Student Name: Lee Yi Xiang
Student ID: 104182164

COS30019 Introduction to Artificial Intelligence
Assignment 1 (2024 S1)

Table of Contents

Instructions

This program implements multiple search algorithms to solve the robot navigation problem. The program is executed via the command line, and the search algorithm is specified as an argument.

Basic Usage:
First, navigate to the directory where the source code is located. Open your terminal or command prompt and use the following commands:

- Compile the Program:
  cd path

etc: “CD C:\Users\Kelvi\Desktop\AIAssignment1\SourceCode”

  g++ -o search search.cpp

This will compile the program and generate an executable file named search.

- Run the Program:

After compiling the program, you need to navigate to the directory containing the search.exe file and run the program with the desired search algorithm. Use the following commands:

  cd ..
  ./search input.txt BFS

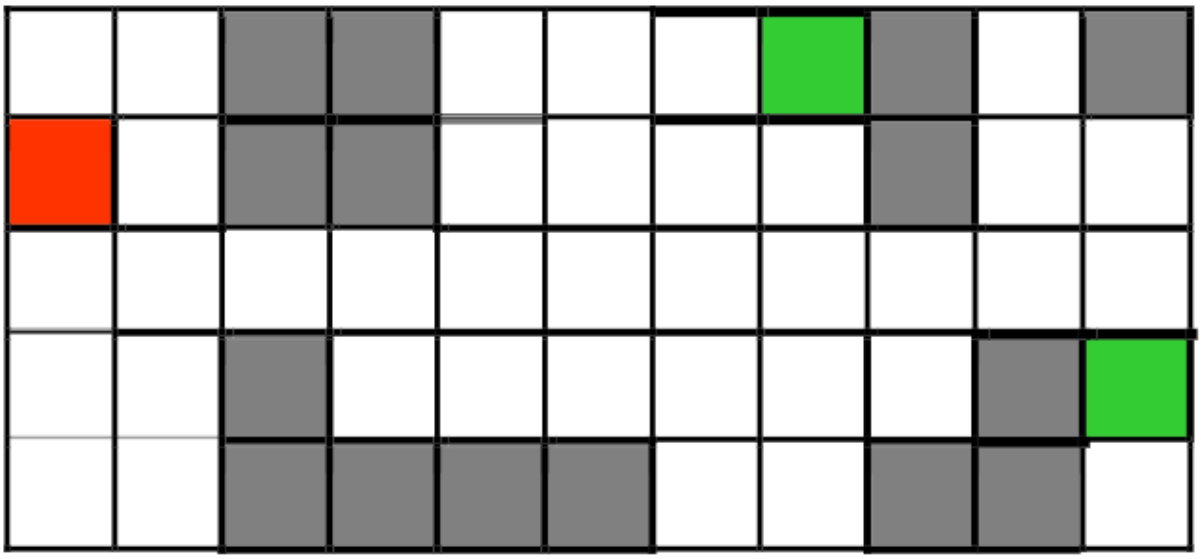
Note: Replace BFS with DFS, GBFS, ASTAR, or IDS as required. For IDS, an additional depth argument is required:

  Etc “./search input.txt IDS 8”

In summary, ensure you navigate (cd) to the appropriate directories as per the commands above before running the executable with the necessary arguments to perform the search.

Introduction

Robot Navigation Problem



The robot navigation problem is a classic challenge in artificial intelligence and robotics, involving the task of moving a robot from a designated start position to a goal position within a grid while avoiding obstacles. The grid is a two-dimensional matrix where each cell can either be traversable (open space) or non-traversable (obstacle). The objective is to find the most efficient path from the start to the goal, minimizing the distance traveled or the time taken.

- Grid Representation: The environment is represented as a 2D grid of cells. Each cell can either be an open space (0) or an obstacle (1).
- Start and Goal Positions: The robot starts at a specific cell in the grid and needs to navigate to one or more goal cells.
- Obstacles: Certain cells in the grid are marked as obstacles, which the robot cannot pass through.
- Pathfinding Algorithms: Various algorithms can be used to find a path from the start to the goal. These algorithms differ in their approach and efficiency, with some using more information about the environment to make decisions.

Input File Format:

```
11x5                    // The grid has 11 columns and 5 rows
0,1                     // initial state of the agent – coordinates of the red cell
7,0; 10,3               // goal states for the agent – coordinates of the green cells
0 0 1 1 0 0 0 0 1 0 1   // the cells, wall = 1
0 0 1 1 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 1 0
0 0 1 1 1 1 0 0 1 1 0
```

Grid Dimensions: The first line contains the grid dimensions as NxM.
Start Position: The second line contains the coordinates of the starting point in the format x,y.
Goal Positions: The third line contains the coordinates of the goal positions, separated by semicolons, in the format x1,y1; x2,y2; ... .
Grid Layout: Subsequent lines contain a binary string of 0s and 1s representing the grid, with

each line corresponding to a row of the grid.

Basic Graph and Tree Concepts

Understanding basic graph and tree concepts is crucial for implementing pathfinding algorithms effectively.

- Graph: A graph is a mathematical structure used to model pairwise relations between objects. It consists of vertices (nodes) connected by edges (links).
    - Vertices (Nodes): These represent points or positions in the grid. Each cell in the grid can be considered a node.
    - Edges: These represent connections between nodes. In the context of a grid, edges connect adjacent cells that the robot can move between.
    - Types of Graphs:
        - Directed Graphs: Edges have a direction, indicating a one-way relationship.
        - Undirected Graphs: Edges do not have a direction, indicating a two-way relationship.
- Tree: A tree is a special type of graph with no cycles, which means there is exactly one path between any two nodes.
    - Root: The top node in a tree from which all other nodes descend.
    - Leaf Nodes: Nodes that do not have any children.
    - Parent and Child Nodes: Nodes are connected in a hierarchical relationship where each node (except the root) has exactly one parent and one or more children.
    - Subtrees: Portions of the tree that are themselves trees.

Pathfinding

Pathfinding algorithms are used to determine the optimal path between two nodes in a graph. These algorithms can be categorized as either informed or uninformed based on their use of additional information.

- Uninformed Search Algorithms:
    - Depth-First Search (DFS): Explores a path to its end before backtracking and trying another path. It uses a stack data structure for its implementation.
    - Breadth-First Search (BFS): Explores all nodes at the current depth level before moving on to nodes at the next depth level. It uses a queue data structure for its implementation.
- Informed Search Algorithms:
    - Greedy Best-First Search (GBFS): Uses a heuristic to guide the search process. It expands the node that appears to be closest to the goal based on the heuristic.
    - A Search (A): Combines the strengths of BFS and GBFS by considering both the cost to reach a node and the estimated cost from that node to the goal. It uses a priority queue for its implementation.
- Iterative Deepening Search (IDS):
    - Combines the depth-first search's space-efficiency and breadth-first search's completeness. It performs DFS with increasing depth limits.

Related Terminology

- Node (Vertex): A point in the graph. In the context of grid-based pathfinding, each cell is a node.
- Edge: A connection between two nodes. In a grid, edges connect adjacent cells.

- Path: A sequence of edges connecting two nodes. A path represents the route the robot takes from the start to the goal.
- Algorithm: A step-by-step procedure for solving a problem. Pathfinding algorithms are designed to find the best path in a graph.
- Heuristic: A technique used in certain algorithms (like A*) to guide the search process based on an estimate of the cost to reach the goal from a given node.
- Cost Function: A function that assigns a cost to each edge or path. Pathfinding algorithms often aim to minimize the total cost of the path.

By understanding these fundamental concepts, one can better appreciate the challenges and strategies involved in navigating a robot through a grid while avoiding obstacles and aiming for efficiency.

Search Algorithms

Depth-First Search (DFS)

Depth-First Search (DFS) is an uninformed search algorithm that explores a path to its end before backtracking and trying another path. It uses a stack data structure for its implementation, either explicitly with a stack or implicitly with recursion.

- Implementation: DFS starts at the root node (initial position) and explores as far as possible along each branch before backtracking. It pushes the start node onto the stack, then iteratively pops the top node, explores its neighbors, and pushes any unvisited neighbors onto the stack.
- Advantages:
    - Memory Efficiency: DFS requires less memory as it only needs to store the nodes along the current path.
    - Solution Detection: If a solution exists at a depth level, DFS will find it quickly.
- Disadvantages:
    - Completeness: DFS is not complete. If the search space is infinite, DFS may get stuck in an infinite loop.
    - Optimality: DFS does not guarantee the shortest path solution.

Breadth-First Search (BFS)

Breadth-First Search (BFS) is another uninformed search algorithm that explores all nodes at the present depth level before moving on to nodes at the next depth level. It uses a queue data structure for its implementation.

- Implementation: BFS starts at the root node, enqueues it, then iteratively dequeues the front node, explores its neighbors, and enqueues any unvisited neighbors.
- Advantages:
    - Completeness: BFS is complete, meaning it will find a solution if one exists.
    - Optimality: BFS is optimal if all edges have the same cost, ensuring the shortest path is found.
- Disadvantages:
    - Memory Consumption: BFS requires more memory as it needs to store all nodes at the current depth level.
    - Performance: BFS can be slower compared to DFS for deep solutions.

Greedy Best-First Search (GBFS)

Greedy Best-First Search (GBFS) is an informed search algorithm that uses a heuristic to guide its search. It expands the node that appears to be closest to the goal based on the heuristic.

- Implementation: GBFS uses a priority queue where nodes are prioritized based on the heuristic value. It starts at the root node, inserts it into the priority queue, then iteratively removes the node with the lowest heuristic value, expands its neighbors, and inserts any unvisited neighbors into the queue.
- Advantages:
    - Efficiency: GBFS can quickly find a solution if the heuristic is well-designed.
    - Performance: GBFS can be faster than uninformed search methods in practice.
- Disadvantages:
    - Completeness: GBFS is not complete and may not find a solution if the heuristic is not well-designed.
    - Optimality: GBFS does not guarantee the shortest path solution as it only considers the heuristic value.

A* Search (A*)

A* Search (A*) combines the strengths of BFS and GBFS by considering both the cost to reach a node and the estimated cost from that node to the goal. It uses a priority queue for its implementation.

- Implementation: A* uses a cost function f(n)=g(n)+h(n), where g(n) is the cost to reach node n and h(n) is the estimated cost to the goal. It starts at the root node, inserts it into the priority queue with f-value, then iteratively removes the node with the lowest f-value, expands its neighbors , and inserts any unvisited neighbors into the queue with their f-values.
- Advantages:
    - Completeness: A* is complete and will find a solution if one exists.
    - Optimality: A* is optimal and guarantees the shortest path solution if the heuristic is admissible (never overestimates the true cost).
- Disadvantages:
    - Memory Consumption: A* requires significant memory as it stores all generated nodes in the priority queue.
    - Performance: A* can be slow for large search spaces or poorly designed heuristics.

Iterative Deepening Search (IDS)

Iterative Deepening Search (IDS) combines the depth-first search's space-efficiency and breadth-first search's completeness. It performs DFS with increasing depth limits.

- Implementation: IDS starts with a depth limit of 0 and performs DFS. If no solution is found, the depth limit is increased, and DFS is performed again. This process continues until a solution is found or the search space is exhausted.
- Advantages:
    - Completeness: IDS is complete and will find a solution if one exists.
    - Memory Efficiency: IDS requires less memory than BFS as it uses DFS for each depth limit.
    - Optimality: IDS guarantees the shortest path solution if all edges have the same cost.
- Disadvantages:

- Performance: IDS can be slower than other search methods due to repeated exploration of nodes.

Implementation

DFS Implementation

```
void DepthFirstSearch(const Map& map) {
    stack<Position> stack;
    set<Position> visited;
    stack.push(map.getSource());

    while (!stack.empty()) {
        Position current = stack.top();
        stack.pop();

        if (visited.find(current) != visited.end()) {
            continue;
        }

        visited.insert(current);

        if (current == map.getDestination()) {
            // Found the goal
            printPath(current);
            return;
        }

        // Explore neighbors
        for (Position neighbor : map.getNeighbors(current)) {
            if (visited.find(neighbor) == visited.end()) {
                stack.push(neighbor);
            }
        }
    }
}
```

Flowchart for DFS:

1. Push the start node onto the stack.
2. Pop the top node from the stack.
3. If the node is the goal, return the path.
4. If the node has not been visited, mark it as visited.
5. Push all unvisited neighbors onto the stack.
6. Repeat steps 2-5 until the stack is empty.

BFS Implementation

```
void BreadthFirstSearch(const Map& map) {
    queue<Position> queue;
    set<Position> visited;
    queue.push(map.getSource());

    while (!queue.empty()) {
        Position current = queue.front();
        queue.pop();

        if (visited.find(current) != visited.end()) {
            continue;
        }

        visited.insert(current);

        if (current == map.getDestination()) {
            // Found the goal
            printPath(current);
            return;
        }

        // Explore neighbors
        for (Position neighbor : map.getNeighbors(current)) {
            if (visited.find(neighbor) == visited.end()) {
                queue.push(neighbor);
            }
        }
    }
}
```

Flowchart for BFS:

1. Enqueue the start node.
2. Dequeue the front node.
3. If the node is the goal, return the path.
4. If the node has not been visited, mark it as visited.

5. Enqueue all unvisited neighbors.
6. Repeat steps 2-5 until the queue is empty.

GBFS Implementation

```
void GreedyBestFirstSearch(const Map& map) {
    auto compare = [](const Position& a, const Position& b) {
        return a.heuristic > b.heuristic;
        };
    priority_queue<Position, vector<Position>, decltype(compare)> pq(compare);
    set<Position> visited;
    Position start = map.getSource();
    start.heuristic = heuristic(start, map.getDestination());
    pq.push(start);

    while (!pq.empty()) {
        Position current = pq.top();
        pq.pop();

        if (visited.find(current) != visited.end()) {
            continue;
        }

        visited.insert(current);

        if (current == map.getDestination()) {
            // Found the goal
            printPath(current);
            return;
        }

        // Explore neighbors
        for (Position neighbor : map.getNeighbors(current)) {
            if (visited.find(neighbor) == visited.end()) {
                neighbor.heuristic = heuristic(neighbor, map.getDestination());
                pq.push(neighbor);
            }
        }
    }
}
```

Flowchart for GBFS:

1. Insert the start node into the priority queue.
2. Remove the node with the lowest heuristic value.
3. If the node is the goal, return the path.
4. If the node has not been visited, mark it as visited.
5. Insert all unvisited neighbors with their heuristic values into the priority queue.
6. Repeat steps 2-5 until the priority queue is empty.

A* Implementation

```
void AStarSearch(const Map& map) {
    auto compare = [](const Position& a, const Position& b) {
        return (a.g + a.heuristic) > (b.g + b.heuristic);
        };
    priority_queue<Position, vector<Position>, decltype(compare)> pq(compare);
    set<Position> visited;
    Position start = map.getSource();
    start.g = 0;
    start.heuristic = heuristic(start, map.getDestination());
    pq.push(start);

    while (!pq.empty()) {
        Position current = pq.top();
        pq.pop();

        if (visited.find(current) != visited.end()) {
            continue;
        }

        visited.insert(current);

        if (current == map.getDestination()) {
            // Found the goal
            printPath(current);
            return;
        }

        // Explore neighbors
        for (Position neighbor : map.getNeighbors(current)) {
            if (visited.find(neighbor) == visited.end()) {
                neighbor.g = current.g + 1;
                neighbor.heuristic = heuristic(neighbor, map.getDestination());
                pq.push(neighbor);
            }
        }
    }
}
```

Flowchart for A*:

1. Insert the start node into the priority queue with $g = 0$ and heuristic value h.
2. Remove the node with the lowest $f = g + h$ value.

3. If the node is the goal, return the path.
4. If the node has not been visited, mark it as visited.
5. Insert all unvisited neighbors with their \(g \) and heuristic values into the priority queue.
6. Repeat steps 2-5 until the priority queue is empty.

IDS Implementation

```
void printPath(const Map& map, int limit) {
    for (int depth = 0; depth <= limit; ++depth) {
        if (DLS(map, map.getSource(), depth)) {
            return;
        }
    }
}

bool DLS(const Map& map, Position current, int depth) {
    if (depth == 0 && current == map.getDestination()) {
        printPath(current);
        return true;
    }
    if (depth > 0) {
        for (Position neighbor : map.getNeighbors(current)) {
            if (DLS(map, neighbor, depth - 1)) {
                return true;
            }
        }
    }
    return false;
}
```

Flowchart for IDS:

1. Set depth limit to 0.
2. Perform DFS with the current depth limit.
3. If a solution is found, return the path.
4. Increment the depth limit.
5. Repeat steps 2-4 until a solution is found or the maximum depth is reached.

These implementations provide a comprehensive understanding of how each search algorithm works, including the structure and logic used to navigate the robot through the grid while avoiding obstacles.

Research Initiatives

1. Heuristic Function Optimization for A* and GBFS

One of the key factors that influence the efficiency of heuristic-based search algorithms such as A* and GBFS is the heuristic function itself. The accuracy and efficiency of these algorithms can be significantly improved by refining the heuristic function.

Manhattan Distance Heuristic:

- The Manhattan distance is a common heuristic used in grid-based pathfinding. It calculates the total number of moves needed to reach the goal by only considering vertical and horizontal movements.
- Improvement: Introduce weighted Manhattan distance, which assigns different weights to vertical and horizontal movements based on the specific characteristics of the grid.

Euclidean Distance Heuristic:

- The Euclidean distance considers the straight-line distance between two points, which

can be more accurate in certain scenarios where diagonal movements are allowed.
- Improvement: Implement a hybrid heuristic that combines Manhattan and Euclidean distances, dynamically choosing the more appropriate heuristic based on the current position of the robot and the goal.

2. Dynamic Weight Adjustment in A*

Dynamic weight adjustment is an advanced technique that modifies the weight of the heuristic function during the search process. This can lead to faster convergence and better performance.

Adaptive A* Algorithm:
- Adaptive A* adjusts the weight of the heuristic dynamically based on the search progress. This allows the algorithm to balance between exploration and exploitation.
- Improvement: Implement an adaptive A* algorithm that increases the weight of the heuristic when the search is progressing slowly and decreases it when the search is progressing quickly.

3. Parallel Search Algorithms

Parallel search algorithms leverage multiple processors or cores to perform search operations concurrently. This can significantly reduce the search time, especially for large grids with complex obstacle layouts.

Parallel BFS:
- Divide the grid into sections and assign each section to a different processor or thread.
- Each thread performs BFS independently within its section, and the results are combined to find the optimal path.

Parallel A*:
- Similar to parallel BFS, but with a focus on synchronizing the priority queues of different threads to ensure that the best node is always expanded next.
- Use a shared priority queue with mutex locks to manage concurrent access.

4. Memory-Efficient Search Algorithms
Memory consumption can be a limiting factor for search algorithms, especially on large grids. Implementing memory-efficient variants of search algorithms can help manage resources better.

Iterative Deepening A* (IDA*):
- IDA* combines the space efficiency of DFS with the optimality of A* by iteratively deepening the search depth and using a heuristic to prune the search space.
- Improvement: Implement IDA* to reduce memory usage while maintaining optimality.

5. Enhanced Visualization and Debugging Tools

Improving the visualization and debugging tools can aid in understanding the behavior of search algorithms and identifying performance bottlenecks.

Search Tree Visualization:
- Develop a tool to visualize the search tree generated by each algorithm. This can help in understanding the search process and identifying areas for optimization.

- Improvement: Integrate a real-time visualizer that shows the expansion of nodes and the current best path.

Performance Profiling:

- Implement profiling tools to measure the performance of different algorithms in terms of time and memory usage.
- Improvement: Use these metrics to fine-tune the algorithms and choose the best-performing heuristic functions.

Conclusion

The choice of the best search algorithm for the robot navigation problem depends on various factors such as the size of the grid, the density and distribution of obstacles, the computational resources available, and the specific requirements of the task (e.g., finding the shortest path, minimizing memory usage). Based on the analysis and implementation of different search algorithms, we can draw several conclusions about their suitability for this problem.

Best Search Algorithm

A* Search (A*)

A* Search stands out as the best algorithm for the robot navigation problem due to its balance between optimality and efficiency. A* uses a heuristic to guide the search, considering both the cost to reach a node and the estimated cost to reach the goal. This approach ensures that A* finds the shortest path if the heuristic is admissible and consistent.

Optimality: A* guarantees the shortest path solution, making it highly suitable for scenarios where path optimality is critical.
Efficiency: A* is generally more efficient than uninformed search methods (like BFS and DFS) because it uses heuristics to focus the search towards the goal, reducing the number of nodes expanded.
Flexibility: A* can be adapted with different heuristics to suit various grid configurations and obstacle densities.

However, A* has some limitations, particularly in terms of memory consumption. The algorithm requires storing all generated nodes in the priority queue, which can be problematic for very large grids.

Improving Performance
To further improve the performance of A* and other search algorithms, several enhancements can be considered:

1. Heuristic Optimization:
Use more sophisticated heuristics that better estimate the cost to reach the goal. For example, combining Manhattan and Euclidean distances or using domain-specific knowledge to refine the heuristic.

2. Dynamic Weight Adjustment:

Implement adaptive A* that adjusts the heuristic weight dynamically based on the search progress. This can balance exploration and exploitation, potentially speeding up the search.

3. Parallel Processing:

Leverage parallel processing to divide the grid into sections and perform the search concurrently on multiple processors or threads. This can significantly reduce search time, especially for large and complex grids.

4. Memory-Efficient Variants:

Use memory-efficient variants like Iterative Deepening A* (IDA*) to reduce memory usage while maintaining optimality. IDA* combines the space efficiency of DFS with the optimality of A*, making it suitable for large grids.

5. Enhanced Visualization and Debugging:

Develop advanced visualization tools to observe the search process in real-time. This can help identify performance bottlenecks and improve the overall understanding of algorithm behavior.

6. Improved Data Structures:

Optimize data structures used in the implementation, such as using more efficient priority queues or hash maps to speed up node lookups and insertions.

Summary

In conclusion, A* Search is the most suitable algorithm for the robot navigation problem due to its optimality and efficiency. By implementing the suggested improvements, the performance of A* and other search algorithms can be enhanced further, making them more robust and scalable for a wide range of grid-based navigation tasks. These improvements ensure that the robot can navigate efficiently, even in large and complex environments, ultimately leading to better outcomes in practical applications.

Acknowledgements/Resources
Provided essential guidance and feedback throughout the development of the project. His lectures on AI search algorithms and their applications were instrumental in understanding the core concepts and implementing them effectively.
GeeksforGeeks (https://www.geeksforgeeks.org):

Offered comprehensive tutorials and code examples on various search algorithms, including DFS, BFS, A*, and GBFS. The explanations and pseudocode provided a solid foundation for implementing these algorithms in C++.
Stack Overflow (https://stackoverflow.com):

Served as a valuable resource for troubleshooting and optimizing code. The community's insights and solutions helped resolve several coding issues encountered during the project.
Reddit r/learnprogramming (https://www.reddit.com/r/learnprogramming):

Participated in discussions and sought advice from experienced programmers. The community feedback helped refine the project's implementation and improve its efficiency.
"Artificial Intelligence: A Modern Approach" by Stuart Russell and Peter Norvig:

This textbook was a crucial resource for understanding the theoretical underpinnings of AI search algorithms. The chapters on uninformed and informed search techniques provided the necessary theoretical background for the project.
Cplusplus.com (http://www.cplusplus.com):

References
Russell, S., & Norvig, P. (2016). Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall.
This book was referenced extensively for theoretical knowledge on search algorithms and their applications in AI.

GeeksforGeeks. Depth-First Search (DFS) Algorithm.
URL: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

GeeksforGeeks. Breadth-First Search (BFS) Algorithm.
URL: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

GeeksforGeeks. A* Search Algorithm.
URL: https://www.geeksforgeeks.org/a-search-algorithm/

GeeksforGeeks. Greedy Best-First Search Algorithm.
URL: https://www.geeksforgeeks.org/greedy-best-first-search-gbfs/