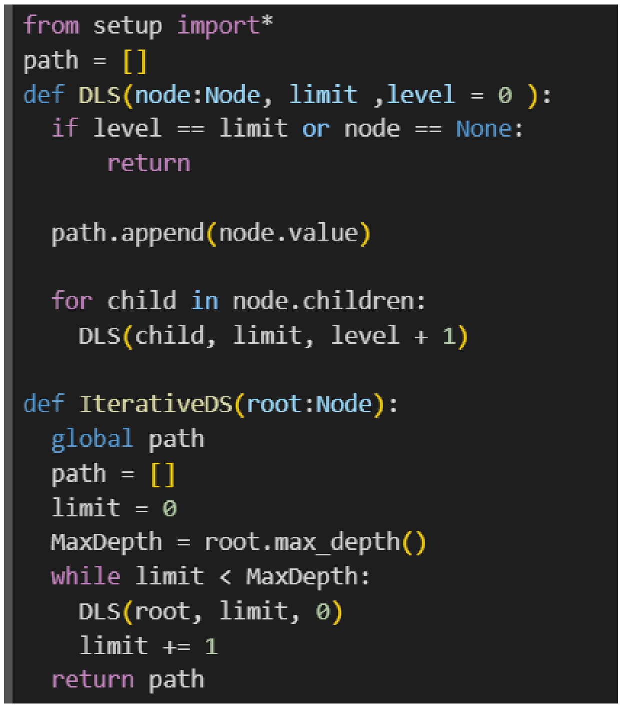
5-Applications

1. Navigation Systems: Finding the shortest route between two points in road networks.

2. AI in Games: Computing paths for non-player characters between two points.

3. Social Network Analysis: Identifying the shortest connection path between two users.

6- code explanation



```python
from setup import*
path = []
def DLS(node:Node, limit ,level = 0 ):
  if level == limit or node == None:
      return

  path.append(node.value)

  for child in node.children:
    DLS(child, limit, level + 1)

def IterativeDS(root:Node):
  global path
  path = []
  limit = 0
  MaxDepth = root.max_depth()
  while limit < MaxDepth:
    DLS(root, limit, 0)
    limit += 1
  return path
```

This code implements Bidirectional Search to find the shortest path between a `root` node and a `target` node by performing two BFS searches: one from the `root` and another from a `joint` node towards the `target`.

Steps:

`bfs(root, target)`:

- Purpose: Performs BFS from `root` to `target`.
- It uses a queue to explore nodes and a `visited` set to track explored nodes. If the target is found, it returns the path from `root` to `target`.

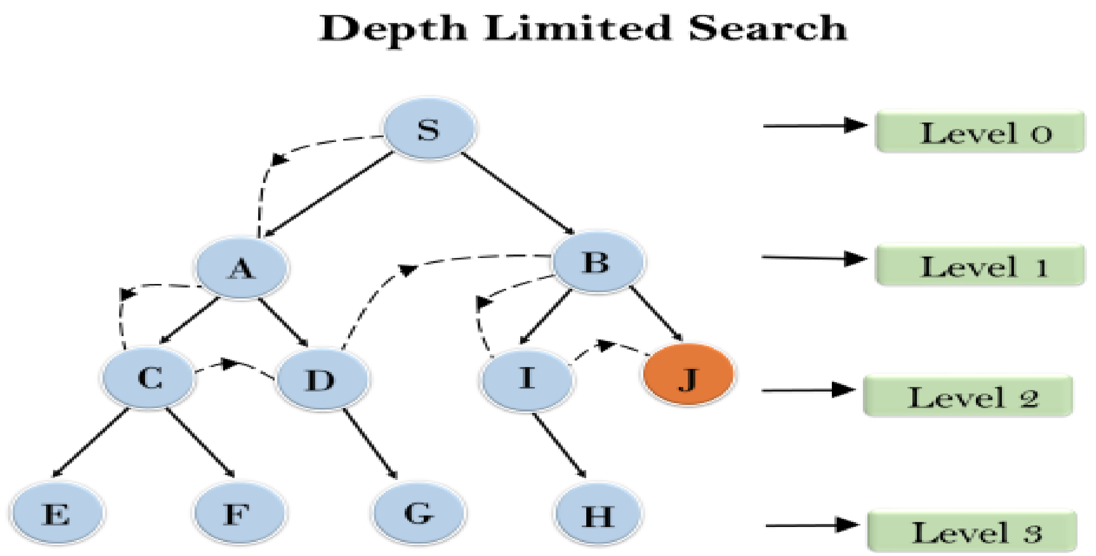
`bidirectional_search(root, joint, target)`:

- Purpose: Finds the shortest path using Bidirectional Search.
- First, it performs BFS from `root` to `joint`.
- Then, it performs BFS from `joint` to `target` (reversed).
- Combines both paths and returns the final path from `root` to `target`

Resources

https://en.wikipedia.org/wiki/Bidirectional_search

DLS Depth-Limited Search



1-Overview

Depth-Limited Search is an uninformed search algorithm that extends Depth-First Search (DFS) by setting a predefined limit on the search depth. This modification addresses one of DFS's main challenges: infinite looping in infinite-depth spaces. The algorithm assumes that nodes beyond the specified depth limit have no successors, effectively bounding the search to a manageable depth.

2-Algorithm steps

1. Initialization: Start at the root node, initializing a depth counter.
2. Exploration: Traverse down each branch of the tree or graph to explore child nodes.
3. Depth Check: Stop exploring a branch if the depth limit is reached and backtrack to explore other paths.
4. Goal Check: If the target node is found before hitting the depth limit, return success.
5. Backtracking: If no solution is found within the limit, return failure.

3-Advantages

1. Prevents infinite looping in search spaces.
2. Reduces memory requirements compared to standard DFS.
3. Enables control over the computational resources used.

4-Disadvantages

1. Not guaranteed to find a solution if the depth limit is too low.
2. May not find the optimal solution in cases with multiple solutions.
3. Choosing an appropriate depth limit can be challenging.

5-Applications

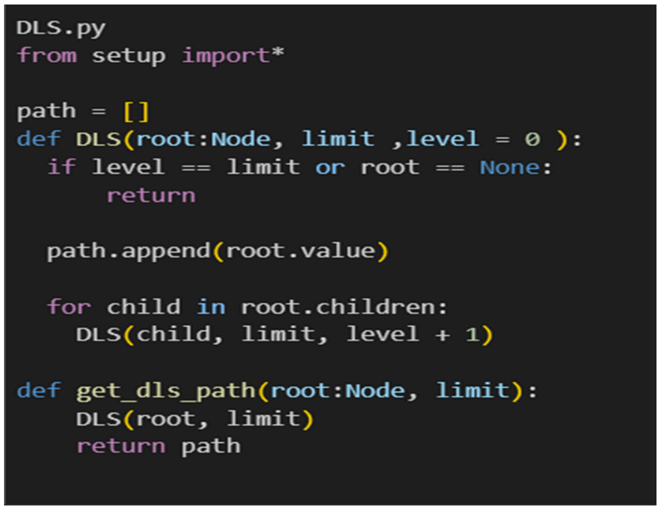
1. Robotics: Used in motion planning to restrict a robot's search space to a specific area, avoiding excessive exploration.

2. Game AI: Common in games where only a limited number of moves ahead need to be evaluated (e.g., chess).

3. Puzzle Solving: Applied to puzzles like the 8-puzzle or Sudoku with predefined move limits.

4. Network Routing: Ensures efficient routing by limiting the number of hops between nodes.

6- code explanation



```
DLS.py
from setup import*

path = []
def DLS(root:Node, limit ,level = 0 ):
  if level == limit or root == None:
      return

  path.append(root.value)

  for child in root.children:
    DLS(child, limit, level + 1)

def get_dls_path(root:Node, limit):
    DLS(root, limit)
    return path
```

1. **Arguments**:

    1- root: The root node of the tree to be traversed.

    2- limit: The maximum depth to explore.

2. **Process**:

1- Calls the DLS function to traverse the tree up to the given limit.

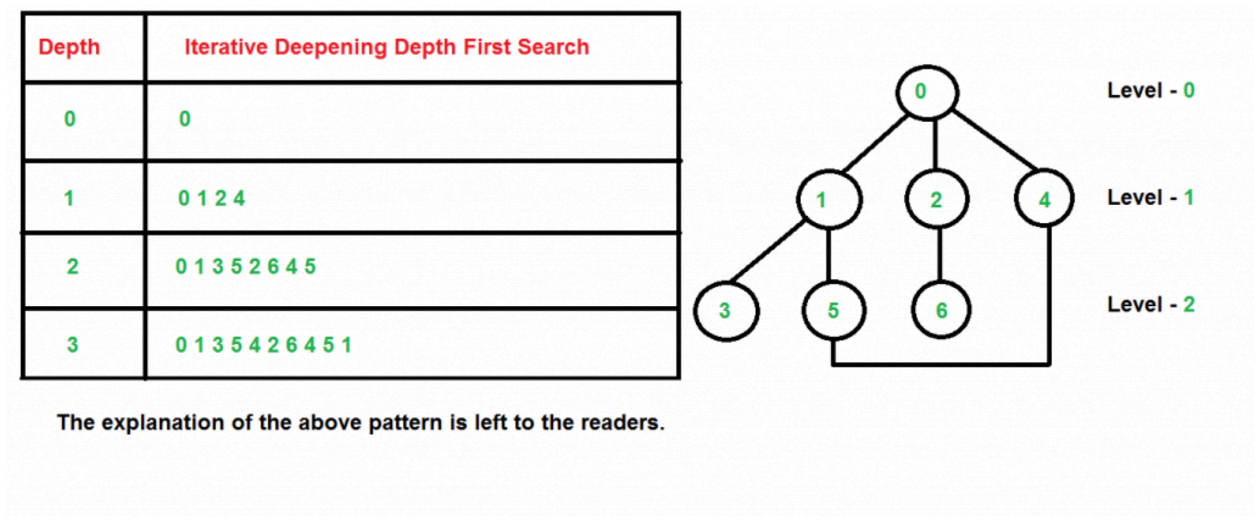
2- Returns the path list containing the values of the visited nodes.

Resources

https://www.geeksforgeeks.org/depth-limited-search-for-ai/

https://iq.opengenus.org/depth-limited-search/

IDDFS iterative deepening search



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.

1-Overview

Iterative Deepening Depth-First Search (IDDFS) is an algorithm that combines the benefits of Depth-First Search (DFS) and Breadth-First Search (BFS). It performs DFS with increasing depth limits, effectively exploring all nodes at a given depth before incrementing the limit. This iterative strategy makes IDDFS both complete and optimal for unweighted graphs.

2-Algorithm steps

1. **Depth-Limited Search (DLS): IDDFS begins by performing a DFS up to a depth limit of 0. It then increments this limit progressively (1, 2, 3,...).**
2. **Repetition: At each depth limit, the algorithm restarts the search from the root node.**
3. **Termination: The process stops when the goal node is found or when all nodes are explored up to the maximum depth.**

This approach leverages the memory efficiency of DFS and the completeness of BFS.

3-Advantages

1. **Completeness: IDDFS is guaranteed to find a solution if it exists, provided the graph/tree is finite.**
2. **Optimality: It finds the shortest path in unweighted graphs.**
3. **Applicable for Infinite Graphs: IDDFS can handle graphs of unknown or infinite depth by incrementally exploring deeper levels.**

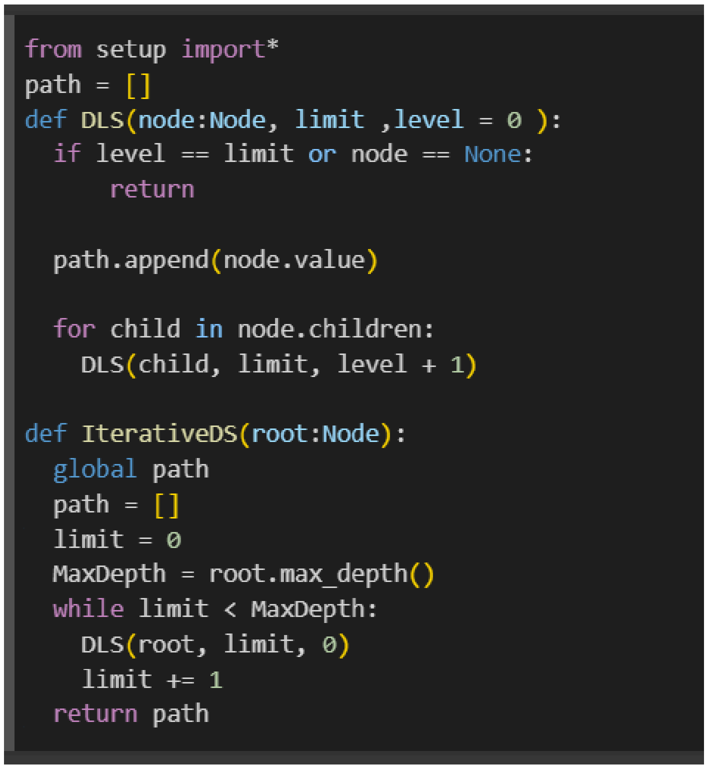
4-Disadvantages

Recomputations: Nodes at shallower depths are revisited multiple times, leading to redundant calculations.

5-Applications

1. **Artificial Intelligence: Used in search algorithms for game trees and puzzles like the 8-puzzle or chess.**
2. **Web Crawling: helpful for incrementally exploring websites layer by layer.**
3. **Robotics and Pathfinding: IDDFS is employed in navigation systems where memory constraints are significant.**
4. **Data Retrieval: useful in databases with hierarchical data structures.**

6- code explanation



```python
from setup import*
path = []
def DLS(node:Node, limit ,level = 0 ):
  if level == limit or node == None:
      return

  path.append(node.value)

  for child in node.children:
    DLS(child, limit, level + 1)

def IterativeDS(root:Node):
  global path
  path = []
  limit = 0
  MaxDepth = root.max_depth()
  while limit < MaxDepth:
    DLS(root, limit, 0)
    limit += 1
  return path
```

This code implements Iterative Deepening Search (IDS), which combines Depth-First Search (DFS) and Breadth-First Search (BFS). It explores nodes level by level by increasing the depth limit iteratively.

Steps:

1. **Start at depth 0 and explore with DLS.**
2. **Increase the depth limit after each iteration.**
3. **Return the final path after all depths are explored.**

Resources

1. https://iq.opengenus.org/iterative-deepening-search/