Problem Statement: -

How to implement Best-First Search algorithm in Python for efficient pathfinding in large-scale datasets, such as graphs, with the aim of finding the optimal path.

Solution: -

Pathfinding in large-scale datasets, such as graphs, is a fundamental problem in computer science and real-world applications like navigation systems and logistics optimization. The Best-First Search algorithm is a powerful technique for finding the optimal path in such scenarios. It combines the benefits of informed search, leveraging heuristic information, with the efficiency of a priority queue-based traversal strategy.

To implement the Best-First Search algorithm in Python for large-scale datasets, several key components need consideration. First, the choice of data structures is crucial. Utilizing an appropriate representation of the graph, like an adjacency list or matrix, is essential. Additionally, an efficient priority queue, often implemented as a min-heap, is used to manage nodes to be explored based on their estimated costs.

One of the critical aspects of the Best-First Search algorithm is the selection of a heuristic function. This function provides an estimate of the cost from the current node to the goal node. In large-scale datasets, optimizing both memory usage and traversal time is of utmost importance. Python's heap module is invaluable for handling the priority queue efficiently, and a dictionary can be employed to store and retrieve node information.

In the implementation, you'll follow a sequence of steps that includes initializing data structures, exploring nodes, and backtracking to find the optimal path when the goal is reached. Admissible heuristics that provide reasonable cost estimates for reaching the goal are paramount in large-scale datasets. Furthermore, you can employ various optimization techniques like pruning, early termination, and even strategies for handling datasets too large to fit in memory, such as external memory search or parallel processing.

By mastering the implementation of the Best-First Search algorithm in Python and understanding the intricacies of managing large-scale datasets, you can efficiently navigate through complex graphs and find the optimal path, making this algorithm a valuable tool in a variety of applications.

Introduction: -

The realm of Artificial Intelligence (AI) is a captivating journey into the world of machines emulating human intelligence. Within the intricate landscape of AI, the search for optimal solutions to complex problems is a fundamental quest. One algorithm that shines brightly in this pursuit is the Best-First Search algorithm. This algorithm, born from the heart of AI, stands as a linchpin in a wide array of applications, from guiding autonomous vehicles through labyrinthine city streets to deciphering the mysteries of intricate mazes in video games.

Best-First Search is a beacon of efficiency and intelligence, a mechanism that navigates through graphs, evaluating the promise of different paths, and dynamically selecting the most enticing route to a predefined destination. It balances the cost accrued thus far with an intelligent estimation of the remaining journey, making decisions not solely based on brute force but on strategic insights. It's the digital equivalent of a seasoned explorer, choosing paths through a dense forest not just by the length of the trail but by the promise of the hidden treasures ahead.

In this journey of exploration, we will venture deep into the intricate mechanisms that power the Best-First Search algorithm in Python, customized meticulously for graph traversal. We will embark on a voyage that unwraps the layers of this algorithm, unveiling its inner workings, from node representations to priority queues. We'll explore heuristic functions, the compass of Best-First Search, guiding it toward its goal with intelligence. Through this comprehensive guide, we aim to demystify Best-First Search, making it accessible to both novices eager to grasp the fundamentals of AI and seasoned developers seeking a robust tool for real-world problem-solving.

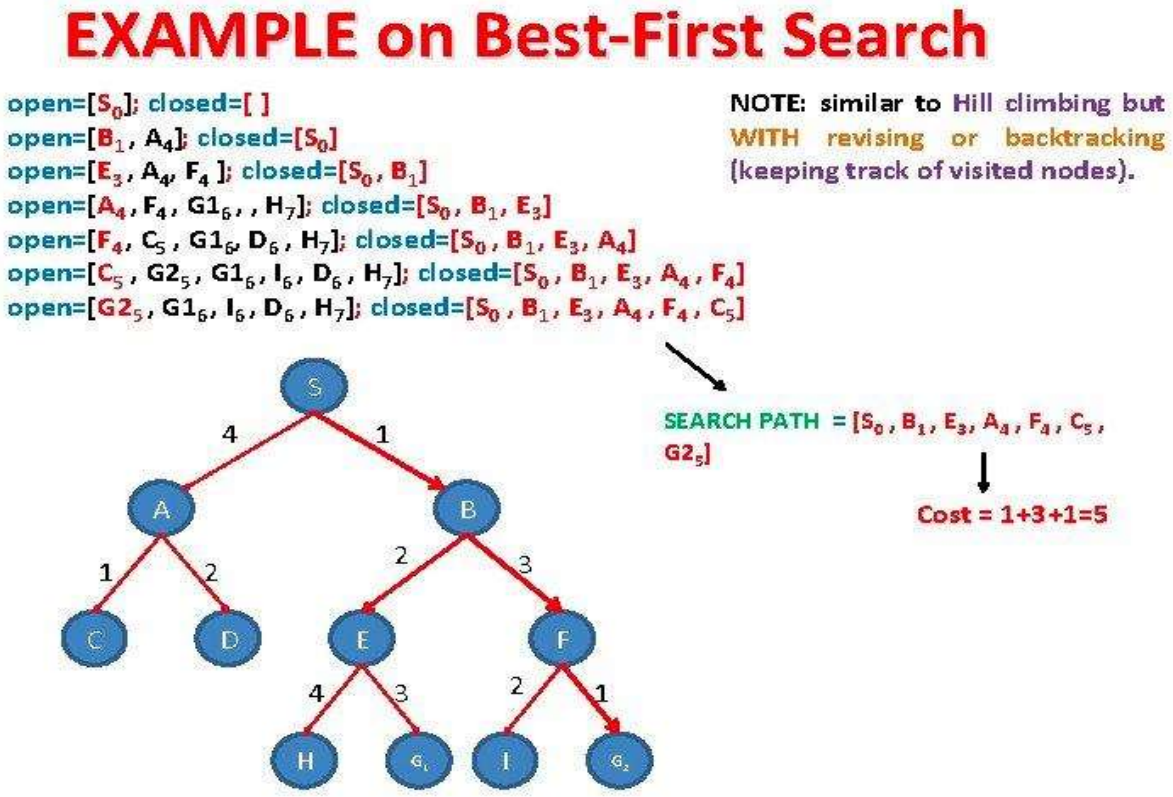
Prepare to embark on this expedition into the world of Best-First Search, as we equip you with the knowledge, skills, and practical insights needed to harness the algorithm's potential. Whether you're a student yearning to fathom the intricacies of AI or a professional ready to tackle complex challenges, our journey through Best-First Search will illuminate your path to success in the world of artificial intelligence and problem-solving.

Best First Search Algorithm: -

Best first search (BFS) is a search algorithm that functions at a particular rule and uses a priority queue and heuristic search. It is ideal for computers to evaluate the appropriate and shortest path through a maze of possibilities. Suppose you get stuck in a big maze and do not know how and where to exit quickly. Here, the **best first search in AI** aids your system program to evaluate and choose the right path at every succeeding step to reach the goal as quickly as possible.

For example, imagine you are playing a video game of Super Mario or Contra where you have to reach the goal and kill the enemy. The best first search aid computer system to control the Mario or Contra to check the quickest route or way to kill the enemy. It evaluates distinct paths and selects the closest one with no other threats to reach your goal and kill the enemy as fast as possible.

The **best first search in artificial intelligence** is an informed search that utilizes an evaluation function to opt for the promising node among the numerous available nodes before switching (transverse) to the next node. The **best first search algorithm in AI** utilizes two lists of monitoring the transversal while searching for graph space, i.e., Open and CLOSED list. An Open list monitors the immediate nodes available to transverse at the moment. In contrast, the CLOSED list monitors the nodes that are being transferred already.



Method of Best First Search: -

1. Create two empty lists
2. Start from the initial node and add it to the ordered open list
3. Next the below steps are repeated until the final node or endpoint is reached
4. If the open list is empty exit the loop and return a False statement which says that the final node cannot be reached
5. Select the top node in the open list and move it to the closed list while keeping track of the parent node
6. If the node removed is the endpoint node return a True statement meaning a path has been found and moving the node to the closed list
7. However, if it is not the endpoint node then list down all the neighbouring nodes of it and add them to the open list
8. According to the evaluation function re order the nodes.

This algorithm will traverse the shortest path first in the queue. The time complexity of the algorithm is O(n*log(n)).

Key Concepts Of BFS: -

Best-First Search (BFS) is an informed search algorithm used in computer science and artificial intelligence for solving various problems. It prioritizes nodes for exploration based on a heuristic function, which estimates the cost to reach the goal from a given node. Here are some key concepts associated with the Best-First Search algorithm:

Evaluation of Path

While using the best first search, your system always seeks possible nodes or paths that can be taken. Then, it picks the most promising or best node or path that is eligible to traverse the shortest distance node or path to reach the goal and exit the maze.

Use of Heuristic Function

The best first search uses a heuristic function in informed decisions. It helps in finding the right and quick path towards the goal, called heuristic search. The current state of the user in the maze is the input of this function, based on which it estimates how close the user is to the goal. Based on the analysis, it assists in reaching the goal in a reasonable time and with minimum steps.

Keeping Track

The Best-First Search algorithm in AI assists the computer system in tracking the paths or nodes it has traversed or plans to traverse. It prevents the system from becoming entangled in loops of previously tested paths or nodes and helps avoid errors.

Iteration of Process

The computer program keeps repeating the process of the above three criteria until it reaches the goal and exits the maze. Therefore, the **best first search in artificial intelligence** consistently revaluates the nodes or paths that are most promising based on the heuristic function.

These concepts are fundamental to understanding the Best-First Search algorithm and its variations, which are widely used in pathfinding, puzzle-solving, and optimization problems in AI and computer science.

Heuristic function: -

A heuristic function, often simply referred to as a "heuristic," is a function used in problem-solving and search algorithms to estimate or guess the cost or value associated with a particular decision or state in a problem. It provides a quick, approximate solution to a problem when an exact solution is computationally expensive or not feasible.

In the context of search algorithms like A*, Best-First Search, and various other informed search methods, the heuristic function is used to estimate the cost or distance from a current state or node to a goal state. This estimate helps guide the search algorithm's decision-making process by prioritizing which nodes to explore first.

Here are some key points about heuristic functions:

- Purpose: The primary purpose of a heuristic function is to provide an informed estimate of the cost or value. It helps algorithms focus on promising paths, potentially improving efficiency.

- Admissibility: A heuristic is considered admissible if it never overestimates the true cost to reach the goal. In other words, for all states, h(n) ≤ true cost from state n to the goal. Admissible heuristics ensure that search algorithms find an optimal solution when combined with cost-based information.

- Consistency: A heuristic is considered consistent (or monotonic) if, for every state n and its successor n', the heuristic value satisfies the inequality: h(n) ≤ c (n, n') + h(n'), where c (n, n') is the cost of transitioning from state n to n'. Consistent heuristics guarantee the optimality of algorithms like A.

- Quality: The quality of a heuristic greatly affects the efficiency of search algorithms. A good heuristic provides accurate estimates, guiding the algorithm efficiently toward the goal.

Heuristic functions are a crucial component of many artificial intelligence and optimization algorithms, enabling them to make informed decisions in situations where an exhaustive search of all possibilities is not practical.

Algorithmic Detail: -

- **Uniformed Algorithm**

  It is also called a blind method or exhaustive method. The search is done without additional information, which means based on the information already given in the problem statement. For instance, Depth First Search and Breadth First Search.

- **Informed Algorithm**

  The computer system performs the search based on the additional information provided to it, allowing it to describe the succeeding steps for evaluating the solution or path towards the goal. This popularly known method is the Heuristic method or Heuristic search. Informed methods outperform the blind method in terms of cost-effectiveness, efficiency, and overall performance.

  There are generally two variants of informed algorithm, i.e.,

1. **Greedy Best First Search**: Going with the name, this search algorithm is greedy and hence chooses the best path available at the moment. It uses a heuristic function and search, which is combined with depth and breadth-first search algorithms and combines the two algorithms where the most promising node is chosen while expanding the node present in proximity to the goal node.

2. **A* Best First Search**: It is the widely used type of best-first search. The search is efficient in nature due to the presence of combined features of greedy best-first search and UCS. Compared to greedy search, A* uses a heuristic function to look for the shortest path. It is quick and utilizes UCS with varied forms of heuristic function.

Applications: -

1. Robotics

   Best first search guides robots in a challenging situation and takes effective moves to navigate to their destination. Efficient planning is crucial in complex tasks so that it can evaluate the right paths toward the goal and make informed decisions accordingly.

2. Game Playing

   It helps game characters observe the threat, avoid obstacles, make the right decision-making strategic moves and evaluate the accurate path to reach the objectives within the time goal.

3. Navigation Apps

4. Data Mining and Natural Language Processing

   In data mining, artificial intelligence employs the best first search to assess the most suitable features that align with the data, facilitating selection. This reduces computational complexity in machine learning and enhances data model performance. Best first search algorithms also assess semantically similar phrases or terms to provide relevance.

5. Scheduling and Planning

   Best first search in artificial intelligence finds application in scheduling work and activities, enabling resource optimization and meeting deadlines. This functionality is integral to project management, logistics, and manufacturing.

Software: -

Visual Studio Code (VS Code) is an exceptional code editor widely adopted by developers for its lightweight design, extensibility, and cross-platform compatibility. It has proven invaluable in our project for several key reasons:

- Lightweight and Versatile: VS Code offers fast performance, making it an excellent choice for projects of all sizes, from small scripts to complex applications.

- Extensible and Customizable: With a rich library of extensions, VS Code can be tailored to the specific needs of the project. Extensions cover various programming languages, frameworks, and tools, enhancing productivity.

- Cross-Platform Compatibility: VS Code runs seamlessly on Windows, macOS, and Linux, ensuring a consistent experience regardless of the operating system, a crucial advantage for collaborative projects.

- Rich Ecosystem: VS Code's vibrant extension ecosystem supports various programming languages, version control systems, and development tools, making it suitable for multi-technology projects.

- Integrated Development Environment (IDE) Features: It offers integrated debugging, syntax highlighting, autocompletion with IntelliSense, Git integration, and a built-in terminal, all of which are essential for efficient coding.

- Version Control Integration: VS Code integrates seamlessly with Git, facilitating version control, collaboration, and code repository management.

- Free and Open Source: VS Code is open source under the MIT License, making it cost-effective and accessible to all project budgets.

In our project, Visual Studio Code played a pivotal role in enhancing coding efficiency, streamlining collaboration, and improving the overall development experience. Specific extensions and features were leveraged to meet project requirements, contributing significantly to our project's success.

Python: -

Python is the primary programming language employed in our project, and its selection has been instrumental in the successful development and execution of our tasks. Python is celebrated for several reasons:

- Readability and Simplicity: Python's clean and readable syntax enhances code clarity, making it easier for developers to write and understand code. This is particularly valuable when working collaboratively on projects.

- Versatility: Python is a versatile language that supports a wide range of applications. It is equally proficient in web development, data analysis, scientific computing, machine learning, and automation, making it a flexible choice for our project's diverse needs.

- Abundant Libraries: Python's extensive standard library and a rich ecosystem of third-party libraries provide ready-made solutions to numerous common programming tasks, reducing development time and effort.

- Open Source and Community Support: Python's open-source nature fosters a supportive and active community that contributes to the language's growth. This ensures a wealth of resources, documentation, and community-driven assistance for developers.

- Cross-Platform Compatibility: Python is cross-platform, making it adaptable to various operating systems. This versatility facilitates the development and deployment of our project on different platforms.

- Strong Integration: Python offers seamless integration with other languages and technologies, enabling the incorporation of specialized tools when required. This interoperability is particularly beneficial for our project, which involves diverse components.

Python's adaptability, readability, and extensive library support align perfectly with the multifaceted nature of our project. It has empowered our team to create efficient solutions, foster collaboration, and navigate the complexities of our tasks. With Python as the cornerstone of our project, we have harnessed a powerful and accessible language to achieve our goals.

Result & Analysis: -

To implement the best first search, the computer programs write code in different computer languages like Python, C, C++, and Java. It provides instructions to the computer system to evaluate the routes, paths or solutions and use heuristic functions. Here is a brief overview of steps on how the **best first search in artificial intelligence** can be implemented

Steps: -

- **Step 1: Initialization**
  - Choose a starting node and place it in the `OPEN` list.
- **Step 2: Initial Check**
  - If the `OPEN` list is empty, stop the search and return failure (no path exists).
- **Step 3: Node Selection**
  - Select the node from the `OPEN` list with the lowest heuristic value (h(n)) as the current node.
  - Remove the current node from the `OPEN` list and add it to the `CLOSED` list.
- **Step 4: Expand the Node**
  - Expand the current node by generating its successor nodes.
- **Step 5: Goal Check**
  - For each successor, check if it leads to the goal node.
  - If a successor node is the goal, return success and terminate the search.
- **Step 6: Path Construction**
  - If the goal is not reached, construct the path to the current node.
  - Backtrack from the current node to the starting node to build the path.
- **Step 7: Successor Evaluation**
  - For each successor:
  - Check if the successor node is already in the `OPEN` or `CLOSED` list.
  - If it's not in either list, add it to the `OPEN` list.
- **Step 8: Iteration**
  - Return to Step 2 and continue the search until a path to the goal is found or the `OPEN` list is empty (no path exists).

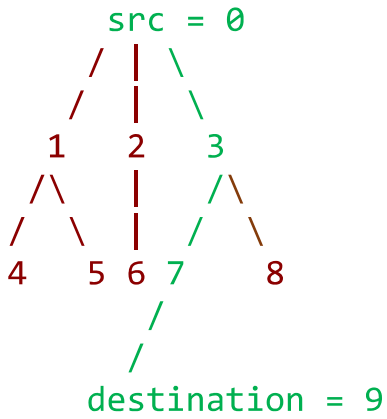
Code:-

```
# A Node class for GBFS Pathfinding
class Node:
def __init__(self, v, weight):
self.v=v
self.weight=weight

# pathNode class will help to store
# the path from src to dest.
class pathNode:
def __init__(self, node, parent):
self.node=node
self.parent=parent

# Function to add edge in the graph.
def addEdge(u, v, weight):
# Add edge u -> v with weight weight.
adj[u].append(Node(v, weight))


# Declaring the adjacency list
adj = []
# Greedy best first search algorithm function
def GBFS(h, V, src, dest)

# Initializing openList and closeList.
openList = []
closeList = []

# Inserting src in openList.
openList.append(pathNode(src, None))

# Iterating while the openList
# is not empty.
while (openList):

currentNode = openList[0]
currentIndex = 0
# Finding the node with the least 'h' value
for i in range(len(openList)):
if(h[openList[i].node] < h[currentNode.node]):
currentNode = openList[i]
```

```
currentIndex = i

# Removing the currentNode from
# the openList and adding it in
# the closeList.
openList.pop(currentIndex)
closeList.append(currentNode)

# If we have reached the destination node.
if(currentNode.node == dest):
# Initializing the 'path' list.
path = []
cur = currentNode

# Adding all the nodes in the
# path list through which we have
# reached to dest.
while(cur != None):
path.append(cur.node)
cur = cur.parent


# Reversing the path, because
# currently it denotes path
# from dest to src.
path.reverse()
return path


# Iterating over adjacents of 'currentNode'
# and adding them to openList if
# they are neither in openList or closeList.
for node in adj[currentNode.node]:
for x in openList:
if(x.node == node.v):
continue

for x in closeList:
if(x.node == node.v):
continue

openList.append(pathNode(node.v, currentNode))
```

```
    return []

    # Driver Code
    """ Making the following graph
                 src = 0
                / | \
               /  |  \
              1   2   3
             /\   |   /\
            /  \  |  /  \
            4   5 6 7    8
                   /
                  /
                 dest = 9
    """
    # The total number of vertices.
    V = 10
    ## Initializing the adjacency list
    for i in range(V):
        adj.append([])

    addEdge(0, 1, 2)
    addEdge(0, 2, 1)
    addEdge(0, 3, 10)
    addEdge(1, 4, 3)
    addEdge(1, 5, 2)
    addEdge(2, 6, 9)
    addEdge(3, 7, 5)
    addEdge(3, 8, 2)
    addEdge(7, 9, 5)

    # Defining the heuristic values for each node.
    h = [20, 22, 21, 10, 25, 24, 30, 5, 12, 0]
    path = GBFS(h, V, 0, 9)
    for i in range(len(path) - 1):
        print(path[i], end = " -> ")

    print(path[(len(path)-1)])
```

Code Output: -




```
0 -> 3 -> 7 -> 9
```