



COS30019
Semester 2024

ASSIGNMENT 1

TREE-BASED SEARCH

Proposal by:

Tran Quoc Dung - 103803891

Contents

1. Instructions	2
2. Introduction	3
3. Search Algorithms	4
3.1. Depth-First Search (DFS)	4
3.2. Breath-First Search (BFS)	5
3.3. Greedy Best-First Search (GBFS)	6
3.4. A Star (A*)	7
3.5. Custom Search Strategy 1 (CUS1)	8
3.6. Custom Search Strategy 2 (CUS2)	10
3.7. Advantages & Disadvantages of Algorithms	11
4. Testing	11
5. Features	14
6. Conclusion	14

1. Instructions

Step 1:

Firstly, my project code is developed & implemented by Visual Studio. Therefore, to open and run our project, Visual Studio is required to be installed.

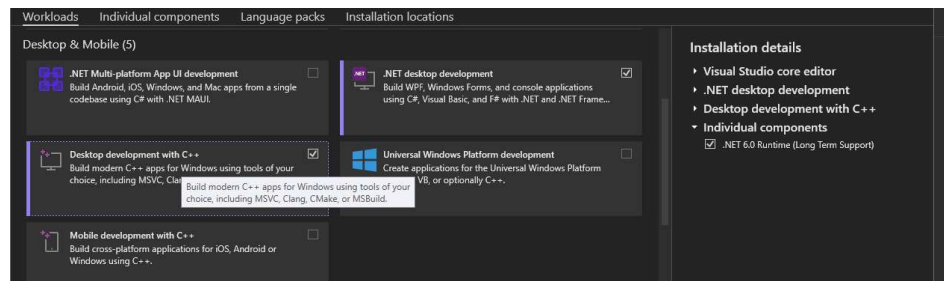
If you have not download Visual Studio, click [here](#) to install.

Step 2:

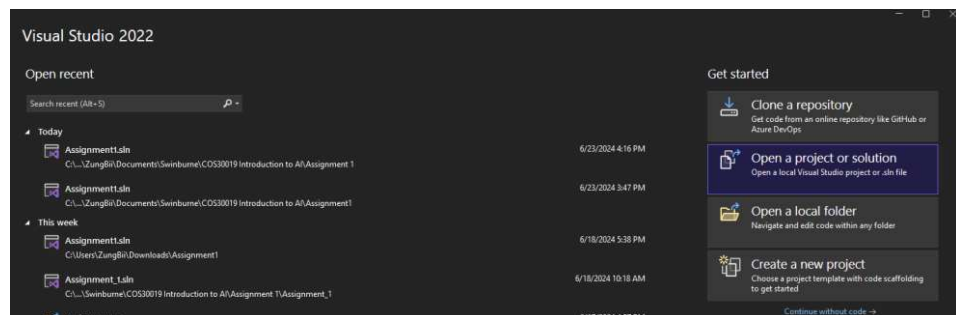
If you just downloaded Visual Studio, remember to choose the “*Desktop Development with C++*” workload, which provide virtual libraries for my project code.

If you have already downloaded it, you can check the workloads by choosing *Tools => Get Tools and Features*.

Remember, this workload “*Desktop Development with C++*” **must be installed**, so the code can run smoothly.

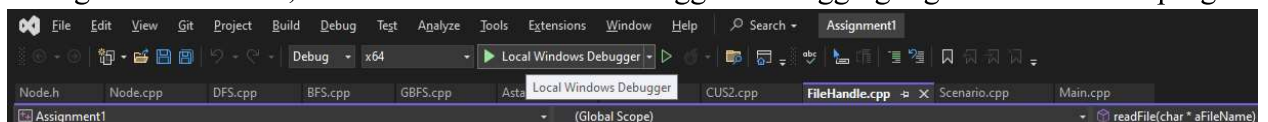


Step 3: Open Visual Studio. Select “*Open a project or solution*”, which will open a project through .sln file.



Then access the “Assignment1.sln” file through this following path: Assignment 1/Assignment1.sln

Step 4: Once the project code is open, you can comprehensively observe the code files, along with running them. Press F5, or this Local Windows Debugger/ Debugging logo here to run the program.



After that, you can see the output of the project, showing respective illustrations of each algorithm.

2. Introduction

The Robot Navigation Problem is a program where a robot must explore an NxM grid to reach a goal located in a random node within the grid. The robot starts on an empty node and moves one node at a time, with only four directions that are right, left, up and down. The grid contains walls that the robot cannot pass through, requiring it to navigate around obstacles. For this assignment, the grid size, agent location, goals, and walls are predefined. The focus is on showcasing theoretical and practical AI skills by illustrating the resolution of intricate pathfinding obstacles in a grid setting using C++ and tree-based search algorithms. By prioritizing algorithmic nuances and strategic AI decision-making, the objective of the document is to offer perspectives on addressing the Robot Navigation issue through detailing the implementation and efficiency of algorithms.

Basic concepts of Robot Navigation:

- Graphs: A collection of vertices connected by edges is referred to as a graph. In the context of the NxM Puzzle, each state can be represented as a vertex, and the edges represent possible transitions between states.
- Trees: The tree is a special kind of graph that starts at a root node and spreads out to leaf nodes. It is a hierarchical structure without any cycles. Trees are essential for representing the search space of the NxM Puzzle, where one node can result in several subsequent states, similar to a branching process.

Glossary:

- Node/Vertex: The fundamental unit in graphs and trees, representing states in the context of NxM Puzzles.
- Edge: A connection between graph nodes, indicating possible moves or transitions within the puzzle.
- Leaf node: A childless node signifying a possible solution conclusion during the search.
- Heuristic: An approach to calculating the cost of achieving a goal from a specific state, used to prioritize paths in intelligent search algorithms.

3. Search Algorithms

3.1. Depth-First Search (DFS)

- Definitions: DFS is an algorithm used for navigating or searching through tree or graph data structures. The algorithm investigates each branch as far as it can after beginning at the root node and then turns around. DFS is unique by means of using a stack with recursion, to manage the nodes under study. Though it may not always identify the shortest path in weighted graphs or the entire exploration in infinite graphs, it performs well when solving puzzles, mazes, and cycles.

- Image 3.1: Depth-First Search (DFS) source code:

```
#include "Scenario.h"

void Scenario::DFS(Node* aSource) const
{
    //Marking the source node as visited, preventing revisitation
    aSource->fVisited = true;

    //If current node is the goal, print the path & stop the search
    if (aSource->fColor == 'G')
    {
        //Initializing the path, with stringstream datatype
        stringstream path;

        //Tracking back, storing the path from the source to this goal node
        printPath(aSource, path);

        // Counting number of steps (By converting "path" to istream)
        istream is(path.str());
        auto numOfSteps = distance(istream_iterator<string>(is), istream_iterator<string>());

        cout << "Number of steps: " << numOfSteps - 4 << endl;
        cout << path.str() << endl;
        cout << "\n";
        return;
    }

    //Explore four possible directions from the current node (up, down, left, right)

    if (isValid(aSource->fX - 1, aSource->fY))
    {
        fNodes[aSource->fX - 1][aSource->fY].fPrevious = aSource;
        DFS(&fNodes[aSource->fX - 1][aSource->fY]);
        fNodes[aSource->fX - 1][aSource->fY].fPrevious = NULL; //Resetting fPrevious
    }

    if (isValid(aSource->fX, aSource->fY - 1))
    {
        fNodes[aSource->fX][aSource->fY - 1].fPrevious = aSource;
        DFS(&fNodes[aSource->fX][aSource->fY - 1]);
        fNodes[aSource->fX][aSource->fY - 1].fPrevious = NULL; //Resetting fPrevious
    }

    if (isValid(aSource->fX + 1, aSource->fY))
    {
        fNodes[aSource->fX + 1][aSource->fY].fPrevious = aSource;
        DFS(&fNodes[aSource->fX + 1][aSource->fY]);
        fNodes[aSource->fX + 1][aSource->fY].fPrevious = NULL; //Resetting fPrevious
    }

    if (isValid(aSource->fX, aSource->fY + 1))
    {
        fNodes[aSource->fX][aSource->fY + 1].fPrevious = aSource;
        DFS(&fNodes[aSource->fX][aSource->fY + 1]);
        fNodes[aSource->fX][aSource->fY + 1].fPrevious = NULL; //Resetting fPrevious
    }
}
```

- In my project, DFS is implemented as a function *Scenario::DFS(Node* aSource) const*, which takes coordinates of *aSource* as inputs, then implements Depth-First Search from that node.
- The code assumes that the grid/maze is represented as a 2D array (*fNodes*) of *Node* objects. Each node is identified by its coordinates (*fX*, *fY*), a route tracking pointer (*fPrevious*), its type and color (*fColor*), and a flag (*fVisited*) that shows if it has been visited.
- The line “*aSource->fVisited = true*” appears as a repeated state checking statement, marking the state of the beginning node as visited.
- The *if* statement, *if (aSource->fColor == “G”)*, means the path, number of steps will be displayed when the agent reaches the goal node.
- Next, the function iteratively explores all nodes by turning up, down, left, or right, that are legitimate moves from the current node. It uses the *isValid* function to determine whether a move is permitted and inside boundaries.
- The *isValid* function is necessary for the method to avoid examining nodes that shouldn't be visited due to the limitations of the challenge or out-of-bounds areas.

3.2. Breath-First Search (BFS)

- Definitions: BFS is also a tree-based search algorithm, quite similar to DFS but with different implementations. Before going on to the nodes at the next depth level, it investigates every node at the current depth starting with the tree root.

- Image 3.2: Breath-First Search (BFS) source code:

```
#include "Scenario.h"

void Scenario::BFS(Node* aSource) const
{
    queue<Node*> Search;

    //Marking the source node as visited, preventing revisitation
    aSource->fVisited = true;

    //Enqueue the source node
    Search.push(aSource);

    // Continue until there are no more nodes to explore
    while (!Search.empty())
    {
        //Get the next node to explore
        Node* temp = Search.front();

        //Remove this node from the queue
        Search.pop();

        //If the current node is the goal, print the path and exit
        if (temp->fColor == 'G')
        {
            //Initializing the path, with stringstream datatype
            stringstream path;

            //Tracking back, storing the path details from the source to this goal node
            printPath(temp, path);

            // Counting number of steps (By converting "path" to istream)
            stringstream is(path.str());
            auto numOfSteps = distance(istream_iterator<string>(is), istream_iterator<string>());

            cout << "Number of steps: " << numOfSteps - 4 << endl;
            cout << path.str() << endl;
            cout << "\n";
        }

        //Explore each neighbor of the current node and enqueue if valid (i.e., within bounds and not visited)
        //Set the current node as the predecessor of each valid neighbor to trace the path later

        if (isValid(temp->fX - 1, temp->fY))
        {
            fNodes[temp->fX - 1][temp->fY].fPrevious = temp;
            fNodes[temp->fX - 1][temp->fY].fVisited = true;
            Search.push(&fNodes[temp->fX - 1][temp->fY]);
        }

        if (isValid(temp->fX, temp->fY - 1))
        {
            fNodes[temp->fX][temp->fY - 1].fPrevious = temp;
            fNodes[temp->fX][temp->fY - 1].fVisited = true;
            Search.push(&fNodes[temp->fX][temp->fY - 1]);
        }

        if (isValid(temp->fX + 1, temp->fY))
        {
            fNodes[temp->fX + 1][temp->fY].fPrevious = temp;
            fNodes[temp->fX + 1][temp->fY].fVisited = true;
            Search.push(&fNodes[temp->fX + 1][temp->fY]);
        }

        if (isValid(temp->fX, temp->fY + 1))
        {
            fNodes[temp->fX][temp->fY + 1].fPrevious = temp;
            fNodes[temp->fX][temp->fY + 1].fVisited = true;
            Search.push(&fNodes[temp->fX][temp->fY + 1]);
        }
    }
}
```

- A queue named *Search* oversees managing the nodes throughout the exploring stage. The source node is marked as visited to prevent it from being re-visited and enqueued from the start of the search.
- The algorithm loops back on itself if there still are nodes in the queue. This implies that the search will go on for as long as there are still accessible nodes that haven't been investigated. The neighbor is noted as visited and added to the search queue. This is a compulsory step in level-by-level exploration process of BFS, since it guarantees that each node at the current depth is examined before moving on to the next level.
- The node at the front of the queue is removed once it has been established that it is the node to examine at this time (temp). If the current node is the goal node (*if* statement), the procedure builds the path to the current node using a *stringstream* and outputs it before exiting the loop.
- The function iteratively explores all nodes by turning up, down, left, or right, that are legitimate moves from the current node. It uses the *isValid* function to determine whether a move is permitted and inside boundaries.

3.3. Greedy Best-First Search (GBFS)

- Definitions: GBFS is an informed search method, which ignores the edge weights in a weighted graph by just considering the heuristic value, with the evaluation function precisely equal to the heuristic function. It expands the node that is closest to the target, as calculated by the heuristic function, to seek for a goal node.

- Image 3.3: Greedy Best-First Search (GBFS) source code:

```
#include "Scenario.h"

//This constructor compare node based on the cost
struct compare {

    //Define priority node with a cost
    bool operator()(const Node* a, const Node* b) { return a->fCost > b->fCost; }
};

void Scenario::GBFS(Node* aSource, Node* aDestination) const {

    //Priority queue for node which prioritized with a cost
    priority_queue<Node*, vector<Node*>, compare> pq;

    //Initialized the cost for the source node
    aSource->fCost = cost(aSource, aDestination);
    pq.push(aSource);

    //Marked the source node as visited
    aSource->fVisited = true;

    //Continues until no nodes left has been visited
    while (!pq.empty()) {

        //Get the nodes with the lowest cost
        Node* temp = pq.top();
        //Remove this node from queue
        pq.pop();

        //Check if the current node is the destination node
        if (temp->fX == aDestination->fX && temp->fY == aDestination->fY) {

            //Initializing the path, with stringstream datatype
            stringstream path;

            //Tracking back, starting the path from the source to this goal node
            printPath(temp, path);

            // Counting number of steps (By converting "path" to stringstream)
            stringstream is(path.str());
            auto numOfSteps = distance(isstream_iterator<string>(is), isstream_iterator<string>());

            cout << "Number of steps: " << numOfSteps << endl;
            cout << path.str() << endl;
            cout << "\n";

            return;
        }

        //Check neighboring nodes in all four directions if they are not visited and within bounds
        int tx = temp->fX;
        int ty = temp->fY;

        //Each if condition checks a direction, updates the node's previous pointer, cost, visited status,
        //and adds it to the priority queue for further exploration.

        if (isValid(tx - 1, ty)) {
            #Nodes[tx - 1][ty].fPrevious = temp;
            #Nodes[tx - 1][ty].fCost = cost(&Nodes[tx - 1][ty], aDestination);
            #Nodes[tx - 1][ty].fVisited = true;
            pq.push(&Nodes[tx - 1][ty]);
        }

        if (isValid(tx, ty - 1)) {
            #Nodes[tx][ty - 1].fPrevious = temp;
            #Nodes[tx][ty - 1].fCost = cost(&Nodes[tx][ty - 1], aDestination);
            #Nodes[tx][ty - 1].fVisited = true;
            pq.push(&Nodes[tx][ty - 1]);
        }
    }
}
```


- According to the algorithm, the agent tends to explore the nodes with the closest distance consecutively, according to the heuristic that defines the cost to the goal node.
- A queue called *priority_queue* is utilized for storing arranged nodes, based on their heuristic ranking.
- The line *aSource->aCost = cost(aSource, aDestination)* helps calculating the distance between the source node to the destination node, which will be used for ranking in queue.
- Nodes in the queue are checked by the algorithm continuously, especially the one with lowest cost to the goal node will be chosen and removed from queue. This process is implemented until the queue is empty.
- This algorithm establishes whether a node eliminated from the queue is the goal node. If so, the path from the source to the destination is created and printed using the *printPath* function.
- If the present node is not the goal node, the agent will continue exploring four neighbors by turning up, down, left, and right. Each case's heuristic cost to the destination is determined, the neighbor is added to the priority queue, the neighbor is marked as visited, and the present node is positioned as the neighbor's predecessor (to trace the path).

3.4. A Star (A*)

- Definitions: A* is an informed search algorithm, which helps finding the shortest path in a graph, from the start node to the goal node. A* algorithm utilizes the features of both GBFS and Uniform Cost Search. By calculating the cost from a node to the goal based on heuristics, A* maximizes efficiency and ensures completeness in its search for the objective.

- Image 3.4: AStar Search (A*) source code:

```

#include "Scenario.h"

//Custom comparator for prioritizing nodes in the priority queue based on their cost and heuristic distance
struct compare
{
    bool operator()(const Node* a, const Node* b)
    {
        //Prioritize by the use of actual cost from the start and the heuristic estimate to the goal
        return (a->aCost + a->fDistance) > (b->aCost + b->fDistance);
    }
};

void Scenario::AStar(Node* aSource, Node* aDestination) const
{
    //Priority queue to manage nodes during search
    priority_queue<Node*, vector<Node*>, compare> pq;

    //Initialize the source node with the heuristic cost to the destination, set its distance to 0, and mark as visited
    aSource->fDistance = 0;
    aSource->fVisited = true;

    //Add the source node to the priority queue
    pq.push(aSource);

    //Direction vectors for exploring orthogonal neighbors (up, down, left, right)
    int dx[] = { -1, 0, 1, 0 };
    int dy[] = { 0, -1, 0, 1 };

    //Continue until there are no more nodes to explore
    while (!pq.empty())
    {
        //Current node being explored
        Node* temp = pq.top();
        pq.pop();

        //Check if the current node is the destination
        if (temp->fx == aDestination->fx && temp->fy == aDestination->fy)
        {
            //Initializing the path, with stringstream datatype
            stringstream path;

            //Tracking back, storing the path from the source to this goal node
            printPath(temp, path);

            //Counting number of steps (by converting "path" to istream)
            istream_iterator is(path.str());
            auto numOfSteps = distance(istream_iterator<is>(), istream_iterator<string>());

            cout << "Number of steps: " << numOfSteps << endl;
            cout << path.str() << endl;
            cout << "A*";
            return;
        }

        // Explore all valid orthogonal neighbors, update their cost, distance, and visited status, and add them to the queue
        for (int i = 0; i < 4; i++)
        {
            int tx = temp->fx + dx[i];
            int ty = temp->fy + dy[i];

            if (isvalid(tx, ty) && !Nodes[tx][ty].fVisited)
            {
                #Nodes[tx][ty].fPrevious = temp;
                #Nodes[tx][ty].fDistance = temp->fDistance + 1;
                #Nodes[tx][ty].fCost = cost(&Nodes[tx][ty], aDestination);
                #Nodes[tx][ty].fVisited = true;
                pq.push(&Nodes[tx][ty]);
            }
        }
    }
}

```


- According to the above code, $fCost$ stands for the heuristic estimated cost between the current node & the destination, while $fDistance$ is the true cost between them.
- The nodes in the priority queue is determined by a specially constructed comparator structure, which adds the heuristic estimated cost ($fCost$) to the true cost ($fDistance$) to reach a node.
- The cost function is applied to the source node, which has a distance of 0 and is marked as visited, to evaluate the heuristic cost to the destination. The source node is then added to the priority queue.
- To ensure the node with the total lowest cost, which is $fCost + fDistance$, is explored first, a priority queue is also be used to manage the nodes in order, similar to GBFS.
- The *isValid* function is also be used for checking the validity of the agent's move. In addition, A* assess the heuristic expense for the goal node, then it backtracks the neighbor's path and adds the neighbor to the queue. At long last the neighbor's distance is processed as the current node in addition to one, for each valid neighbor.
- The objective is the point at which the goal node is taken out from the queue, indicates the shortest way has been found, ends the search. From that point forward, the path is printed out subsequent to being backtracked, utilizing the $fPrevious$ pointers from the goal to the source node.

3.5. Custom Search Strategy 1 (CUS1)

- Definitions: CUS1 algorithm is one of my customize tree-based search algorithms, which utilizes a priority-based search method to investigate the nodes in a graph. The calculation oversees node investigation in view of cost, which is progressively resolved by using node colours, utilizing a priority queue.

- Image 3.5: Custom Search Algorithm 1 (CUS1):

```

#include "Scenario.h"

//Custom comparator for the priority queue to prioritize nodes with lower cost
struct compare
{
    bool operator()(const Node* a, const Node* b) { return a->fCost > b->fCost; }
};

//Cost function determining the cost between two nodes based on their color
int cost(Node* a, Node* b)
{
    return (a->fColor == b->fColor) ? 1 : 2;
}

void Scenario::CUS1(Node* aSource) const
{
    //Priority queue for nodes to explore
    priority_queue<Node*, vector<Node*>, compare> Search;

    //Mark the source node as visited
    aSource->fVisited = true;

    //Set initial cost to 0
    aSource->fCost = 0;

    //Start with the source node
    Search.push(aSource);

    //Continue until there are no nodes left to explore
    while (!Search.empty())
    {
        //Get the node with the lowest cost
        Node* temp = Search.top();

        //Remove it from the priority queue
        Search.pop();

        //Check if the current node is the goal
        if (temp->fColor == 'G')
        {
            //Initializing the path, with stringstream datatype
            stringstream path;

            //Tracking back, storing the path from the source to this goal node
            printPath(temp, path);

            //Counting number of steps (By converting "path" to istream)
            stringstream is(path.str());
            auto numOfSteps = distance(istream_iterator<string>(is), istream_iterator<string>());

            cout << "Number of steps: " << numOfSteps - 4 << endl;
            cout << path.str() << endl;
            cout << "\n";
        }

        for (int i = -1; i <= 1; i++)
        {
            for (int j = -1; j <= 1; j++)
            {
                if (abs(i) + abs(j) != 1) continue;

                //Check if the neighbor is valid (within bounds and not previously visited)
                if (isValid(temp->fX + i, temp->fY + j))
            }
        }
    }
}

```

- In the priority queue, lower costs will equally mean higher priority, which is used to evaluate the nodes. Moreover, essential component is added to pathfinding respectively based on the color of the nodes, by working out the cost between two nodes and leaning toward moves between nodes of a similar color with a lower cost (1 for matching colors, 2 otherwise).
- The source node is set as visited normally, then added to the priority queue. For each node, the path from the source to the objective is made, printed, and the search continues in case the node is the objective ($fColor == 'G'$). This makes it conceivable to distinguish at least one objective.
- The calculation then, at that point, takes a gander at the ongoing node's symmetrical neighbors (up, down, left, right), preventing from slanting moves (symmetry is guaranteed by the way that $abs(i) + abs(j) \neq 1$).
- For each valid neighbor, CUS1 marks the visitation of it. For path tracking, this marks the current node as the neighbor's ancestor ($fPrevious$). Then, the cost to reach this neighbor is evaluated by means of the cost function.
- The manner in which CUS1 decides the cost of moving between nodes while representing the color comparability, which might address different landscapes or zones in a circumstance where moving inside comparable zones is more beneficial or more affordable, makes it unique & special. Verifies that nodes with lower total cost are explored first, coordinating the search toward conceivably more powerful paths to the goal.

3.6. Custom Search Strategy 2 (CUS2)

- Definitions: CUS2 is my second customize tree-based search algorithm. CUS2 algorithm is an exceptionally planned search methodology that incorporates way cost and distance measures to track down the most proficient way from a source node to an objective node.

- Image 3.6: Custom Search Algorithm 2 (CUS2):

```
#include "Scenario.h"

void Scenario::CUS2(Node* aSource, Node* aDestination) const
{
    //Custom comparison function for sorting nodes based on the sum of their cost and distance
    auto comp = [](const Node* a, const Node* b)
    {
        return (a->fCost + a->fDistance) > (b->fCost + b->fDistance);
    };

    //Open list to store nodes that are yet to be explored
    std::vector<Node*> openList;

    //Start with the source node
    openList.push_back(aSource);

    //Continue until there are no nodes left to explore
    while (!openList.empty())
    {
        //Index of the node with the minimum cost + distance
        int Minimum = 0;
        for (int i = 1; i < openList.size(); i++)
        {
            if ((openList[i]->fCost + openList[i]->fDistance) < (openList[Minimum]->fCost + openList[Minimum]->fDistance))
            {
                Minimum = i;
            }
        }

        //Node with the minimum cost + distance
        Node* temp = openList[Minimum];

        //Remove it from the open list
        openList.erase(openList.begin() + Minimum);

        //Mark the node as visited
        temp->fVisited = true;

        //Check if the current node is the destination node
        if (temp->fX == aDestination->fX && temp->fY == aDestination->fY)
        {
            //Initializing the path, with stringstream datatype
            stringstream path;

            //Tracking back, storing the path from the source to this goal node
            printPath(temp, path);

            //Counting number of steps (By converting "path" to istream)
            stringstream is(path.str());
            auto numOfSteps = distance(istream_iterator<string>(is), istream_iterator<string>());

            cout << "Number of steps: " << numOfSteps - 1 << endl;
            cout << path.str() << endl;
            cout << "\n";
            break;
        }

        int tx = temp->fX;
        int ty = temp->fY;

        //Lambda function to attempt pushing valid neighboring nodes onto the open list
        auto Push = [&](int dx, int dy)
        {
            if (isValid(tx + dx, ty + dy))
            {
                Node* n = new Node(tx + dx, ty + dy);
                n->fPrevious = temp;
                n->fCost = temp->fCost + 1;
                n->fDistance = distance(n, aDestination);
                openList.push_back(n);
            }
        };
    }
}
```

- Focuses on nodes as per the absolute of their crossing cost (*fCost*) and heuristic distance (*fDistance*) to the goal, by involving a lambda function for nodes correlation in the open rundown. This heuristic strategy really coordinates the inquiry towards the goal.
- To identify nodes that require further investigation, I utilize an open list (which is implemented as a *std::vector*). This list is dynamically updated with nodes that are investigated and discovered.
- Iteratively, the node with the lowest combined cost and distance is selected from the open list to be investigated. Using the specified comparator criteria, the open list is linearly searched to find the most promising node.
- Valid, undiscovered neighbors are processed after calculating their *fCost* for the destination, modifying the *fDistance* from the starting point, setting respective *fPrevious* pointer for path recording, and storing them to the open list for additional investigation.
- The technique builds the path from the final point back to the source using the *fPrevious* pointers after discovering the destination node. It then emits this route, showing the journey taken, using a *stringstream*.

3.7. Advantages & Disadvantages of Algorithms

Algorithms	Advantages	Disadvantages
DFS	<ul style="list-style-type: none">- Simple to be used with recursion- Requires less memory than BFS- Most suitable for solving puzzles, which requires only finding a way to the goal state.	<ul style="list-style-type: none">- Not always the shortest path- Without repeated state check, the agent might be stuck in infinity loops.
BFS	<ul style="list-style-type: none">- Seeks for shortest path quicker than DFS- Can prevent unnecessary paths	<ul style="list-style-type: none">- Requires more memory than DFS- If the solution requires deep explore in the tree, the performance might be slower
GBFS	GBFS can be more efficient than DFS or BFS, by means of utilizing the heuristics towards the goal node.	Not always the quickest path, since the heuristic quality can totally affect the performance of the algorithm.
A*	<ul style="list-style-type: none">- Guarantees the shortest path displayed- Combining both the completeness of BFS & heuristic cost of GBFS	<ul style="list-style-type: none">- Requires efficient heuristic- Insufficient heuristics might negatively affect the performance- Might require much memory
CUS1 & CUS2	It custom fitted to specific issue limitations or goals, perhaps giving more powerful solutions in certain conditions.	<ul style="list-style-type: none">- Requires more fine-tuning or testing to optimize the output.- Custom solutions probably won't transfer well to different sorts of issues or circumstances.

4. Testing

To finish the program's Testing Procedures, it requires a testing methodology. As a result, I believe **Metamorphic Testing** is the best option that I can use to approach the testing phase.

By specifying relationships between inputs and outputs, **Metamorphic Testing** is a software testing approach that helps create and generate related test cases and solve the test oracle problem. A test oracle is defined as a mechanism or technique that compares the program's outputs to its test cases to determine the program's correctness. **Metamorphic Relations** are equally essential notions needed to implement **Metamorphic Testing**.

For verifying the program implementation under these conditions, which include different linked inputs and their outputs, **Metamorphic Relations (MR)** are essential qualities. New test cases can be produced by creating related inputs based on the identification of metamorphic relations.

The following actions will enable us to begin building Metamorphic Testing:

- Carrying out several arbitrary test scenarios.
- Determine the characteristics of the issue, which will lead to the identification of MR.
- Using the discovered MR to generate new test cases.
- Confirming the metamorphic relations using the calculated results.

Test case 1: Displaying path from customized start node, goal nodes

MR: Different coordinate of start node might lead to different and respective paths. Input 1.2 is generated from input 1.1 by adjusting the coordinates of the start node, where the agent begins.

Input 1.1: Original inputs from the assignment

```
RobotNav-test.bat  x  BotWinPasswords.bat
File Edit View
[5,11]
(9,1)
(7,0) | (10,3)
(2,0,2,2)
(8,0,1,2)
(10,0,1,1)
(2,3,1,2)
(3,4,3,1)
(9,3,1,1)
(8,4,2,1)
```

Output 1.1: Display paths of each algorithm from Node(0,1) to Node(7,0) & Node(10,3)

```
Search First Search
Number of steps: 28
1:up,'right','down','down','right','right','down','right','up','up','up','right','down','down','down','right','up','up','right'
Goal: Node (7, 0)

Number of steps: 28
1:up,'right','down','down','right','right','down','right','up','up','up','right','down','down','down','right','up','up','right','down','down'
Goal: Node (10, 3)

Breadth First Search
Number of steps: 28
1:down,'right','right','right','up','up','right','right','right'
Goal: Node (7, 0)

Number of steps: 22
1:down,'right','right','right','right','right','right','right','right','down'
Goal: Node (10, 3)

Greedy Best First Search
Number of steps: 28
1:right,'down','right','right','up','up','right','right','right'
Goal: Node (7, 0)

Number of steps: 26
1:down,'right','right','right','down','right','right','right','right','up','up','right','right','down'
Goal: Node (10, 3)

A* Search
Number of steps: 28
1:right,'down','right','right','up','up','right','right','right'
Goal: Node (7, 0)

Number of steps: 22
1:down,'right','right','right','right','right','right','right','right','down'
Goal: Node (10, 3)

Dijkstra Search
Number of steps: 28
1:right,'down','right','right','right','up','up','right','right','up'
Goal: Node (7, 0)

Number of steps: 22
1:right,'down','right','right','right','right','right','right','right','right','down'
Goal: Node (10, 3)

D* Search
Number of steps: 28
1:down,'right','right','right','right','right','right','up','up'
Goal: Node (7, 0)

Number of steps: 22
1:right,'down','right','right','right','right','right','right','right','right','down'
Goal: Node (10, 3)
```

Input 1.2: New coordinates of the agent/the start node

```
RobotNav-test.bat  x  BotWinPasswords.bat
File Edit View
[5,11]
(0,3)
(7,0) | (10,3)
(2,0,2,2)
(8,0,1,2)
(10,0,1,1)
(2,3,1,2)
(3,4,3,1)
(9,3,1,1)
(8,4,2,1)
```

Output 1.2: New paths are display respectively, from Node(0,3) to Node(7,0) & Node(10,3)

```
Search First Search
Number of steps: 22
1:up,'up','right','down','down','right','right','down','right','up','up','up','right','down','down','down','right','up','up','right'
Goal: Node (7, 0)

Number of steps: 26
1:up,'up','right','down','down','right','right','down','right','up','up','up','right','down','down','down','right','up','up','right','down','down'
Goal: Node (10, 3)

Breadth First Search
Number of steps: 28
1:up,'right','right','right','up','up','right','right','right'
Goal: Node (7, 0)

Number of steps: 22
1:up,'right','right','right','right','right','right','right','right','down'
Goal: Node (10, 3)

Greedy Best First Search
Number of steps: 28
1:up,'right','right','right','right','up','up','right','right','right'
Goal: Node (7, 0)

Number of steps: 26
1:right,'down','right','right','down','right','right','right','right','up','up','right','right','down'
Goal: Node (10, 3)

A* Search
Number of steps: 28
1:up,'right','right','right','up','up','right','right','right'
Goal: Node (7, 0)

Number of steps: 22
1:right,'down','right','right','right','right','right','right','right','down'
Goal: Node (10, 3)

Dijkstra Search
Number of steps: 28
1:up,'right','right','right','right','up','up','right','right','up'
Goal: Node (7, 0)

Number of steps: 22
1:right,'down','right','right','right','right','right','right','right','right','down'
Goal: Node (10, 3)

D* Search
Number of steps: 28
1:up,'right','right','right','right','right','right','up','up'
Goal: Node (7, 0)

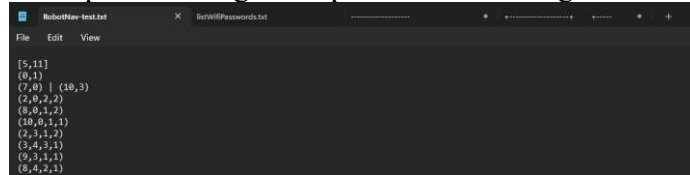
Number of steps: 22
1:right,'down','right','right','right','right','right','right','right','right','down'
Goal: Node (10, 3)
```

Conclusion: After adjusting the coordinates of the agent, the path changes accurately. So, the test case passes the requirements.

Test case 2: Checking whether goal node is reachable or not

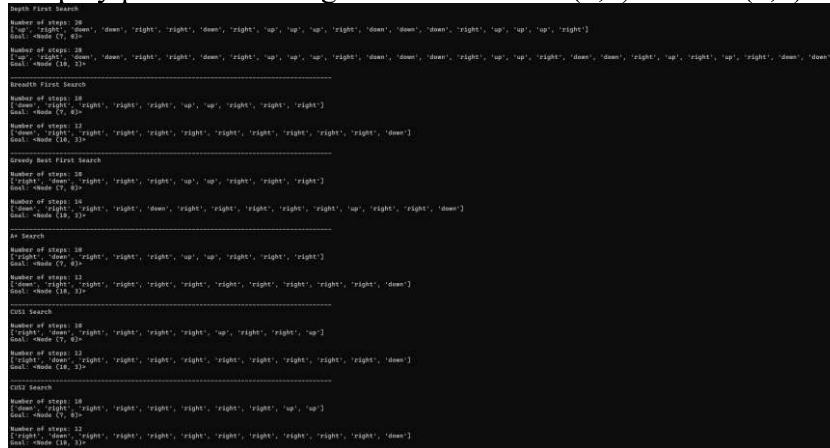
MR: Different coordinate of goal nodes might lead to less paths displayed, since several nodes with inappropriate coordinates cannot be reached.

Input 2.1: Original inputs from the assignment



```
RobotNav-test.txt
[5,11]
(0,1)
(7,0) | (10,3)
(2,0,2,2)
(8,0,1,2)
(10,0,1,1)
(2,3,1,2)
(3,4,3,1)
(9,3,1,1)
(8,4,2,1)
```

Output 2.1: Display paths of each algorithm from Node(0,1) to Node(7,0) & Node(10,3)



```
RobotNav-test.txt
Number of steps: 30
[up, right, down, down, right, right, down, right, up, up, right, down, down, down, right, up, up, right]
Goal: Node (7, 0)

Number of steps: 28
[up, right, down, down, right, right, down, right, up, up, right, down, down, down, right, up, up, right, down, down]
Goal: Node (10, 3)

Breadth First Search
Number of steps: 18
[up, right, right, right, up, up, right, right, right]
Goal: Node (7, 0)

Number of steps: 12
[down, right, right, right, right, right, right, right, right, down]
Goal: Node (10, 3)

Greedy Best First Search
Number of steps: 10
[up, down, right, right, right, up, up, right, right, right]
Goal: Node (7, 0)

Number of steps: 16
[down, right, right, down, right, right, right, right, right, up, right, right, down]
Goal: Node (10, 3)

A* Search
Number of steps: 18
[up, right, up, right, right, up, up, right, right, right]
Goal: Node (7, 0)

Number of steps: 12
[down, right, right, right, right, right, right, right, right, down]
Goal: Node (10, 3)

CDS Search
Number of steps: 18
[up, right, down, right, right, right, right, right, right, right, right, right, right, right, right]
Goal: Node (7, 0)

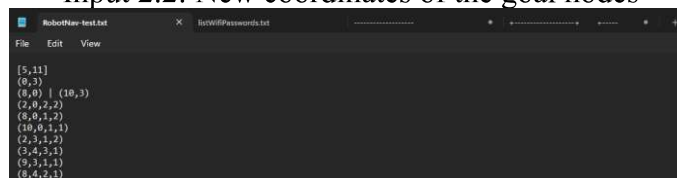
Number of steps: 12
[up, right, down, right, right, right, right, right, right, right, right, right, down]
Goal: Node (10, 3)

CDS Search
Number of steps: 18
[up, right, right, right, right, right, right, right, right, right, up, up]
Goal: Node (7, 0)

Number of steps: 12
[up, right, down, right, right, right, right, right, right, right, right, right, down]
Goal: Node (10, 3)

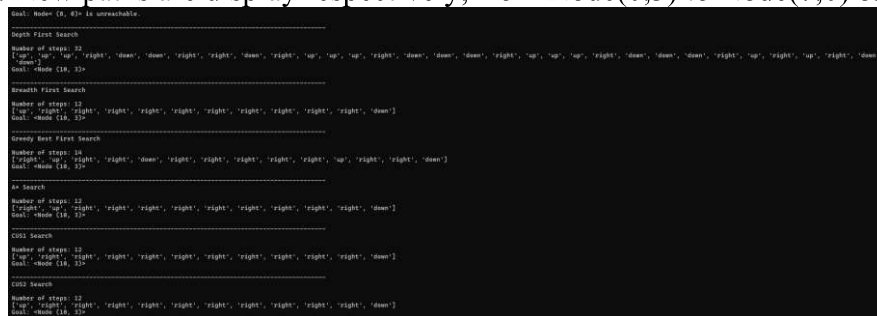
CDS Search
Number of steps: 12
[up, right, right, right, right, right, right, right, right, right, right, right]
Goal: Node (10, 3)
```

Input 2.2: New coordinates of the goal nodes



```
RobotNav-test.txt
[5,11]
(0,3)
(8,0) | (10,3)
(2,0,2,2)
(8,0,1,2)
(10,0,1,1)
(2,3,1,2)
(3,4,3,1)
(9,3,1,1)
(8,4,2,1)
```

Output 2.2: New paths are display respectively, from Node(0,3) to Node(7,0) & Node(10,3)



```
RobotNav-test.txt
Goal: Node (0, 0) is unreachable

Depth First Search
Number of steps: 22
[up, up, up, right, down, down, right, right, down, right, up, up, right, down, down, down, right, up, up, right, down, down, down, right, up, right, down]
Goal: Node (10, 3)

Breadth First Search
Number of steps: 12
[up, right, right, right, right, right, right, right, right, right, down]
Goal: Node (10, 3)

Greedy Best First Search
Number of steps: 14
[up, right, up, right, down, right, right, right, right, up, right, right, down]
Goal: Node (10, 3)

A* Search
Number of steps: 12
[up, right, up, right, right, right, right, right, right, right, right, down]
Goal: Node (10, 3)

CDS Search
Number of steps: 12
[up, right, right, right, right, right, right, right, right, right, right, down]
Goal: Node (10, 3)

CDS Search
Number of steps: 12
[up, right, right, right, right, right, right, right, right, right, right, down]
Goal: Node (10, 3)
```

Conclusion: After adjusting the coordinates of one goal node, that unreachable goal node was notified to the user. The remaining paths to the remaining goal node are still displayed ordinarily.

5. Features

- Depth-First Search (DFS): Extremely skilled at finding new paths or locations inside a grid or graph. use a backtracking technique to find new paths and trace back its movements.
- Breath-First Search (BFS): Explore all the nodes from the source at the same level first, ensuring that the nearest nodes are extensively searched first, using a queue to monitor the next nodes to probe in a First In, First Out (FIFO) fashion.
- Greedy Best-First Search (GBFS): Determines the cost or distance to the destination and applies this heuristic to prioritize nodes, hence focusing the search. tries to choose the most direct route to find a solution as soon as possible, often requiring less investigation.
- A Star (A*): Assures the quickest route to the destination by combining heuristic estimates with real path expenses. successfully finds a middle ground between pursuing intriguing directions and getting closer to the goal.
- Custom Search Strategy 1 (CUS1): By altering the cost function, this algorithm tailors the search to specific issue constraints. focuses on cost-effective routes while supervising the node discovery process using a priority queue.
- Custom Search Strategy 2 (CUS2): This algorithm decides which node will be inspected through a unique interaction, considering total information, by utilizing an open list to accommodate node insertion and elimination throughout the searching processes.

6. Conclusion

Which search calculation is best depends on the issue's aims and size: A*, or alternatively BFS, is used for small areas that need meticulous planning; GBFS, or customized approaches, are used for large, effectiveness-focused areas; and DFS is used for simpler, memory-cognitive tasks. The selection characteristics emphasize how important it is to understand the problem, test out several computations, and choose which one best satisfies the objectives by comparing memory requirements, optimality, and proficiency.

7. References

- A fast algorithm for finding better routes by AI search techniques:
<https://ieeexplore.ieee.org/abstract/document/396824/>
- The PN*- Search Algorithm:
<https://www.sciencedirect.com/science/article/pii/S0004370201000844>
- The variants of the harmony search algorithm:
<https://link.springer.com/article/10.1007/s10462-010-9201-y>