Unit Name: Introduction to AI

Unit Code: COS30019

Title: Assignment 1 (Tree-Based Search)

Name: S M RAGIB REZWAN

ID: 103172423

Table of Contents

Instructions:

The Instructions of how to use the Search Algorithm Program discussed about here (i.e. the attached codes) can be separated into the following stages:

1. **Starting stage of the Program:**

Before running the program, move the search.exe file and the desired map file to a separate folder. This would ensure that the map file would be detected by the algorithm:

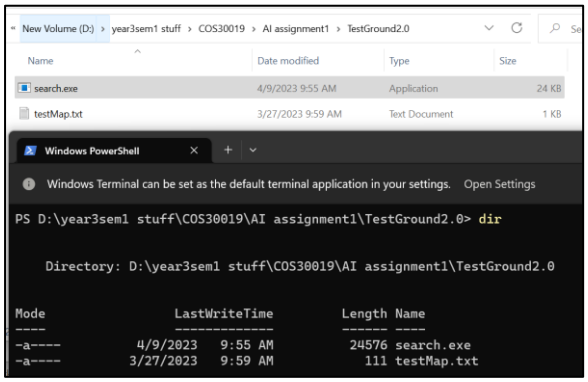


Fig-1: Keeping Search Program and map file in same folder

Then run the program using the following syntax in command line:

"***.\search.exe*** <***mapFileName***> <***algoName***>"

```
PS D:\year3sem1 stuff\COS30019\AI assignment1\TestGround2.0> .\search.exe testMap bfs
```

Fig-2: To run the program

Here, make sure to only write the name of the file containing map in '**mapFileName'** as its extension will be automatically added by the program. Furthermore, the algorithm will be called in the '**algoName'** (irrelevant of casing) where:

1. **bfs** => Breadth First Search
2. **dfs** => Depth First Search
3. **dfslim** => Limited Depth First Search
4. **gbfs** => Greedy Best First Search
5. **astar** => A Star Search
6. **astarlim** => Limited A Star Search

2. **Running stage of the program:**

After this, the program will load in the map and start running. Here it will automatically load the GUI form in the command line to allow the user to see how the search is running. It will also let the user know which nodes are currently in its frontier and which nodes it's currently expanding in a text below the picture. Each time it would wait 0.1 second before loading the next node to allow the user to appreciate the way the search algorithm is running.

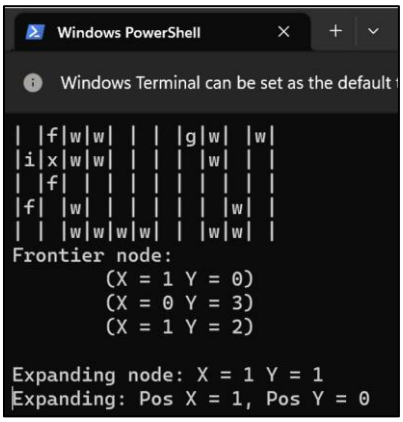


Fig-3: Search Program running

Note: Here, the '**i'** refers to initial state, '**g'** refers to goal state, '**x'** refers to the currently visiting node it is in, and 'f' refers to the nodes in its frontier. It should be kept in mind that once the nodes passes from frontier to visiting node and also when its removed from visiting node, it will no longer be seen in the diagram as those areas have already been explored by agent

Note: In this program, the multi-goal function has been coded in such a way so that the program will use it to create multiple instances of initial to goal state paths. Thus in order to see the path to the second or more goal, press the "Enter".

```
BFS 33 down; right; right; right; right; up; up; right; right; right;
This is for Goal coordinates: X = 7 and Y = 0
Press Enter to see for alternative goal Coordinates (if any)
```

Fig-4: Search Program complete for 1 of the goals

3. **Ending stage of the Program:**

Once the program has completed for all goals case, the user must press "Enter" twice in order to exit. Here the output is in the following syntax:

"<**algoName**> <**No of search Nodes**> <**Action taken to goal**>"

Alongside this, it also has a line stating which goal coordinates it had found the solution for

(alongside instruction to press “Enter” in case any other alternative goal exists on map)

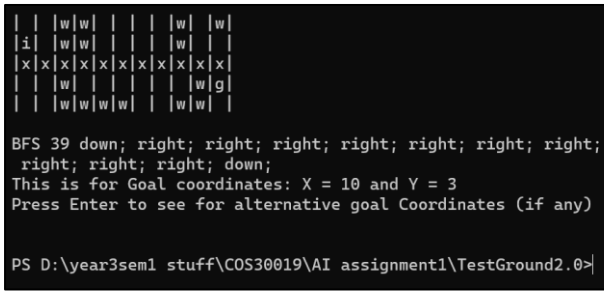


Fig-5: Final Output of the Program

Introduction:

This program had been developed in order to solve the “Robot Navigation Problem”**[1]** which is basically the question considering how to teach the robot to find the path it can take to reach its goal from its current location, whilst overcoming obstacles (if any).This is an extremely important problem in both the area of Robotics **[2][3]** (as it can allow robot to navigate environments hazardous to humans, transport goods and resources more efficiently accurately, etc.) and AI **[4]**,(as it can act rationally **[5]** and thus perceive its environment, including any change occurring in it, and take actions that maximizes the chance of finding the optimal path) leading to various numbers of research being performed on the topic.

Although all these research provide various search algorithms as the solutions, it is difficult to understand how they are working without first understanding how the navigation robot is setup.

Navigation robots are built using the design technique of rational agent (from the paradigm of Ai: Systems that act rationally **[5]** where we set up a PEAS description of the environment **[6]** (i.e. the Performance measure its using, Environmental knowledge it knows, Actuators that it uses to perform action, and sensors that it used to detect its environment and any change). This is then used to create state spaces to describe its environment (which includes agent (i.e. the robot’s) position, goal position, boundaries of environment, empty rooms, walls, etc.) and the actions the agent (i.e. the robot) can take to move from one state to the next in a simplistic manner.

Once state space has been created, information regarding them is used to make nodes. These are basically a form of data structure which contains other necessary information needed for the agent to navigate (i.e. parent node leading to current node, action taken to reach this node, past cost to the node, depth of the node, etc.). These nodes in turn are used to make a tree diagram. In there, the nodes are organized and connected in such a way that the initial position is stored in the root node, goal position is stored in any of its child node (on any of its branches) and the nodes are connected to one another using branches.

Now that the tree has been established, the robot understands its environment, the outcomes its actions have (i.e. change in its position) and its goal (reaching a certain position from its starting position). Furthermore, by going through or searching the tree, it can find out the way it can reach its goal!

But creating a tree and going through it takes time and memory. Also, if it’s not properly done, some nodes can be missed or even looked through twice. Furthermore, some of the nodes can even have the same state and the tree can even have loops where a child node and its ancestor node (i.e. its parent’s parent node) are basically the same.

Thus, to avoid these issues, systematic approaches or search algorithms have been developed which inserts each child node of root node into frontiers, checks whether they have been visited or have any repeated state, and then check whether it has the goal state in its node. If it has, then the algorithm stops and states the goal and path to it. Otherwise it finds and adds the child nodes of the child nodes and repeats until goal is obtained.

These search algorithms can be categorized into two types:

1. Blind or uninformed Search Strategy: Strategy that doesn’t exploit any of the information contained in a state **[7]** like Depth first Search, Breadth First Search, etc.

2. Heuristic of informed Search Strategy: Strategy that exploits that information to

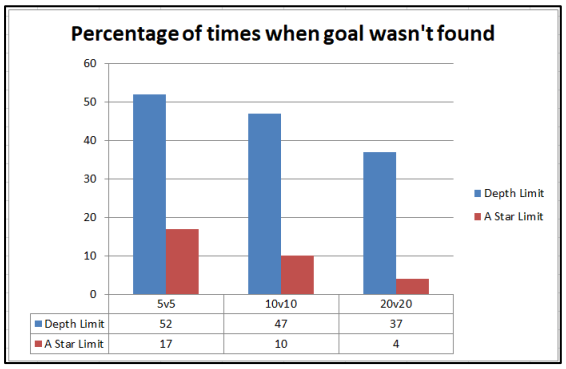
assess which nodes are more promising and check them out first **[7]** like Greedy Best First Search, A Star Search, etc.

But which algorithm is the best at solving the Robot Navigation Problem? That's what this investigation is attempting to find out.

Search Algorithms:

In this investigation, 3 blind and 3 Heuristic searches have been used. The blind searches include: "Depth First Search (DFS)", Breadth First Search (BFS)" and "Custom Blind Search (Depth Limited Search (DFSLimited))", whilst the Heuristic ones include: "Greedy Best First Search (GBFS)", "A Star Search (AStar)" and "Custom Heuristic Search (A Star Limited Search (AStarLimited))".

Here, DFS, BFS, GBFS and AStar algorithms have been used as the regular search algorithms due to assignment criteria whilst Depth limited aspect of DFS and A Star had been used as the custom search algorithm to make them more memory efficient **[8]** (and thus reach the goal faster) at the expense of not always finding the goal.



Graph-1: Percentage of times when Goal wasn't found for Depth Limit (DFSLimited) and A Star Limit (AStarLimited) for 5v5, 10v10 and 20v20 map sizes.

Note: 5v5 means map whose map width and length is 5.

Note: For the Custom Algorithms, the depth limit had been set to half of the total possible rooms (irrelevant of wall and clear states) for the given map using the following line:

```
//setting depth limit as (map's total no of rooms (wall or not)/2)
_maxDepth = (_agentMap.Length * _agentMap.Width) / 2;
```

Fig-6: Depth Limit set on the custom algorithms

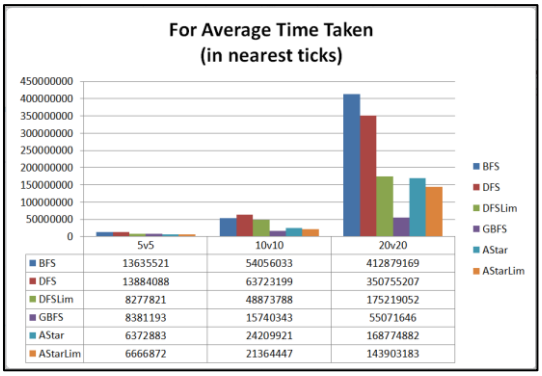
Furthermore, the category of "Best" had been too vague to be a proper metric. Hence it had been subdivided into the following performance metrics to make more accurate comparison:

Note: For the two custom algorithms, it should be kept in mind that the data collected for them has only been for the cases where solution had been obtained and thus it would be better remember information from **Graph-1** whilst viewing these Charts.

Note: Each data provided here is the average value that has been found for 100 such cases

1. **Average Time taken to run the algorithm (in ticks):**

   Here average time taken to run the algorithms has been noted. This had been noted in numbers of ticks in order to ensure accuracy and also cope with the fast rates the small maps were processed by the algorithms



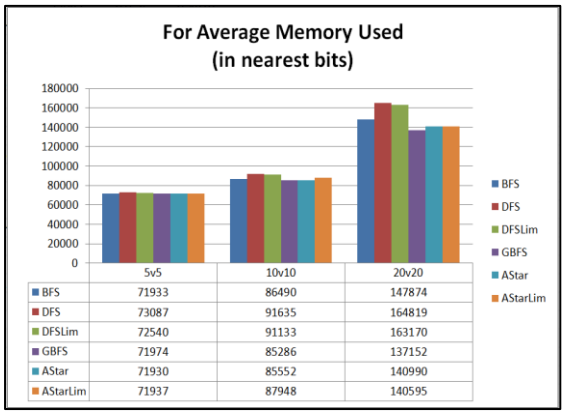
Graph-2: Average time taken by the algorithms

Note: 10,000 ticks make 1 millisecond

The data shows that A Star Limited had been fastest for 5v5 size map, whilst GBFS has been fastest for 10v10 and 20v20 size map. Also, it shows that DFS was the slowest for 5v5 and 10v10 size maps whilst BFS was the slowest for the 20v20 size maps. Furthermore, it can be seen that the time taken fluctuates a lot between algorithms for the same map size and that it increases by an enormous amount when the map size increases.

2. Average Memory Usage (in bits)

Here average memory usage for running the algorithm has been noted. This had been done in bits to ensure accuracy.



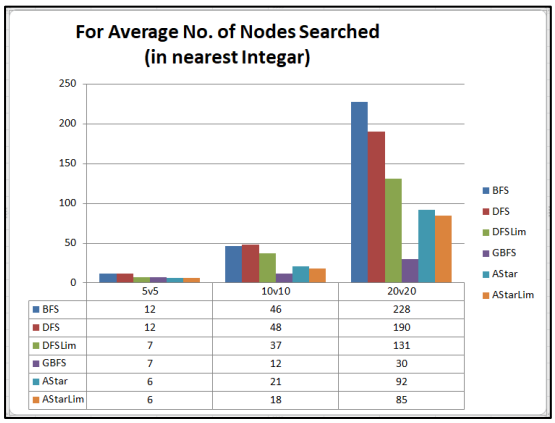
	5v5	10v10	20v20
BFS	71933	86490	147874
DFS	73087	91635	164819
DFSLim	72540	91133	163170
GBFS	71974	85286	137152
AStar	71930	85552	140990
AStarLim	71937	87948	140595

Graph-3: Average memory used by the algorithms

Here we can see that that A Star used the least amount of memory for 5v5 map size whilst GBFS used the least amount of memory for 10v10 and 20v20 map sizes. Also, DFS used the most amount of memory for all map sizes. Furthermore, the memory usage by all the algorithms was quite similar for the same map size and increased by little when map size had been increased.

3. Average No. of Nodes searched

Here the average number of nodes searched by algorithm has been noted.



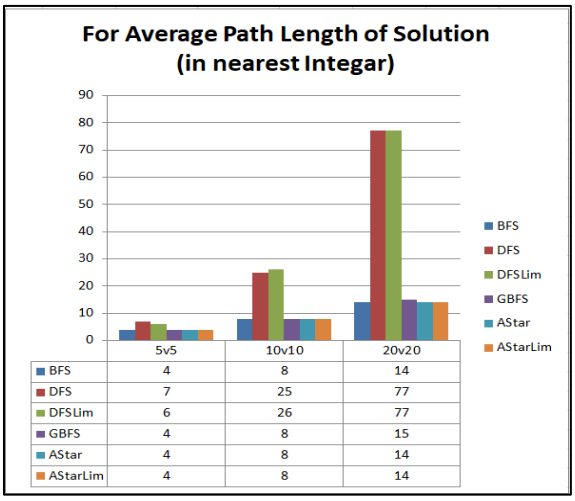
	5v5	10v10	20v20
BFS	12	46	228
DFS	12	48	190
DFSLim	7	37	131
GBFS	7	12	30
AStar	6	21	92
AStarLim	6	18	85

Graph-4: Average no. of nodes searched by the algorithms

The data shows that A Star Limited had searched the least amount of nodes for 5v5 size map, whilst GBFS had searched the least amount for 10v10 and 20v20 size map. Also, it shows that BFS and DFS searched the most amounts of nodes for 5v5, DFS searched the most amounts for the 10v10 and BFS searched the most amounts for 20v20 size maps. Furthermore, it can be seen that the number of nodes searched fluctuates a lot for the same map size and increases by an enormous amount when the map size increases.

4. Average Path Length of Solution:

Here the average path length of the solution (i.e. path from initial to goal position) found by the algorithm has been noted.



	5v5	10v10	20v20
BFS	4	8	14
DFS	7	25	77
DFSLim	6	26	77
GBFS	4	8	15
AStar	4	8	14
AStarLim	4	8	14

Graph-5: Average path length of solution found by the algorithms

The data shows that BFS, GBFS, A Star and A Star Limited had the least path length of the solution for 5v5 map size and 10v10 map size, whilst BFS, A Star and A Star Limited has the least path length to solution for 20v20 map size. Also, the path length was most for DFS in 5v5 map size, DFS Limited for 10v10 map size and both DFS Limited and DFS for 20v20 map size. Furthermore, other than DFS and DFS Limited, the rest of the algorithms had almost similar path length for the same map size and it increased by a small amount as the map size increased.

Overall, by considering the data obtained for these 4 factors, the best algorithm for 5v5 size maps is a tie between A Star and A Star Limited and for 10v10 and 20v20 size maps, it is GBFS.

Implementation:

Here, brief explanation regarding how the program had been designed and created has been noted using UML diagram and pseudo codes.

Overall the entire program can be seen at a glance using the class diagram shown below:

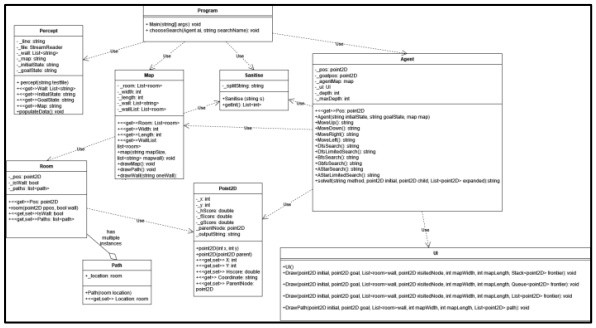


Fig-6: UML Class Diagram of the full program

Basically the program works in the following manner:

At first, the **Program** class takes in the inputs and separates them into file name and search algorithm names. Then it uses **Percept** class to read through and separate the lines in the file into those related to map size, initial position, goal position(s), and wall(s). Then it uses the **Map** class to set up the map and assign the rooms, walls and paths. During this time, the **Map** class also uses the **Sanitise** class to extract the necessary integers from the passed lines.

Once this is done, the **Program** class uses **Sanitise** class to see the goal positions and determine whether its multiple goals or single goal. After this has been decided, it uses the **Agent** class to create itself using the initial state, goal state and the **Map**.

This causes the **Agent** class to use **Sanitise** class to extract the necessary initial and goal positions, loads up the passed **Map** and creates new **UI.**

After that, the **Program** class chooses which search algorithm to use and asks the **Agent** class to run it. This in turn makes the **Agent** class run the search algorithm **(See Fig-7 to 13 for the necessary pseudo codes)** and also call on the **UI** class to draw all the nodes it goes through in its search to goal.

This results in two outcomes:

1. Goal is not found and thus no solution found (or no solution at given depth of x [where x is half of the total no of rooms on the map regardless of empty, wall, start or goal]) is stated as outcome.

2. Goal has been found and thus the solution path (from initial position to goal position) is recreated and sent to **UI** to be drawn

Furthermore, the logic used to create the search algorithms have been briefly explained in the figures below via pseudo codes **[9]**:

```
DfsSearch()
IF (intial position's X and Y = goal position's X and Y) THEN
Return that the solution is here and no need to move
ELSE
make frontier as a Stack
make visited as a List
make visitedNode as a point2D
Push current position node to frontier
    WHILE (anything is still in frontier)
    visitedNode <- POP the node from frontier
    ADD visitedNode to visited
    DRAW it using ui
        FOREACH(room r in map.room)
            IF(Visited node's X and Y = r's position's X and Y ) THEN
                IF(r has a path) THEN
                    FOREACH(path p in r.path)
                        IF(X and Y position noted in any of the nodes, in either
                        visited or frontier, doesn't matches with that stated on
                        the adjacent room attached on the other end of the p)
                        THEN
                        ParentNode for the room located on the other side of the p
                        <- visitedNode
                        PUSH the room located on the other side of path into the
                        frontier
                        ENDIF
                    ENDFOR
                ENDIF
                IF(visited node's X and Y = goal node's X and Y) THEN
                Return that solveIT method needs to be run with the necessary
                parameters
                ENDIF
            ENDIF
        ENDFOR
    ENDWHILE
    Return that No solution
ENDIF
```

Fig-7: Pseudo code for DFS

Note: Here STACK has been used instead of a normal list as STACK stores elements in LIFO (Last in First Out manner **[10]**)

Note: The extremely nested "if" condition is basically doing repeated state checking

```
DfsLimitedSearch()
maxdepth <- map's length * height /2
depth <- max depth
IF (intial position's X and Y = goal position's X and Y and depth greater than 0)
THEN
Return that the solution is here and no need to move
ELSE
make frontier as a Stack
make visited as a List
make visitedNode as a point2D
Push current position node to frontier
    WHILE (anything is still in frontier and depth is greater than 0)
    visitedNode <- POP the node from frontier
    ADD visitedNode to visited
    DRAW it using ui
    depth <- depth - 1
        FOREACH(room r in map.room)
            IF(Visited node's X and Y = r's position's X and Y ) THEN
                IF(r has a path) THEN
                    FOREACH(path p in r.path)
                        IF(X and Y position noted in any of the nodes, in either
                        visited or frontier, doesn't matches with that stated on
                        the adjacent room attached on the other end of the p)
                        THEN
                        ParentNode for the room located on the other side of the p
                        <- visitedNode
                        PUSH the room located on the other side of path into the
                        frontier
                        ENDIF
                    ENDFOR
                ENDIF
                IF(visited node's X and Y = goal node's X and Y and depth greater
                than 0) THEN
                Return that solveIT method needs to be run with the necessary
                parameters
                ENDIF
            ENDIF
        ENDFOR
    ENDWHILE
    Return that No solution at given max depth
ENDIF
```

Fig-8: Pseudo code for DFSLimited

```
BfsSearch()
IF (intial position's X and Y = goal position's X and Y) THEN
Return that the solution is here and no need to move
ELSE
make frontier as a Queue
make visited as a List
make visitedNode as a point2D
ENQUEUE current position node to frontier
    WHILE (anything is still in frontier)
    visitedNode <- DEQUEUE the node from frontier
    ADD visitedNode to visited
    DRAW it using ui
        FOREACH(room r in map.room)
            IF(Visited node's X and Y = r's position's X and Y ) THEN
                IF(r has a path) THEN
                    FOREACH(path p in r.path)
                        IF(X and Y position noted in any of the nodes, in either
                        visited or frontier, doesn't matches with that stated on
                        the adjacent room attached on the other end of the p)
                        THEN
                        ParentNode for the room located on the other side of the p
                        <- visitedNode
                        ENQUEUE the room located on the other side of path into the
                        frontier
                        ENDIF
                    ENDFOR
                ENDIF
                IF(visited node's X and Y = goal node's X and Y) THEN
                Return that solveIT method needs to be run with the necessary
                parameters
                ENDIF
            ENDIF
        ENDFOR
    ENDWHILE
    Return that No solution
ENDIF
```

Fig-9: Pseudo code for BFS

Note: Here QUEUE has been used instead of a normal list as QUEUE stores elements in FIFO (First in First Out manner **[11]**)

```
GbfsSearch()
IF (intial position's X and Y = goal position's X and Y) THEN
Return that the solution is here and no need to move
ELSE
make frontier as a List
make visited as a List
make visitedNode as a point2D
ADD current position node to frontier
    WHILE (anything is still in frontier)
    frontier <- ORDER elements in frontier BY Hscore and convert output TO LIST
    visitedNode <- FIRST element in frontier
    REMOVE the FIRST element from frontier
    ADD visitedNode to visited
    DRAW it using ui
        FOREACH(room r in map.room)
            IF(Visited node's X and Y = r's position's X and Y ) THEN
                IF(r has a path) THEN
                    FOREACH(path p in r.path)
                        IF(X and Y position noted in any of the nodes, in either
                        visited or frontier, doesn't matches with that stated on the
                        adjacent room attached on the other end of the p)
                        THEN
                        ParentNode for the room located on the other side of the p <-
                        visitedNode
                        Hscore for the room located on the other side of the p <-
                        squareroot of(sum of (square of difference between X value of
                        room located on the other side of the p and goal) and (square
                        of difference between Y value of room located on the other
                        side of the p and goal))
                        ADD the room located on the other side of path into the
                        frontier
                        ENDIF
                    ENDFOR
                ENDIF
                IF(visited node's X and Y = goal node's X and Y) THEN
                Return that solveIT method needs to be run with the necessary
                parameters
                ENDIF
            ENDIF
        ENDFOR
    ENDWHILE
    Return that No solution
ENDIF
```

Fig-10: Pseudo code for GBFS

Note: For all the heuristic algorithms, the Hscore is basically the straight line distance from the "next to enter" node's position to goal's position

```
AStarSearch()
IF (intial position's X and Y = goal position's X and Y) THEN
Return that the solution is here and no need to move
ELSE
make frontier as a List
make visited as a List
make visitedNode as a point2D
ADD current position node to frontier
current position Gscore <- 0
    WHILE (anything is still in frontier)
    frontier <- ORDER elements in frontier BY Fscore and convert output TO LIST
    visitedNode <- FIRST element in frontier
    REMOVE the FIRST element from frontier
    ADD visitedNode to visited
    DRAW it using ui
        FOREACH(room r in map.room)
            IF(Visited node's X and Y = r's position's X and Y ) THEN
                IF(r has a path) THEN
                    FOREACH(path p in r.path)
                        IF(X and Y position noted in any of the nodes, in either visited or
                        frontier, doesn't matches with that stated on the adjacent room attached on
                        the other end of the p)  THEN
                        ParentNode for the room located on the other side of the p <- visitedNode
                        Gscore for the node for the room located on the other side of p <-
                        visitedNode's Gscore + 1
                        Fscore for the room located on the other side of the p <- squareroot of(sum
                        of (square of difference between X value of room located on the other side
                        of the p and goal) and (square of difference between Y value of room
                        located on the other side of the p and goal))
                        ADD the room located on the other side of path into the
                        frontier
                        ENDIF
                    ENDFOR
                ENDIF
                IF(visited node's X and Y = goal node's X and Y) THEN
                Return that solveIT method needs to be run with the necessary
                parameters
                ENDIF
            ENDIF
        ENDFOR
    ENDWHILE
    Return that No solution
ENDIF
```

Fig-11: Pseudo code for A Star

Note: Here, the Gscore is basically the cost taken to reach current node and Fscore is basically the summation of Gscore and the straight line distance from the "next to enter" node's position to goal's position

```
AStarLimitedSearch()
maxdepth <- map's length * height /2
depth <- max depth
IF (intial position's X and Y = goal position's X and Y and depth greater than 0) THEN
Return that the solution is here and no need to move
ELSE
make frontier as a List
make visited as a list
make visitedNode as a point2D
ADD current position node to frontier
current position Gscore <- 0
    WHILE (anything is still in frontier and depth greater than 0)
    frontier <- ORDER elements in frontier BY Fscore and convert output TO LIST
    visitedNode <- FIRST element in frontier
    REMOVE the FIRST element from frontier
    ADD visitedNode to visited
    DRAW it using ui
    depth <- depth - 1
        FOREACH(room r in map.room)
            IF(Visited node's X and Y = r's position's X and Y ) THEN
                IF(r has a path) THEN
                    FOREACH(path p in r.path)
                        IF(X and Y position noted in any of the nodes, in either visited or frontier, doesn't
                        matches with that stated on the adjacent room attached on the other end of the p)
                        THEN
                        ParentNode for the room located on the other side of the p <- visitedNode
                        Gscore for the node for the room located on the other side of p <- visitedNode's Gscore + 1
                        Fscore for the room located on the other side of the p <- squareroot of(sum of (square of
                        difference between X value of room located on the other side of the p and goal) and
                        (square of difference between Y value of room located on the other side of the p and goal))
                        ADD the room located on the other side of path into the frontier
                        ENDIF
                    ENDFOR
                ENDIF
                IF(visited node's X and Y = goal node's X and Y and depth greater than 0) THEN
                Return that solveIT method needs to be run with the necessary parameters
                ENDIF
            ENDIF
        ENDFOR
    ENDWHILE
    Return that No solution at given max depth
ENDIF
```

Fig-12: Pseudo code for A Star Limited

Note: Here it can be seen that most of the algorithms had common lists and variables which could have been reused, instead of creating new instances for each algorithm every time the program is run. This had been purposefully done here in order to both ensure that the internal logic remains self-contained and segmented, and that the same objects do not get shared between the algorithms (as it would give an edge to the algorithms which ran later, in case the program is ran back to back for the same map). But if this was created for commercial use (instead of experimenting the with the individual search algorithms), then it would have been better to reuse those codes as it would reduce the overall

load the entire program will have on the memory and storage.

Moreover, the logic used in solveIt method to create the solution has been briefly explained using the figure below via pseudo code:

```
SolveIt(algorithm as method, intial position, current goal position as child node, list of nodes to solution as
expanded)
Make new solution string
Make new path list
Make a new action list
REVERSE the expanded path
FOREACH(point2D p in expanded path)
    IF(p's X and Y = goal's X and Y) THEN
    ADD p to path list
    ENDIF
    IF(anything is inside path) THEN
        IF(X and Y of the ParentNode of the LAST element in path = X and Y of p) THEN
        ADD p to path list
        ENDIF
    ENDIF
ENDFOR
REVERSE the path list
FOR(i = 0 to total number of elements in path list)
    IF(i = total number of elements in path list - 1 )THEN
    BREAK
    ENDIF
    IF(next element in path's Y value = current element in path's Y value - 1 )THEN
    ADD MOVEUP() to action list
    ENDIF
    IF(next element in path's X value = current element in path's X value - 1 )THEN
    ADD MOVELEFT() to action list
    ENDIF
    IF(next element in path's Y value = current element in path's Y value + 1 )THEN
    ADD MOVEDOWN() to action list
    ENDIF
    IF(next element in path's Y value = current element in path's Y value + 1 )THEN
    ADD MOVEDOWN() to action list
    ENDIF
ENDFOR
FOREACH(string a in action list)
solution <- solution + a + "; "
ENDFOR
DRAWPATH using ui and necessary parameters
Return that method, no. of elements in expanded path and the solution
```

Fig-13: Pseudo code for SolveIt

Note: Here the no. of elements in expanded is basically the no. of nodes in the search tree.

Features/Bugs/Missing:

During the creation of this program (and also during the modification of program for automated data collection), certain bugs had been noticed, certain features which had been developed and certain features were missed. All of these have been briefly noted here in this section:

Bugs:

A bug is basically an unexpected problem that arises when external software interferes with the program being observed in such a way that wasn't foreseen by the programmer **[12].** In this program, a bug had been noticed when running the algorithm from command line for large maps like 10v10 and 20v20. There it had been seen that the **UI** would not clean the screen for the previous instance (even though Console.Clear()) has been called at the start of all UI methods) and instead would try to append the new diagram at the bottom of the previous diagram. Even so, it would still automatically scroll the screen so that only the latest diagram is being shown. So if the user doesn't scroll up, they wouldn't be able to notice the glitch.

In the documentation of the code used to clear the screen (Console.Clear()**[13]**), it had been mentioned that the command works by clearing the console's buffer which in turn clears the information being displayed in console window. Thus, this is probably a buffer issue for big maps (as it doesn't occur on 5v5 maps size) which could be resolved by modifying the default buffer set on the program and the console.

Note: Alternatively, it's also possible to resolve it by setting a scroll limit in either the program or command prompt to ensure that user can't scroll up from the current display

Features:

Features are basically distinguishing characteristics of software **[14]**. Hence in this program the following can be considered as features:

1) Creation of Custom algorithms by the inclusion of Depth Limit for both DFS and A Star. This had been done in order to the gauge the capability of the algorithms with their depth limited counterparts and see which is better.

   In case of blind search (DFS and DFSLimited), there had been significant difference in both the average time taken to run the limited algorithm to find the goal and also in average number of nodes searched. But average memory used and average path length of solution was almost similar

   In case of Heuristic Search (AStar and AStarLimited), there has been almost no significant difference between them for the matrices.

   Hence, considering the fact that limiting depth has the inherent disadvantage of not being able to find the goal (due to depth limit being reached before getting to goal), it can be concluded that setting the limit is actually more disadvantageous, especially for Heuristic searches. Thus it would be better to run them without any depth limit

Note: All of these had been done for 5v5, 10v10 and 20v20 map sizes and different outcomes may be possible for larger maps. Furthermore,

currently the depth limit had been set to (Maplength * MapHeight)/2 (ensuring it goes through 50% of the all the rooms regardless of wall or empty) and thus it is also possible that a more optimum depth limit could provide better statistics (i.e. finding goal more often whilst using lesser resource) than its non-limited counterpart.

2) In this program, a GUI had been implemented using the **UI** class which would draw the current agent position, goal position, wall position, empty room's position, frontier's node's position and the visited node's position. Furthermore, it also details the node being present in its frontier, and also the node that is currently being visited (i.e. the one that is being expanded) in text below in a verbose manner (whilst algorithm is processing).

   Moreover, once the solution is found, it would also output the solution path in the diagram (linking initial to goal position) and also say the actions it took to reach the goal in a verbose manner

3) In this program, multiple goal aspect has been programmed in the following manner:

   When the program goes through the file and notices that it has multiple goals (i.e. notices the occurrence of " | " in the line), it automatically create multiple instances of the goal and pass them to the agent one by one. Then it asks the agent to first find the solution from its own position to one of the goal position, and then repeat the process for the 2nd goal position, and so on.

   Thus, in the diagram produced by the UI class, it will always display only 1 initial and only 1 goal position (with the solution path linking them) and in order to see for the alternative goal position the user needs to press Enter (as advised by the program once the solution path to first goal has been displayed) which would display for initial position and the alternative goal.

4) In order to gather necessary data for matrices mentioned in **Search Algorithm section**, an automatic data generation and collection method had been required. In order to accomplish this, a separate program had been created called TestMapMaker (which would be used to make valid map files), a list of scripts had been created (to automatically run algorithms on the maps through command line) and some modifications had been made in the program itself (these have been commented out following the format displayed in **Fig-14.** These would cause it to create relevant files to store the required data, which would then be exported to a Microsoft Excel sheet (using technique discussed in Microsoft's support documentation **[15]**) in order to organize the data and create the charts displayed in the **Search Algorithm section.**

Fig-14: Commenting style used to segment automation commands

Missing feature:

Here, missing feature basically means stating if a required feature had not been implemented in this program. In this case, the feature of enforcing the algorithms to perform their search in the order of Up, Left, Down and Right had not been properly implemented. Initially, it had been assumed that by setting up the path creation between rooms to occur in a certain way, it would cause the search algorithms to insert to the nodes to the frontier following that way as well. Unfortunately, that was only applicable for the BFS algorithm's case which used Queue type collection (stored data in first in first out manner) and not for the other algorithms which used STACKS or LIST type collection. Instead, it would have been better to modify all the algorithms so that they would personally check the adjacent nodes and pass them in the order of Up, Left, Down and Right before passing them in the frontier.

Note: This could be easily resolved by modifying the code after the repeated state check (still inside the "foreach (Path p in r.Paths)" loop for all the

algorithms) so that it would check the path's location's (i.e. the room on the other side of the path) position's X and Y and order it so that room at top (i.e. one with Current node' X position and Current node's Y-1 position) is inserted first, then left(i.e. one with Current node' X-1 position and Current node's Y position), then bottom(i.e. one with Current node' X position and Current node's Y +1 position) and finally right(i.e. one with Current node' X+1 position and Current node's Y position).

Research:

In terms of research, the following two ideas had been applied to the search program:

1) **GUI aspect design:**

This had been done in order to ensure that the user will have a visual on how the algorithm is searching through the rooms to find the goal position and creating the path. In order to do this, the program has been designed so that the Agent calls the Draw function each and every time it inserts a node from frontier to visited (i.e. each time it expands the node). Then the Draw function works in the following manner described using pseudo codes:

```
Draw(intial position, current goal position, list of walls, visited node position, mapwidth, maplength, frontier as STACK)
CLEAR the Console
wallDrawn <- false
FOR (i = 0 to mapWidth)
    FOR (j = 0 to mapLength)
        IF(initial's X and Y position match with j and i)THEN
        WRITE "|i" in Console
        CONTINUE
        ENDIF
        IF(goals's X and Y position match with j and i)THEN
        WRITE "|g" in Console
        CONTINUE
        ENDIF
        IF(visitedNode's X and Y position match with j and i)THEN
        WRITE "|x" in Console
        CONTINUE
        ENDIF
        IF(Any Frontier Node's X and Y position match with j and i)THEN
        WRITE "|f" in Console
        CONTINUE
        ENDIF
        FOREACH(room r in wall)
            IF(r's IsWall = true and r' X and Y position match with j and i)THEN
            WRITE "|w" in Console
            wallDrawn <- true
            BREAK
            ENDIF
            wallDrawn <- false
        ENDFOR
        IF(wallDrawn = false)THEN
        WRITE "| " in Console
        ENDIF

    ENDFOR
    WRITELINE "|" in Console
ENDFOR
IF(visitedNode's X and Y = Goal's X and Y)THEN
WRITELINE in console as a new line that solution has been found with X having VisitedNode's X and Y having VisitedNode's Y value
ELSE
WRITELINE in console "Frontier node:"
    FOREACH(var f in frontier)
    WRITELINE in console f's X and Y coordinates tabbed away
    ENDFOR
WRITELINE in console that visitedNode's X and Y is the X and Y for the current node that is being expanded
ENDIF
SLEEP the THREAD for 100 milliseconds
CLEAR the Console
```

Fig-15: Pseudo code for Draw

Note: The Draw function has been overloaded later so that it can also handle for QUEUE and LIST type collection (in order to accommodate for all algorithm's frontier's collection types). But even so, they still follow the same logic as the one describes in the **Fig-15**.

Furthermore, once the goal has been found and solution path has been created, the agent will call the DrawPath function in order to draw the solution path from initial to goal position. This function works in the following manner described using pseudo codes:

```
DrawPath(intial position, current goal position, list of walls, mapwidth, maplength,
path as a list of nodes)
CLEAR the Console
wallDrawn <- false
FOR (i = 0 to mapWidth)
    FOR (j = 0 to mapLength)
        IF(initial's X and Y position match with j and i)THEN
        WRITE "|i" in Console
        CONTINUE
        ENDIF
        IF(goals's X and Y position match with j and i)THEN
        WRITE "|g" in Console
        CONTINUE
        ENDIF
        IF(Any Path Node's X and Y position match with j and i)THEN
        WRITE "|x" in Console
        CONTINUE
        ENDIF
        FOREACH(room r in wall)
            IF(r's IsWall = true and r' X and Y position match with j and i)THEN
            WRITE "|w" in Console
            wallDrawn <- true
            BREAK
            ENDIF
            wallDrawn <- false
        ENDFOR
        IF(wallDrawn = false)THEN
        WRITE "| " in Console
        ENDIF

    ENDFOR
    WRITELINE "|" in Console
ENDFOR
WRITELINE in Console
```

Fig-16: Pseudo code for DrawPath

Note: Do keep in mind the issue about buffer and Console.Clear() that arose for the GUI. It has been already discussed in the **Bugs section** in the **Feature/Bugs/Missing section** and thus not reiterating it here.

2) **Automatic test case generation and data collection:**

This had been done in order to ensure that the necessary data (i.e. average time taken for algorithm to run, average memory used by the algorithm, average no of nodes searched by the algorithm and average path length of the solution) can be collected in a systematic manner to ensure accuracy and validity of the findings. This had been done using the following 4 part process:

i) Creation of Map Files: This had been done by creating a program called TestMapMaker. Its class diagram is as follows:

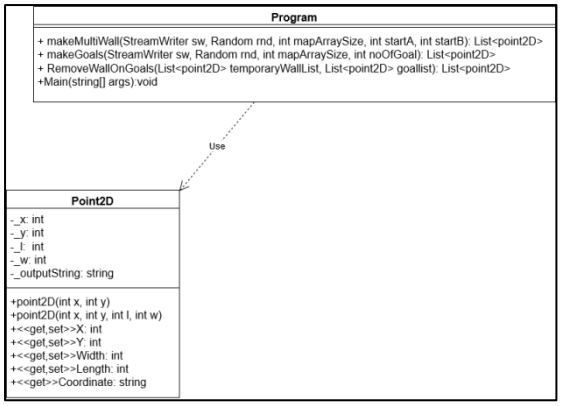


Fig-17: Class Diagram of the TestMapMaker

Note: Here the point2D has been overloaded to that it can account for both normal nodes and wall type nodes.

Its internal functions (i.e. logic behind map creation) can be explained using the Pseudo code provided below:

```
makeMultiWall(streamwriter, random no generator, maparraysize, startA as X position
of start,startB as Y position of start) return a list of point2D
Make a new wallList
Make a new tempwalllist
wallA <- random number from 0 to mapArray size
wallB <- random number from 0 to mapArray size
IF(mapArraySize is less than 2)THEN
mapArraySize <- 2
ENDIF
wallLength <- random number from 1 to mapArray size/2
wallWidth <- random number from 1 to mapArray size/2
finalwallA <- -1
finalwallB <- -1
finalwallLength <- -1
finalwallWidth <- -1
check <- false
FOR(j = wallB to wallB + walllength)
    FOR (k = wallA to wallA + wallWidth)
        If(k = starA and j = startB or K greater than mapArraySize or j greater than
        mapArraySize)THEN
        check <- true
        ENDIF
    ENDFOR
ENDFOR
IF(check = true)THEN
tempwalllist <- makeMultiWall with the necessary parameters
ENDIF
ELSE
finalwallA <- wallA
finalwallB <- wallB
finalwallLength <- wallLength
finalwallWidth <- wallWidth
Return wallList
```

Fig-18: Pseudo code for makeMultiwall method

```
RemoveWallOnGoal( temporary wall list, goal list)
FOREACH(point2D wall in temporary wall list as a LIST())
    FOREACH(point2D goal in goal list)
        FOR(j = wallB to wallB + walllength)
            FOR (k = wallA to wallA + wallWidth)
                IF(k and j match with any of the goal's X and Y)THEN
                REMOVE wall from temporary wall list
                ENDIF
            ENDFOR
        ENDFOR
    ENDFOR
ENDFOR
Return temporary wall list
```

Fig-19: Pseudo code for remove WallOnGoals method

```
makeGoals(streamwriter, random no generator, maparraysize, no of goals) return a list
of point2D
Make a new goalist
goalMakeRND <- no of goal
goalline <- ""
FOR(g = 0 to goalMakeRND)
    ADD new point2D (made using random number generator choosing from 0 to
    maparraysize) to goallist
    WRITELINE that goallist elements's X and Y value to console
ENDFOR
FOR(gg = 0 to goalMakeRND)
    IF(gg = last element of goallist)THEN
    goalline <- goalline + "(X,Y)" where X and Y are that goallist's element's X and Y
    BREAK
    ELSE
    goalline <- goalline + "(X,Y) | " where X and Y are that goallist's element's X
    and Y
    ENDIF
ENDFOR
WRITELINE goalLine to the file
Return goallist


RemoveWallOnGoal( temporary wall list, goal list)
FOREACH(point2D wall in temporary wall list as a LIST())
    FOREACH(point2D goal in goal list)
        FOR(j = wallB to wallB + walllength)
            FOR (k = wallA to wallA + wallWidth)
                IF(k and j match with any of the goal's X and Y)THEN
                REMOVE wall from temporary wall list
                ENDIF
            ENDFOR
        ENDFOR
    ENDFOR
ENDFOR
Return temporary wall list
```

Fig-20: Pseudo code for makeGoals method

```
Main(args)
Run the Garbage collector
FOR(i = 0 to 100)
    myfile <- the path where you want to inject the map files to + Map[i].txt
    USING(Append the Text to the file using Streamwriter sw )
        mapArraySize <- size of the map's length and width, currently setting it as 20
        maxA <- mapArraySize
        maxB <- mapArraySize
        WRITELINE to file using sw "[maxA,MaxB]" where maxA and B are map's max width
        and length
        Make a new Random number generator rnd
        startA <- rnd from 0 to mapArraySize
        startB <- rnd from 0 to mapArraySize
        WRITELINE to file using sw "(startA,startB)" where startA and B are initial X
        and Y position
        finalGoalList <- makeGoals with necessary parameters
        temporaryWallList <- makeMultiWall with necessary parameters
        finalWallList <- RemoveWallOnGoal with necessary parameters
        FOREACH(point2D wall in finalWallList)
            WRITELINE to file using sw "(wall.X,wall.Y,wall.width,wall.length)"
        ENDFOR
    ENDUSING
ENDFOR
```

Fig-21: Pseudo code for Main method

Note: Currently the program had been designed in such a way to check the goal coordinates strictly and remove any wall coordinates set (i.e. the wallx, wally, wallwidth, walllenght set) whenever any of the coordinates present in wall list matches with goal. So if one wants to generate impossible goals just remove the line: “removewall on goal” in Main and instead pass temporarywalllist to finalwalllist

ii) After the maps have been created and placed in the same directory as the search file, Scripts are run from the command line. These have been made using the help of Microsoft’s documentation on GetChildItem **[16]** and on about_automatic_variables’s BaseName section **[17]** and looks like this:

```
get-childitem Map*.txt | foreach-object{.\search.exe $_.BaseName gbfs}
```

Fig-22: Script for GBFS algorithm on map files

In **Fig-22**, the "get-childitem Map*.txt" is basically finding the files that is in Map[x].txt where x can be any character or number. Then it is piping the results to run a foreach loop where it's basically running the following command for all the map files found: ".\search.exe" "the map file's name" "algorithm name (i.e. gbfs)".

iii) Once this is done, the search program runs and notes down:

a) The number of ElapsedTicks during the running of the algorithm, the totalMemoryUsed (by asking GC - Garbage Collector) and the number of nodes that have been searched by asking Agent's Visited list for its count. These are then stored into file called TestdataForAlgo[x].txt (where x will be name of the search algorithm).

b) The optimal path by asking inside the agent's solveIt method to tell the path count and storing it in the file called OptimalPathdata20v20[x].txt (where x will be name of the search algorithm).

Note: This can be further improved by passing all these data into the same file for a specific algorithm and map size and making the name more dynamic (i.e. ***TestDataForAlgo[****mapLength****]v[****mapWidth****][****searchName****]***).

iv) After this, the data are passed to an excel sheet and an average is taken (depending on the algorithms used and the map sizes) before using those values to create the graphs. This has been done to ensure that the data obtained for each algorithm isn't impacted by any random error or any specific structure of any of the produced maps.

Note: Some small research had also been done to compare the effectiveness of depth limit on blind and heuristic algorithms (via the custom searches) but these were not mentioned here as they were not part of "research initiative" mentioned in assignment 1 document and instead briefly spoke about them in the **Feature/Bugs/missing section** as **feature no 1**.

Conclusion:

Overall it has been seen that (in terms of the matrices noted in **Search Algorithm Section**) for small maps (i.e. 5v5) A Star and A Star Limited had the best performance whilst for big maps (10v10 and 20v20) GDBF had best performance.

Furthermore, comparing the custom algorithms with their non-depth-limited counterparts showed that setting the depth-limit doesn't actually provide much advantage in performance compared to its tradeoff in reduction in chance of reaching the goal. But it does provide a bigger impact on blind algorithms case than in heuristic algorithms case.

But, even so, it can't be concluded that GBFS algorithm will be the best choice for any large maps (as it has only been tested for those 3 map sizes and not for bigger maps like 100v100) or that setting of depth-limit is disadvantageous (as it has not been tested for all possible depth-limits and so it's possible that the depth-limit used here isn't the optimal one).

Hence, it will be better to repeat this experiment in the future with bigger map sizes (to test for all algorithms) and also test for varying depth sizes (on those maps) for the custom algorithms.

Note: It will be better to create the future program using a different programming language and editor. That's because for C# code, in Visual Studio Community, it's extremely difficult to convert it into a single .exe format (so much so that if you use the wrong project setup in the beginning, like me, then you will have to redo the entire project again **[18]**). Instead, it would be better to use Python and use pyinstaller (as shown in the guide given in GeekforGeek's "Convert Python script to .exe file" webpage **[19]**) to convert it to .exe directly

Acknowledgements/Resources:

I cannot express enough thanks to my tutor and convener A/Prof. Bao Vo for the feedback, support and resources provided by him throughout the unit COS30019 (Introduction to AI) in the form of

lecture materials, videos, tutorials, etc. Without these, it really would not have been possible for me to develop the Search program.

Furthermore, I would also like to acknowledge the help I had received from the following resources collected online:

- The Robot Navigation Wikipedia page in accurately defining the robot navigation problem,
- Software Feature Wikipedia page in accurately defining software features,
- Gartner's Information Technology Glossary's Bug page in providing me the definition for bug,
- TRCCompSci's wikipedia for providing the syntax for writing pseudo code,
- Hui's book on Robot systems for Rail Transit Application and Anis and Ahmad's book on Unmanned Aerial Systems in accurately portraying the importance of Robot Navigation in Robotics,
- Cheng's Paper on Research on Path Planning of Robot Based on Artificial Intelligence Algorithm in accurately portraying the importance of Robot Navigation in AI,
- Sriniketh's blog on Uninformed Search Algorithms: Exploring New Possibilities (Updated 2023) in pointing out the benefits and drawbacks of a depth limited Search (i.e. DFS Limited),
- TutorialsTeacher's C# collection's Stack page for providing me the understandings I needed to use properly use the STACK type collection in my program,
- TutorialsTeacher's C# collection's Queue page for providing me the understandings I needed to use properly use the QUEUE type collection in my program,
- Microsoft's .NET documentation on Console.Clear() command in helping me understand where and how to use it,
- Microsoft's .NET documentation on Single-file deployment in helping me understand that I had set up the wrong project and that I would need to recreate the project in .NET Framework to ensure that only a single .exe file will be produced,
- Microsoft's Powershell documentation on Get-ChildItem command in helping me understand where and how to use it,
- Microsoft's Powershell documentation on about_Automatic_Variables's $this section's BaseName example in helping me get the idea on where and how I can use it,
- Microsoft's Support documentation on how to transfer data from a delimited text file to an Excel sheet,
- GeeksForGeeks' Convert Python Script to .exe File's page in providing me with a simple step-by-step walkthrough on how python codes could easily be converted into .exe.

References:

[1]“Robot navigation,” *Wikipedia*, Mar. 13, 2019. https://en.wikipedia.org/wiki/Robot_navigation (accessed Apr. 13, 2023).

[2] H. Lui, *Robot Systems for Rail Transit Applications*. Elsevier, 2020. doi: https://doi.org/10.1016/c2019-0-04615-8 (accessed Apr. 13, 2023).

[3] A. Koubaa and A. T. Azar, *Unmanned Aerial Systems*. Elsevier, 2021. doi: https://doi.org/10.1016/c2019-0-00693-0 (accessed Apr. 13, 2023).

[4] C. Yan, “Research on Path Planning of Robot Based on Artificial Intelligence Algorithm,” *Journal of Physics: Conference Series*, vol. 1544, no. 1, p. 012032, May 2020, doi: https://doi.org/10.1088/1742-6596/1544/1/012032. (accessed Apr. 13, 2023).

[5] B. Vo. (2023) Introduction [Powerpoint Slide]. Available: https://swinburne.instructure.com/courses/49155/pages/w1-learning-materials-lectures-+-reading?module_item_id=3357404 (accessed Apr. 13, 2023).

[6] B. Vo. (2023) Intelligent Agents [Powerpoint Slide]. Available: https://swinburne.instructure.com/courses/49155/pages/w2-learning-materials-lectures-+-reading?module_item_id=3360947 (accessed Apr. 13, 2023).

[7] B. Vo. (2023) Problem solving and Search [Powerpoint Slide]. Available: https://swinburne.instructure.com/courses/49155/pages/w3-learning-materials-lectures-+-reading?module_item_id=3362085 (accessed Apr. 13, 2023).

[8] S. J, "Uninformed Search Algorithms in AI | Search Algorithms in AI," *Analytics Vidhya*, Feb. 07, 2021. https://www.analyticsvidhya.com/blog/2021/02/uninformed-search-algorithms-in-ai/ (accessed Apr. 13, 2023).

[9] "CSharp to Pseudo Code - TRCCompSci - AQA Computer Science," *www.trccompsci.online*, Apr. 23, 2021. https://www.trccompsci.online/mediawiki/index.php/CSharp_to_Pseudo_Code (accessed Apr. 13, 2023).

[10] "C# Stack," *www.tutorialsteacher.com*. https://www.tutorialsteacher.com/csharp/csharp-stack (accessed Apr. 13, 2023).

[11] "C# Queue," *www.tutorialsteacher.com*. https://www.tutorialsteacher.com/csharp/csharp-queue (accessed Apr. 13, 2023).

[12] Definition of Bug - Gartner Information Technology Glossary," *Gartner*. https://www.gartner.com/en/information-technology/glossary/bug (accessed Apr. 13, 2023).

[13] Microsoft, "Console.Clear Method (System)," *learn.microsoft.com*. https://learn.microsoft.com/en-us/dotnet/api/system.console.clear?view=net-5.0 (accessed Apr. 13, 2023).

[14] "Software feature," *Wikipedia*, Apr. 22, 2020. https://en.wikipedia.org/wiki/Software_feature (accessed Apr. 13, 2023).

[15] Microsoft, "Import or export text (.txt or .csv) files," *support.microsoft.com*. https://support.microsoft.com/en-us/office/import-or-export-text-txt-or-csv-files-5250ac4c-663c-47ce-937b-339e391393ba (accessed Apr. 13, 2023).

[16] Microsoft, "Get-ChildItem (Microsoft.PowerShell.Management) - PowerShell," *learn.microsoft.com*. https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-childitem?view=powershell-7.3 (accessed Apr. 13, 2023).

[17] Microsoft, "about Automatic Variables - PowerShell," *learn.microsoft.com*. https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables?view=powershell-7.3 (accessed Apr. 13, 2023).

[18]Microsoft, "Create a single file for application deployment - .NET," *learn.microsoft.com*. https://learn.microsoft.com/en-us/dotnet/core/deploying/single-file/overview?tabs=vs (accessed Apr. 13, 2023).

[19] GeekforGeeks, "Convert Python Script to .exe File," *GeeksforGeeks*, Feb. 03, 2020. https://www.geeksforgeeks.org/convert-python-script-to-exe-file/ (accessed Apr. 13, 2023).