CONTNET

LIST OF FIGURES **7**

LIST OF ABBREVATIONS **8**

ABSTRACT **9**

INTRODUCTION **10-22**

1.1 Graph Theory……..10
1.1.A Graph and Traversal……..10
1.1.1 BFS(Breadth-First Search) Algorithms……...10
1. Working of BFS……..10
2. BFS Principles……...10
3. BFS Algorithm Step-by-Step……..11
4. Disadvantages of BFS for Robotic path planning.....11
1.1.2 DFS(Depth First Search) Algorithms.....12
1. Recursive DFS.....12
2. Iterative DFS.....12
3. Disadvantages Of DFS For Robotic Path Planning.....13
1.1.B Graph and Shortest path.....14
1.1 Key Shortest path Algorithms.....14
1.1.1 Bellman-Ford Algorithm.....14
1.1.2 Floyd-Warshall Algorithm.....14
1.1.3 A*(Star) Algorithm.....14
1.Working of A* Algorithm.....14
2. Path for A* Algorithm.....15
3. Disadvantages Of A* For robotic Path planning.....15
1.1.4 Dijkstra's Algorithm.....15
1. Algorithm.....15
2. Shortest Path Problem.....16
1.2 Differences.....17
1.3 SLAM.....18
1.3.A Types of SLAM.....18
1. Lidar-based SLAM.....18
2. Multi-Sensor SLAM.....18
3. 3D SLAM.....18
4. Visual SLAM.....18
1.How Visual SLAM Works.....18
2. Types of Visual Slam.....19
3. Key Algorithms Visual SLAM.....19
1. LSD-SLAM.....19
1.1 Key concepts.....19
1.Monocular Vision.....19
2. Direct SLAM.....19
3. semi-Dense Map.....19
4. Keyframe-Based Approach.....19
5. Optimization and Pose Estimation.....20
1.2 Components of LSD-SLAM.....20
1. Tracking.....20
2. Mapping.....20
3. Loop Closure.....20
4. Global Optimization.....20

2. ORB-SLAM.....20
2.1 Key Concepts.....21
1. Feature-Based SLAM.....21
2.Monocular, Stereo, and RGB-D Capabilities.....21
3. Keyframe-Based SLAM.....21
4. Map Points and Tracking.....21
5. Loop Closure Detection.....22
2.2 Components of ORB-SLAM.....22
1. Tracking.....22
2. Local Mapping.....22
3. Loop Closure.....22
4. Relocalization.....22
ROBOTIC PATH PLANNING METHODS.....23
2.1 METHODS.....23
2.1.1 Graph-Based Methods.....23
2.1.2 Grid-Based Methods.....23
COMPONENTS.....24
3.1 Software Required.....24
1. Spyder 6.0.0.....24
FLOW CHART 25
4.1 Flow chart for Robotic Path planning.....25
Robotic Path Planning for ORB-SLAM Map.....26-31
5.1 CODE.....26
5.2 OBSERVATIONS / RESULTS.....31
5.2.1 Observation.....31
5.3 Challenges.....31
CONCLUSION.....32
FUTURE SCOPE.....33
BIBLIOGRAPHY.....34

LIST OF FIGURES

Fig 1.1 BFS Principles

Fig 1.2 BFS Algorithm Graph

Fig 1.3 Depth First Traversal

Fig 1.4 Iterative DFS

Fig 1.5 A* Algorithm graph structure

Fig1.6 path for A* Algorithm tree structure

Fig 1.7 Dijkstra's directed graph

Fig 1.8 Shortest path

Fig 1.9 FAST

Fig 1.10 Brief

Fig 1.11 ORB SLAM

Fig 2.1 Occupancy Grid Mapping

Fig 5.1 Robotic path planning for ORB-SLAM map

LIST OF ABBREVATIONS

BFS - Breadth-First Search

DFS - Depth First Search

SLAM - Simultaneous Localization and Mapping

LSD SLAM - Large-Scale Direct Simultaneous Localization and Mapping

FAST - Features from Accelerated Segment Test

BRIEF - Binary Robust Independent Elementary Features

ORB SLAM - Oriented FAST and Rotated BRIEF

ABSTRACT

Autonomous vehicle development has become trending technology in recent times. Many of these systems use SLAM concept to create maps of the surroundings simultaneously with the location. This helps in autonomous navigation. The navigation depends on the method or algorithm used for finding optimized path.

To navigate, a start point, a goal point and connecting path to be detected along with obstacles avoidance. The same work can be done by Dijkstra which is a graph-based algorithm.

CHAPTER- 1

INTRODUCTION

Robotic path planning

Robotic path planning is a fundamental problem in robotics. It is essential for many tasks, such as navigation, manipulation, and assembly. The goal of path planning is to find a sequence of robot movements that allows the robot to reach a desired goal.

1.1 Graph Theory

The relationship between graphs, shortest path algorithms, and graph traversal techniques is key in understanding how to navigate, analyze, and optimize paths in networks, structures, and systems.

1.1.A Graph and Traversal:

Graph traversal refers to visiting all the vertices or edges in a graph in a systematic manner. Traversal is often needed when you want to search for elements, inspect connections, or explore all parts of a network.

There are two primary methods of graph traversal:

1.1.1 BFS (Breadth-First Search) Algorithms

Breadth-First Search(BFS) is a fundamental algorithm used to traverse and explore graph-based data structures. It systematically visits all the vertices of a graph, exploring all the neighbors at the present depth before moving on the neighbours at the next depth level.

1. Working of BFS

- **Start Node:** The algorithm begins at a specified starting node.
- **Explore Neighbours:** It then explores all the neighboring nodes at the current depth level.
- **Queue Management:** Nodes are added to a queue to be visited in the order they were discovered.

2.BFS Principles:

- **Systematic Exploration:** BFS visits all nodes at the current depth before moving to the next depth level.
- **Shortest Path:** BFS guarantees finding the shortest path between the starting node and any other reachable node.
- **Memory Efficient:** BFS can be implemented using a queue data structure, which is memory efficient.

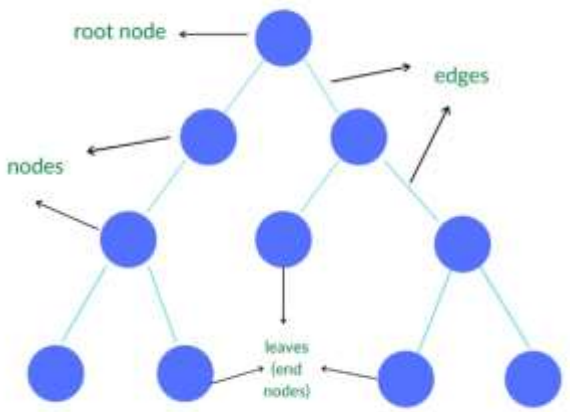


Fig 1.1 BFS Principles

3.BFS Algorithm Step-by-Step

- **Initialize:** Mark the starting node as visited and enqueue it.
- **Dequeue:** Dequeue a node from the queue.
- **Explore Neighbours:** Visit all unvisited neighbours of the dequeue node and enqueue them.

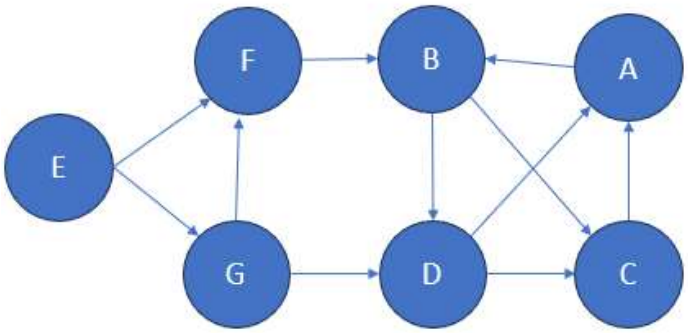


Fig 1.2 BFS Algorithm Graph

4.Disadvantages of BFS for Robotic Path Planning

- High Memory Usage
- Exploration of Unnecessary Areas
- Handling of Dynamic Obstacles
- Slow in Large and Complex Environments

1.1.2 DFS(Depth First Search) Algorithms

Depth-First Search (DFS) is a fundamental graph traversal algorithm that explores as far as possible along each branch before backtracking. It visits all the vertices of a graph in a depth-oriented manner, systematically visiting one child of each vertex before visiting its siblings.

Here the two types in DFS Algorithm

1. Recursive DFS

Recursive DFS uses a stack-based approach, where the algorithm makes recursive calls to explore deeper into the graph. This implementation is often simpler to understand and code, but can consume significant memory for large or deep graphs.

- **Visit Node:** Mark the current node as visited.
- **Explore Neighbours:** Recursively call DFS on each unvisited neighbour.
- **Backtrack:** When all neighbours have been explored, return from the recursive call.

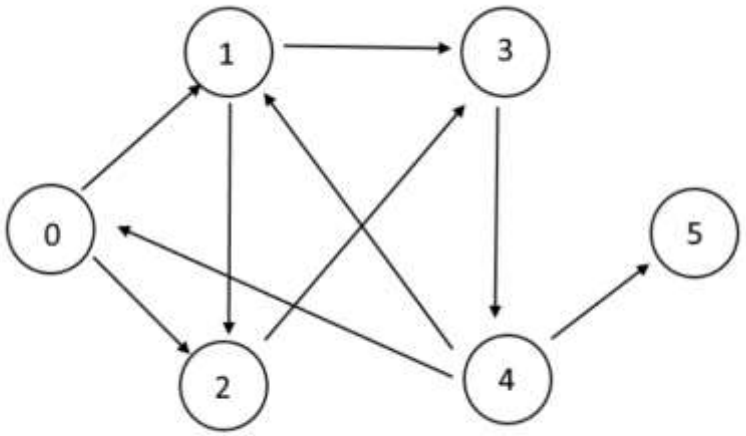


Fig 1.3 Depth First Traversal - 0 1 3 4 5 2

2. Iterative DFS

Iterative DFS uses an explicit stack to keep track of the nodes to visit, instead of relying on the call stack. This can be more memory-efficient for deep or wide graphs, but the code may be more complex**.**

- **Initialize Stack:** Start with the first node and push it on to the stack.
- **Visit Neighbours:** Pop a node from the stack, mark it as visited, and push its unvisited neighbours on to the stack.
- **Repeat:** Continue this process until the stack is empty, indicating all nodes have been visited.

Example

Here in the given tree, the starting node is A and the depth initialized to 0. The goal node is R where we have to find the depth and the path to reach it. The depth from the figure. In this example, we consider the tree as a finite tree, while we can consider the same procedure for the infinite tree as well. We knew that in the algorithm of IDDFS we first do DFS till a specified depth and then increase the depth at each loop. This special step forms the part of DLS or Depth Limited Search. Thus the following traversal shows the IDDFS search.

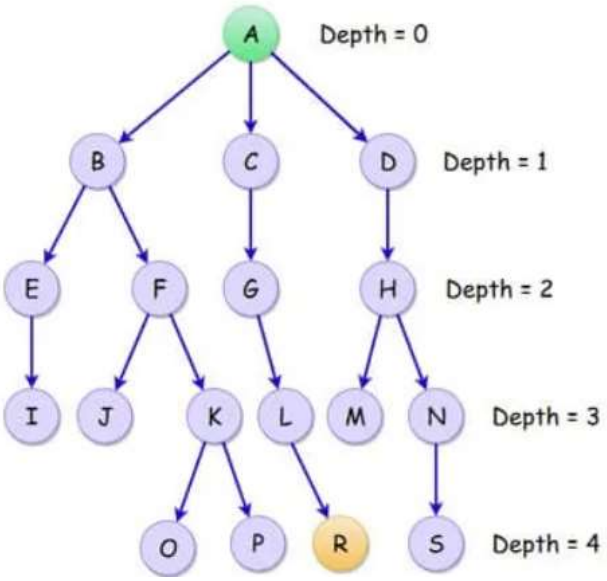


Fig 1.4 Iterative DFS

The tree can be visited as: A B E F C G D H

DEPTH = {0, 1, 2, 3, 4}

DEPTH LIMITS	IDDFS
0	A
1	A B C D
2	A B E F C G D H
3	A B E I F J K C G L D H M N
4	A B E I F J K O P C G L R D H M N S

3. Disadvantages Of DFS For Robotic Path Planning

- Getting Stuck in Dead Ends
- Inefficient search in large/complex spaces
- No Guarantee of optimal solution
- Difficulty in Dynamic Environments
- Infinite loops in unbounded or cyclic graphs

1.1.B Graph and Shortest Path

The **shortest path** in a graph refers to the minimum distance or the least cost to travel between two vertices. This concept is crucial when you want to find the most efficient route from one node to another, especially in real-world problems like finding the quickest way to travel between cities or determining the fastest route in a network.

- **Weighted Graphs:** Shortest path algorithms are especially relevant in weighted graphs where each edge has a weight or cost. The goal is to minimize the sum of these weights along the path.
- **Unweighted Graphs:** In unweighted graphs, the shortest path simply means the one with the fewest edges.

1.1 Key Shortest Path Algorithms

1.1.1 Bellman-Ford Algorithm

Handles graphs with negative edge weights, and can also detect negative weight cycles.

1.1.2Floyd-Warshall Algorithm

Finds the shortest paths between all pairs of vertices in a graph (used in dense graphs).

1.1.3 A*(star) Algorithm

The A* algorithm is a pathfinding algorithm that uses a heuristic function to estimate the cost of reaching the goal. It finds the shortest path from a starting node to a goal node in a graph.

1. Working of A* Algorithm

F(n)=G(n)+H(n)

- **F(n):** The estimate of the total cost from start node to target node through node ‘n’.
- **G(n):** Actual cost from start node to node ‘n’.
- **H(n):** Estimated cost from node ‘n’ to target node.

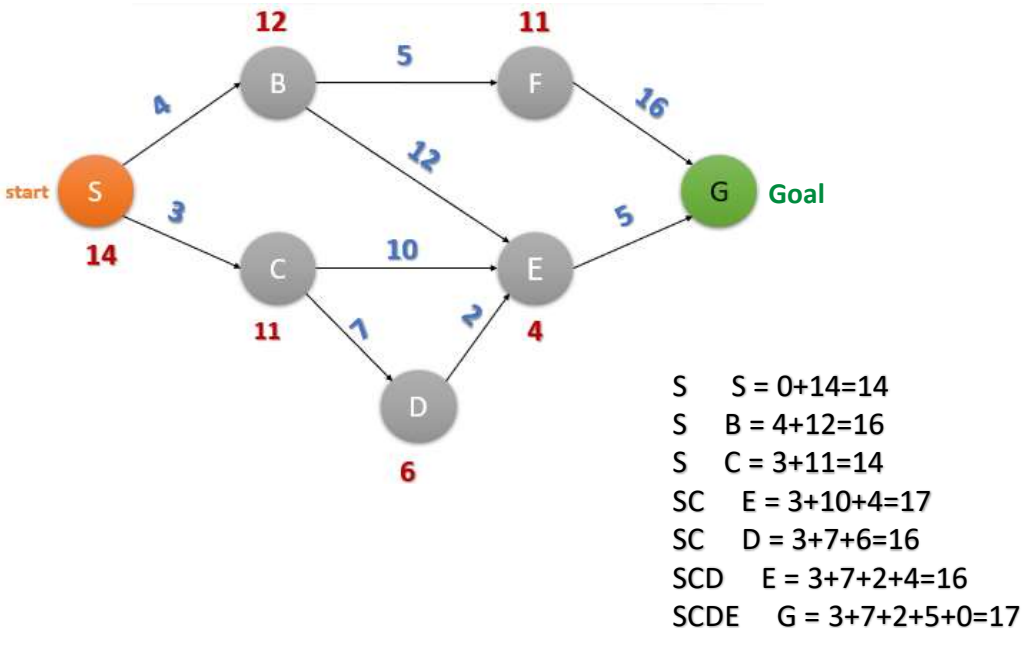


Fig 1.5 A* Algorithm graph structure

2. Path for A*algorithm

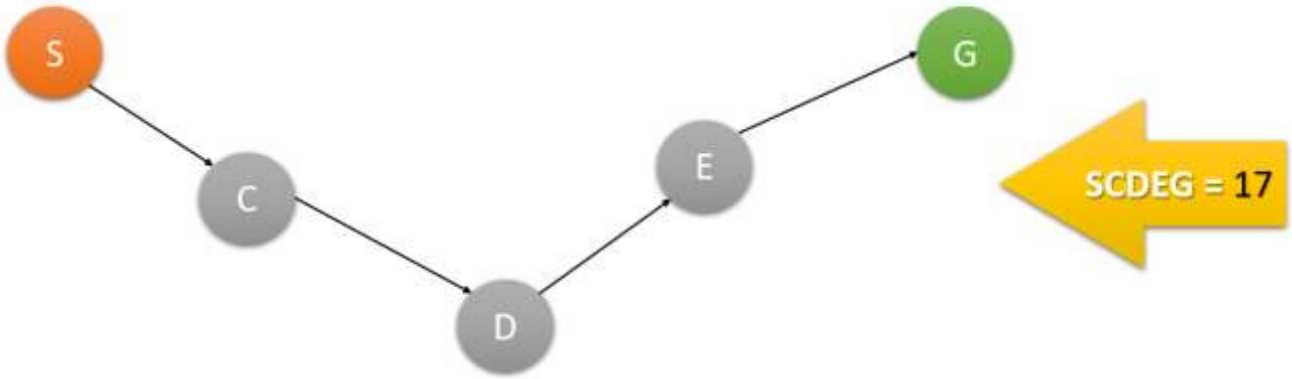


Fig 1.6 path for A* Algorithm tree structure

3. Disadvantages Of A* For Robotic Path Planning

- High Computational Cost
- Memory Limitations
- Dynamic Obstacles

1.1.4 Dijkstra's Algorithm

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a weighted graph, which may represent, for example, road networks.

- By using this we can find the shortest path from the source to all vertices in the given graph.
- It is very near to prim's algorithm.
- Like prim's we generate a shortest path tree.

1. Algorithm

- **Step1:** create a set of shortest path tree set.
- **Step2:** Assign a distance value to all vertices in the input graph.
  - Initialize all distance value as ' inf '.
  - Assign '0' for the source vertex.
- **Step3:** Pick a vertex as 'u' which is not there in shortest set and has a minimum distance value.
  - Include 'u' to shortest set.
  - **d(u)+c(u,v)<d(v)**

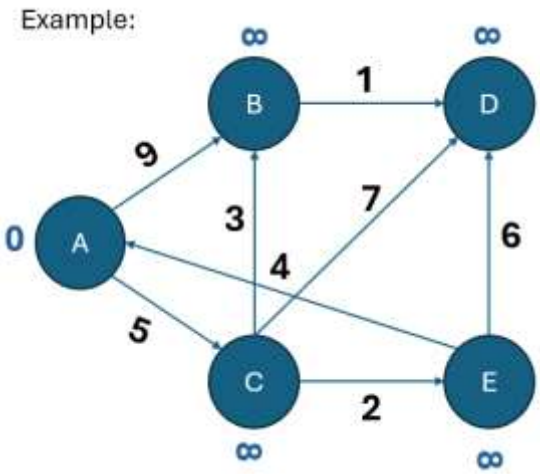


Fig1.7 Dijkstra's directed graph

2. Shortest Path Problem

- **Identifying Optimal Routes:** Dijkstra's algorithm helps solve the problem of finding the shortest or most efficient path between two points in a network.
- **Weighted Connections:** It considers the weights or costs associated with each edge in the graph, allowing for more realistic path planning.

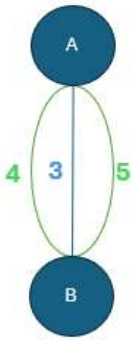


Fig 1.8 Shortest path

1.2 Differences

Feature	BFS (Breadth-First Search)	DFS (Depth-First Search)	A*	Dijkstra's Algorithm
Type	Uninformed Search Algorithm	Uninformed Search Algorithm	Informed Search Algorithm	Uninformed Search Algorithm
Goal	Explore the shortest path in terms of edges in unweighted graphs	Explore as deep as possible in the search tree	Find the shortest path with heuristic	Find the shortest path in weighted graphs
Graph Type	Works best for unweighted or uniformly weighted graphs	Works on all graphs, no preference on weights	Works on weighted graphs	Works on weighted graphs
Heuristic	None	None	Uses a heuristic function (h(n))	None
Exploration Approach	Level by level (breadth-wise)	Deeper nodes first (depth-wise)	Combines cost-so-far (g(n)) + heuristic (h(n))	Expands nodes based on current shortest known distance
Time Complexity	O(V + E)	O(V + E)	O((V + E) * log V)	O((V + E) * log V)
Space Complexity	O(V)	O(V)	O(V)	O(V)

Dijkstra algorithm covers all nodes and gives shortest path than the A* algorithm that is the reason to choose this algorithm in this project.

Reason

To identify the shortest path for the robot we have to know the map for that we are using the Slam, Visual Slam, ORB slam

1.3 SLAM

SLAM (Simultaneous Localization and Mapping) is a process used in robotics and autonomous systems that enables a robot or device to build a map of an unknown environment while simultaneously tracking its own location within that map. This is crucial for mobile robots and autonomous vehicles to navigate in environments where GPS may be unavailable or inaccurate.

1.3.A Types of SLAM

1. Lidar-based SLAM

- Uses Lidar (Light Detection and Ranging) sensors to measure distances and generate a 3D or 2D map of the environment.
- Example: Google Cartographer, Hector SLAM, LOAM (Lidar Odometry and Mapping).

2. Multi-Sensor SLAM

- Combines data from different types of sensors (e.g., Lidar, cameras, GPS, IMU) to enhance accuracy and robustness in challenging environments.
- This fusion approach can help in complex outdoor or GPS-denied environments.

3. 3D SLAM

- Primarily focuses on creating a three-dimensional representation of the environment using multiple types of sensors (Lidar, stereo camera's, etc.).
- Used in autonomous vehicles, drones, and large-scale robotics applications.

4. Visual SLAM

Visual SLAM (Simultaneous Localization and Mapping) is a technology used in robotics and computer vison to create a map of an environment using visual information from camera's, while simultaneously determining the device's location within that map. Visual SLAM systems primarily rely on camera data to estimate the motion of the device (e.g., a robot, drone, or AR device) and build a map of the surrounding environment.

Here's a detailed breakdown of Visual SLAM

1. How Visual SLAM Works

- **Data Collection:** The system collects data from one or more cameras. The cameras capture a continuous stream of images, often at high frame rates.
- **Feature Extraction:** Key features (points of interest like edges or corners) are extracted from the images. This helps the system track changes between frames.
- **Feature Matching:** The system looks for correspondences between features in consecutive images. By matching these features, the system can estimate the movement of the camera in 3D space.
- **Pose Estimation:** Using the matched features, the system estimates the relative motion of the camera between frames. This step determines the camera's pose (its position and orientation).
- **Map Construction:** As the system moves through the environment, it continuously updates a map, typically represented as a point cloud or a sparse 3D structure.
- **Optimization:** To improve accuracy, Visual SLAM uses optimization techniques like bundle adjustment or pose graph optimization to refine the estimated poses and map.

2. Types of Visual SLAM

- **Monocular SLAM:** Uses a single camera to track movement and build a map. It's more lightweight and flexible but can have issues with scale estimation, meaning it can't directly measure depth.
- **Stereo SLAM:** Utilizes two cameras (stereo vision) to estimate depth by comparing the two images. This method improves accuracy, particularly for 3D mapping.
- **RGB-D SLAM:** Incorporates depth sensors along with RGB cameras. The depth sensor provides direct distance measurements, which makes it more robust in creating 3D maps and detecting obstacles. Devices like the Microsoft Kinect are examples of RGB-D sensors used in SLAM.

3. Key Algorithms in Visual SLAM

1. LSD-SLAM

LSD-SLAM (Large-Scale Direct Simultaneous Localization and Mapping) is a real-time monocular SLAM algorithm, developed primarily for visual odometry and 3D scene reconstruction. It is designed to allow a camera (such as one on a mobile robot or a smartphone) to understand its motion through an environment and create a dense, semi-dense, or sparse 3D map of that environment.

Here's how LSD-SLAM works:

1.1 Key Concepts

1.Monocular Vision

- LSD-SLAM only uses a single camera to perceive the world (monocular SLAM). This is a challenging task since monocular systems do not have direct depth information like stereo cameras, which require estimating depth from motion.

2. Direct SLAM

- Unlike feature-based SLAM methods (which rely on keypoints, descriptors, and matching), LSD-SLAM is a direct SLAM approach. It uses the intensity values of pixels directly from the images, without extracting features like corners or edges.
- It works by aligning the brightness patterns across multiple frames to estimate camera movement, which makes it more robust in environments with less texture.

3. Semi-Dense Map

- LSD-SLAM creates a semi-dense map of the environment. Instead of reconstructing the entire scene, it focuses on reconstructing only parts of the scene where information is more reliable (e.g., areas with sufficient texture). This makes it faster than fully dense methods but still more detailed than sparse feature-based approaches.

4. Keyframe-Based Approach

- LSD –SLAM selects keyframes (important frames) to build the map and track the camera's motion. The algorithm continuously estimates the pose of the camera and refines the map by using these keyframes.

5. Optimization and Pose Estimation

- The camera's motion is estimated by minimizing photometric errors (differences in pixel intensities) between consecutive frames.
- Pose optimization ensures accurate tracking over time by adjusting the camera's estimated trajectory to minimize the error.

1.2 Components of LSD-SLAM

1. Tracking

- The camera's motion between frames is tracked by aligning the intensity patterns from the current frame to a reference keyframe.

2. Mapping

- It creates a semi-dense map by estimating depth for a subset of pixels in the keyframes, resulting in a 3D representation of the environment.

3. Loop Closure

- LSD-SLAM supports loop closure, where it recognizes previously visited locations. This helps to correct drift in the camera's position estimates and improve the overall accuracy of the map.

4. Global Optimization

- Once a loop closure is detected, a global optimization step refines the entire map and trajectory to ensure consistency.

2. ORB-SLAM

ORB-SLAM (Oriented FAST and Rotated BRIEF Simultaneous Localization and Mapping) is a feature-based, real-time SLAM system designed to perform localization and mapping using monocular, stereo, or RGB-D cameras. It is widely used in robotics, AR, and other applications that require understanding of a 3D environment.

- **FAST**

  FAST(**Features from Accelerated Segment Test)** is a widely-used algorithm in computer vision for detecting **corner point** in an image. It was developed for its speed and efficiency in detecting corners in real-time applications, like object tracking, 3D reconstruction, and feature matching.

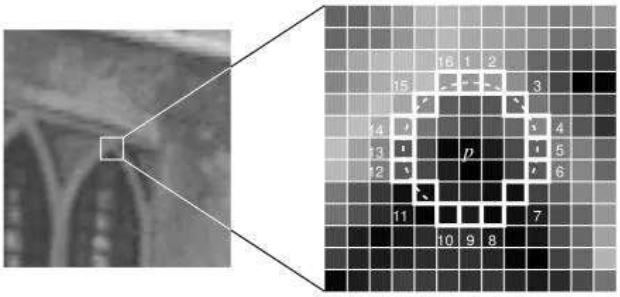


Fig 1.9 FAST

- **BRIEF**

  BRIEF (Binary Robust Independent Elementary Features) is a **feature descriptor** algorithm used in computer vision. It is designed to **extract binary descriptors** for keypoints (features) in an image, allowing for **fast and efficient image matching** and **feature recognition.**

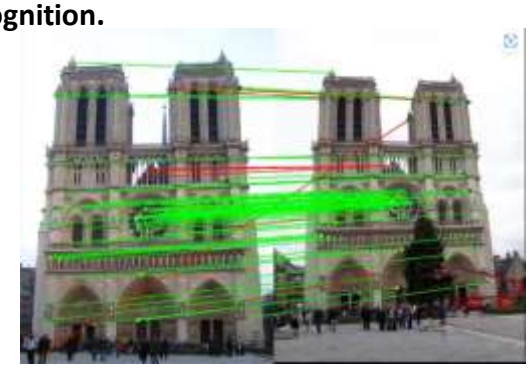


Fig 1.10 Brief

2.1 Key Concepts

1. Feature-Based SLAM

- ORB-SLAM uses features (keypoints) from the image to track the motion of the camera and construct the map. These features are detected and described using the ORB (Oriented FAST and Rotated BRIEF) algorithm, which provides fast and robust keypoint detection and description.
- ORB features are computationally efficient and rotation- and scale-invariant, making them suitable for real-time applications on devices with limited resources, like mobile robots or smartphones.

2. Monocular, Stereo, and RGB-D Capabilities

- ORB-SLAM can operate with
  - **Monocular Cameras**: Where depth has to be estimated from motion.
  - **Stereo Cameras**: Where depth is directly obtained from two camera viewpoints.
  - **RGB-D Cameras**: Where depth information is provided alongside color information.

3. Keyframe-Based SLAM

- Similar to LSD-SLAM, ORB-SLAM uses a keyframe-based approach. It selects certain frames as keyframes and stores the camera's pose and the 3D points in the map.
- Between keyframes, the system tracks the camera's position by matching ORB features between consecutive frames and previously stored keyframes.

4. Map Points and Tracking

- Map points are created from the keyframes, representing the 3D structure of the environment.
- ORB-SLAM uses local bundle adjustment, which optimizes the map and camera poses locally around the current position to refine accuracy.

5. Loop Closure Detection

- ORB-SLAM has a built-in loop closure detection system. When the camera revisits a previously mapped location, ORB-SLAM detects the loop by matching features in the current frame with features from past keyframes.
- When a loop closure is detected, the system corrects drift by performing pose graph optimization, which adjusts the trajectory of the camera and the map to ensure global consistency.

2.2 Components of ORB-SLAM

1. Tracking

- ORB-SLAM continuously tracks the camera's pose by matching ORB features between the current frame and previous keyframes.

2. Local Mapping

- The local mapping process adds new map points to the 3D reconstruction based on keyframes and refines them using local bundle adjustment.

3. Loop Closure

- Detecting when the camera revisits a location is a key strength of ORB-SLAM. When this happens, loop closure is performed, and the entire map is adjusted for consistency.

4. Relocalization

- If the system loses track of the camera, relocalization uses past keyframes to recover the camera's pose by matching features from the current view to the map.

Fig 1.11 ORB SLAM

CHAPTER-2
ROBOTIC PATH PLANNING METHODS

2.1 METHODS

2.1.1 Graph-Based Methods

- **Dijkstra's Algorithm:** A classic graph search algorithm used to find the shortest path between nodes in a graph. It guarantees the shortest path but is computationally expensive for large graphs.

2.1.2 Grid-Based Methods

The sparse map generated by ORB-SLAM, it is sometimes beneficial to convert the map into a grid and perform planning over the discretized space.

- **OCCUPANCY GRID**

  An occupancy grid is a representation commonly used in robotics and autonomous systems to map and understand an environment. It is essentially a two-dimensional or three-dimensional grid where each cell represents a portion of the environment and holds a value corresponding to whether that cell is occupied, free, or unknown.

1. Occupancy Grid Mapping + Dijkstra: ORB-SLAM keyframe data can be used to construct an occupancy grid, where each cell represents the probability of occupancy by an obstacle. Dijkstra can then be used to plan paths through the grid.

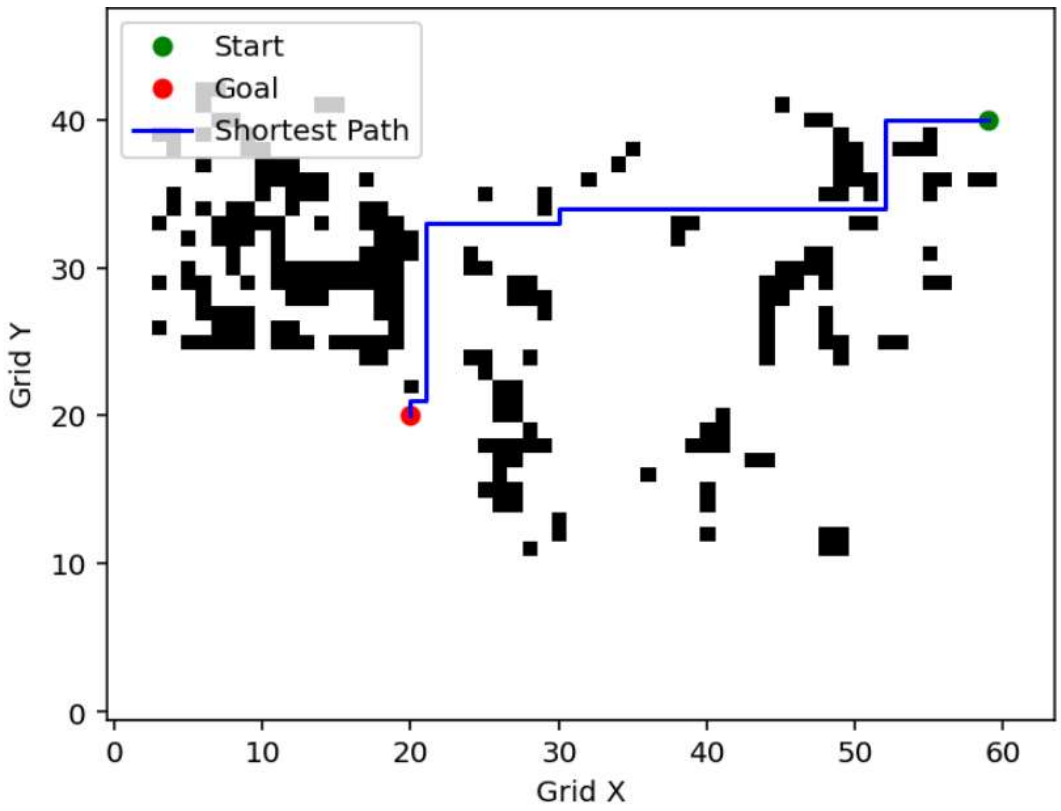


Fig 2.1 Occupancy Grid Mapping

CHAPTER-3
COMPONENTS

3.1 Software Required

1. Spyder 6.0.0

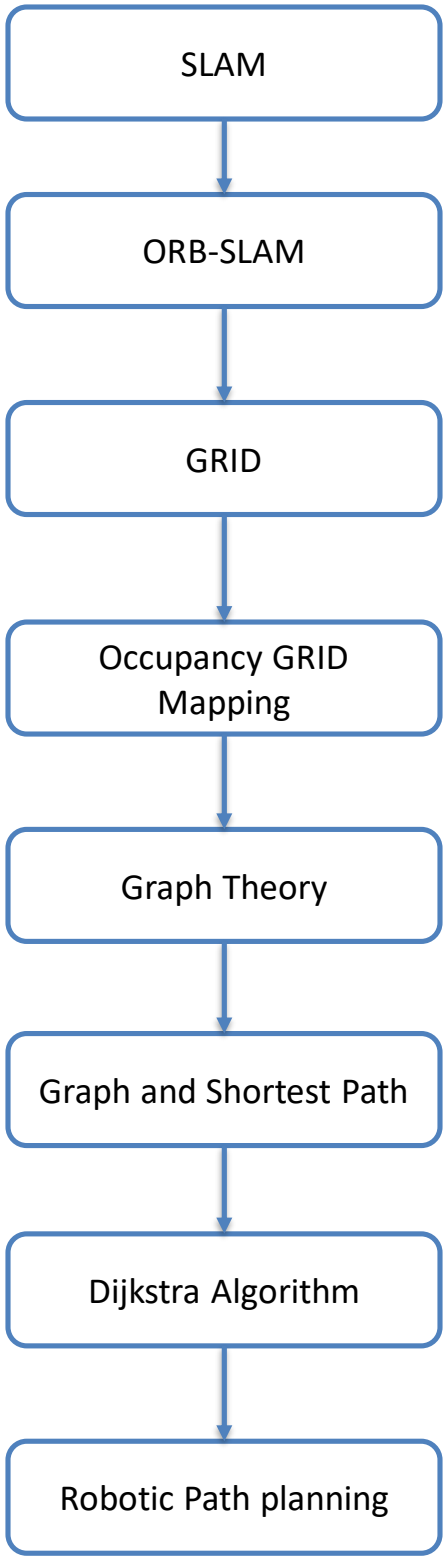
Spyder version 6, released in September 2024, introduces several key improvements, including a revamped Debugger pane, enhanced console management with support for Conda, Pyenv, and remote SSH environments, and better Variable Explorer functionality. The update also brings performance enhancements like faster kernel restarts and improved high-DPI support for visualizations

Spyder version 6 is a major release of the Spyder IDE, designed primarily for scientific computing and data science in Python. It offers a refined user interface, enhanced performance, and integration with modern Python libraries. The update introduces a new debugger based on the debugpy library, improvements in the code editor, and enhanced support for environments like Conda and virtual environments. It also includes better compatibility with Jupyter notebooks and streamlined workflows for data visualization and manipulation.

CHAPTER-4

FLOW CHART

4.1 Flow Chart for Robotic Path Planning



CHAPTER-5

ROBOTIC PATH PLANNING FOR ORB-SLAM MAP

5.1 CODE

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import distance
import heapq
```

- **cv2 (OpenCV)**: This library provides functionality to capture video, detect keypoints, and process images.
- **numpy (np)**: Used for numerical operations, such as creating arrays and manipulating matrices.
- **matplotlib.pyplot**: A plotting library used to visualize the occupancy grid and ORB keypoints.
- **scipy.spatial.distance**: Used to compute Euclidean distance between two points (start and goal).
- **heapq**: Provides a priority queue implementation for Dijkstra's algorithm to find the shortest path.

```
# Initialize ORB Detector
orb = cv2.ORB_create(nfeatures=1000)
```

- ORB (oriented FAST and Rotated BRIEF) is used to detect keypoints in an image. You specify (nfeatures=1000), which limits the detector to detecting a maximum of 1000 keypoints.

```
#create_occupancy_grid
def create_occupancy_grid(keypoints, frame_shape, grid_size=10):
```

- This function creates an **occupancy grid** by dividing the image into a grid of cells and marking which cells contain ORB keypoints.

```
height, width = frame_shape[:2]
grid_height = height // grid_size
grid_width = width // grid_size
```

- **frame_shape[:2]** extracts the height and width of the image (first two elements of the shape tuple).
- **grid_height** and **grid_width** determine how many grid cells fit in the image, given the grid size.

```
occupancy_grid = np.zeros((grid_height, grid_width), dtype=int)
```

- A grid of zeros (representing unoccupied cells) is initialized with dimensions based on the image and the grid size.

```
for kp in keypoints:
    x, y = kp.pt
    grid_x = int(x // grid_size)
    grid_y = int(y // grid_size)

    if 0 <= grid_x < grid_width and 0 <= grid_y < grid_height:
        occupancy_grid[grid_y, grid_x] = 1
```

- For each **keypoint**, its coordinates (**kp.pt**) are divided by the grid size to determine which grid cell it falls into.
- The corresponding cell in the **occupancy grid** is marked as ‘ 1 ’ , indicating it’s occupied by a keypoint

```
# plot_occupancy_grid
def plot_occupancy_grid(grid, start=None, goal=None, path=None):
```

- This function plots the **occupancy grid** and optionally shows the start point, goal point, and path.

```
plt.imshow(grid, cmap='gray_r', origin='lower')
plt.title("Occupancy Grid with Path")
plt.xlabel("Grid X")
plt.ylabel("Grid Y")
```

- **plt.imshow** visualizes the occupancy grid, with occupied cells shown as dark (1) and free cells as light (0).
- **cmap='gray_r'** is the color map used to render the grid in reverse grayscale (0 is white, 1 is black).
- **origin='lower'** makes the origin of the grid appear at the bottom-left of the plot.

```
if start:
    plt.plot(start[0], start[1], 'go', label='Start')
```

- If a **start point** is provided, it is marked with a green circle (**‘ g’** stands for green and circle).

```
if goal:
    plt.plot(goal[0], goal[1], 'ro', label='Goal')
```

- If a **goal point** is provided, it is maked with a red circle (**‘ ro ’** stands for red and circle).

```
if path:
    path_x, path_y = zip(*path)
    plt.plot(path_x, path_y, 'b-', label='Shortest Path')
```

- If a path is provided, the function extracts the x and y coordinates from the path and plots a bule line (**‘ b- ‘)** to represent the shortest path.

```
# compute_distance
def compute_distance(start, goal):
    return distance.euclidean(start, goal)
```

- Computes the **Euclidean distance** between the start and goal points using SciPy's **euclidean** function.

```
# dijkstra
def dijkstra(occupancy_grid, start, goal):
```

- Implements **Dijkstra’s algorithm** to find the shortest path from the start point to the goal point in the occupancy grid. It avoids grid cells marked as obstacles (occupied cells).

```
height, width = occupancy_grid.shape
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
pq = [(0, start)]
dist_map = np.full((height, width), np.inf)
dist_map[start[1], start[0]] = 0
prev = {}
```

- **directions** define the possible movements (up, down, left, right) from any grid cell.
- **pq** is a priority queue storing tuples of the form (distance, (x, y)).
- **dist_map** keeps track of the shortest known distance from the start point to each grid cell.
- **prev** stores the predecessor of each cell to reconstruct the shortest path once the goal is reached.

```
while pq:
    dist, current = heapq.heappop(pq)
    x, y = current
```

- The priority queue is processed by expanding the grid cell with the shortest known distance (**heapq.heappop** pops the minimum distance cell).

```
if (x, y) == goal:
    break
```

- If the goal is reached, the loop terminates.

```
for dx, dy in directions:
    nx, ny = x + dx, y + dy

    if 0 <= nx < width and 0 <= ny < height and
occupancy_grid[ny, nx] == 0:
        new_dist = dist + 1
        if new_dist < dist_map[ny, nx]:
            dist_map[ny, nx] = new_dist
            heapq.heappush(pq, (new_dist, (nx, ny)))
            prev[(nx, ny)] = (x, y)
```

- For each neighbor of the current cell, it checks if the neighbor is a valid and passable cell.
- If a shorter path to the neighbor is found, the **dist_map** is updated, and the neighbor is added to the priority queue.

```
path = []
current = goal
while current in prev:
    path.append(current)
    current = prev[current]
path.append(start)
path.reverse()
```

- After reaching the goal, the path is reconstructed by backtracking from the goal to the start using the **prev** map.

```
# Main Function
def main():
```

- The main function captures video frames, processes them to detect ORB keypoints, and plots the results with the occupancy grid and path.

```
cap = cv2.VideoCapture(0)
```

- Captures video from the default camera (0).

```
while True:
    ret, frame = cap.read()
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    keypoints, descriptors = orb.detectAndCompute(gray_frame,
None)
```

- Continuously captures frames from the camera and converts them to grayscale for ORB keypoint detection.

```
if frame_count % 10 == 0:
    occupancy_grid = create_occupancy_grid(keypoints, gray_frame.shape, grid_size=10)
    start_point_pixel = (595, 400)
    goal_point_pixel = (200, 200)
    start_grid = (start_point_pixel[0] // grid_size, start_point_pixel[1] // grid_size)
    goal_grid = (goal_point_pixel[0] // grid_size, goal_point_pixel[1] // grid_size)
    path = dijkstra(occupancy_grid, start=start_grid, goal=goal_grid)
```

- Every 10 frames, the occupancy grid is created, and the start and goal points are defined. Dijkstra's algorithm is applied to find the shortest path.

```
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
img_with_keypoints = cv2.drawKeypoints(frame, keypoints, None, color=(0, 255, 0), flags=0)
plt.imshow(cv2.cvtColor(img_with_keypoints, cv2.COLOR_BGR2RGB))
plt.title("Real-Time ORB Keypoints")
plt.axis('off')

plt.subplot(1, 2, 2)
plot_occupancy_grid(occupancy_grid, start=start_grid, goal=goal_grid, path=path)
plt.tight_layout()
plt.pause(0.01)
```

- Plots two figures: one showing ORB keypoints on the frame, and the other showing the occupancy grid with the path.

```
cap.release()
```

- Releases the camera when the program finishes.

5.2 OBSERVATIONS /RESULTS

```
Distance between start and goal: 43.83 grid units
Distance between start and goal: 43.83 grid units
Distance between start and goal: 43.83 grid units
Distance between start and goal: 43.83 grid units
Distance between start and goal: 43.83 grid units
Distance between start and goal: 43.83 grid units
Distance between start and goal: 43.83 grid units
Distance between start and goal: 43.83 grid units
Distance between start and goal: 43.83 grid units
Distance between start and goal: 43.83 grid units
Distance between start and goal: 43.83 grid units
Distance between start and goal: 43.83 grid units
Distance between start and goal: 43.83 grid units
```

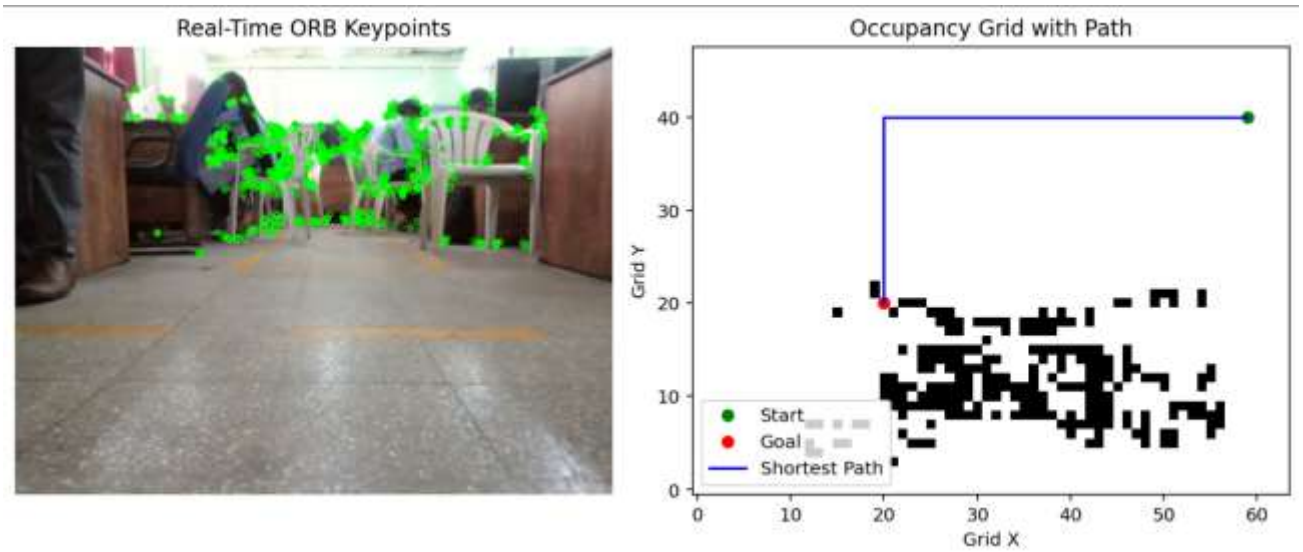


Fig 5.1: Robotic path planning for ORB-SLAM map

5.2.1 Observation:

- This code captures video, detects ORB keypoints, creates an occupancy grid, and finds the shortest path using Dijkstra's algorithm. It provides a visual output of the keypoints and path, along with the grid and distance information.

5.3 challenges : The shadow of the object reflected on the surface. When shadow falls down keypoints are coming on it then robotic path is wrong.

CHAPTER-6
CONCLUSION

Robotic path planning using an ORB-SLAM map effectively combines real-time localization and mapping with efficient pathfinding algorithms to navigate complex environments. ORB-SLAM's ability to generate detailed, accurate maps from visual data allows the robot to understand its surroundings, locate itself precisely, and plan feasible paths towards a goal. By continuously updating the map and adjusting the robot's pose, the system ensures reliable navigation even in dynamic or previously unknown environments. The integration of path planning with obstacle detection further enhances its robustness, making ORB-SLAM a powerful tool for autonomous robotic navigation in real-world applications.

ORB-SLAM offers robots a dynamic understanding of their environment, allowing them to localize themselves in both known and unknown spaces. When integrated with sophisticated path-planning algorithms, this mapping ability transforms simple navigation tasks into intelligent decision-making processes, where robots can explore, learn, and adapt in real-time. These robots are no longer limited by pre-defined routes or static maps but can respond to dynamic environments, anticipating changes, avoiding obstacles, and optimizing paths to achieve specific goals.

This convergence opens doors to a multitude of applications across industries, from self-driving cars navigating chaotic urban streets to drones mapping and exploring uncharted territories. The challenge of path planning in ORB-SLAM is not simply about finding a route—it is about developing systems that understand the environment at a deeper level, incorporating real-time changes, sensor noise, and uncertainty, while still maintaining efficiency and safety.

FUTURE SCOPE

Future of robotic path planning using ORB-SLAM maps is about more than just autonomy—it is about intelligence. As these systems grow in capability, they will not only navigate but also understand their environments in ways that mirror human cognition. This transformation will drive profound changes in how we interact with, rely on, and benefit from autonomous systems in our daily lives, fundamentally altering industries and reshaping the future of robotics. The ability for robots to think, learn, and plan, in real-time, will redefine the boundaries of what is possible in both technology and society.

BIBLIOGRAPHY

ROBOTIC PATH PLANNING FOR ORB-SLAM MAP

- **Wikipedia:** https://www.wikipedia.org
- **ChatGPT 4.0 mini:** https://chat.openai.com
- **Geeks For Geeks:** https://www.geeksforgeeks.org
- **Medium:** https://medium.com