Assignment 1 – Tree Based Search

COS30019 - Introduction to Artificial Intelligence

Written & Submitted By:

Zoie Tad-y

(ID Number: 102884743)

Table of Contents

1 Instructions

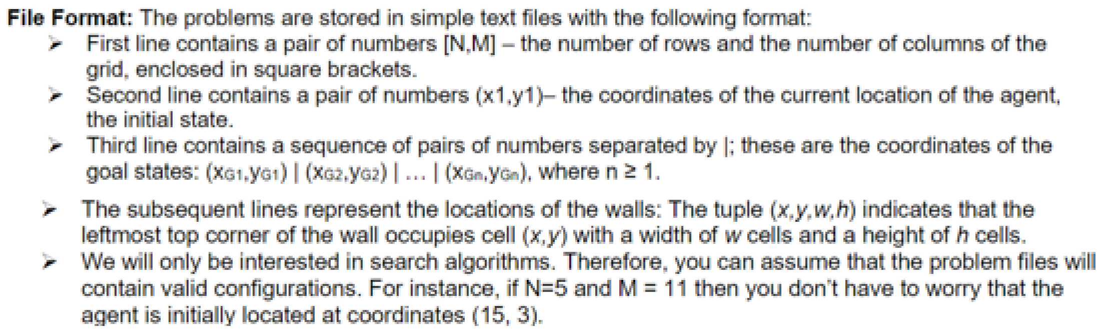
1.1 Base Implementation

The base implementation of the application is accessible via a command-line interface strictly as specified in the *Command Line Operation Section* of the project brief. To run the application, be sure to be in the root of the project folder (i.e., "COS30019-2022 S1-A1") then enter the following:

search <filename> <method>

Where *filename* is the input file that follow the format specified in the project brief as shown in; and *method* is the search method to be used which can be one of the following search implementations as shown in **Table 1**.

Figure 1. File format of file input



File Format: The problems are stored in simple text files with the following format:

- First line contains a pair of numbers [N,M] – the number of rows and the number of columns of the grid, enclosed in square brackets.
- Second line contains a pair of numbers (x1,y1)– the coordinates of the current location of the agent, the initial state.
- Third line contains a sequence of pairs of numbers separated by |; these are the coordinates of the goal states: (x_{G1},y_{G1}) | (x_{G2},y_{G2}) | … | (x_{Gn},y_{Gn}), where n ≥ 1.
- The subsequent lines represent the locations of the walls: The tuple (*x*,*y*,*w*,*h*) indicates that the leftmost top corner of the wall occupies cell (*x*,*y*) with a width of *w* cells and a height of *h* cells.
- We will only be interested in search algorithms. Therefore, you can assume that the problem files will contain valid configurations. For instance, if N=5 and M = 11 then you don't have to worry that the agent is initially located at coordinates (15, 3).

Table 1. Methods

Method	**Search Implementations**
DFS	Depth-first search
BFS	Breadth-first search
GBFS	Greedy best-first
AS	A* ("A Star")
CUS1	Uniform cost search
CUS2	Iterative Deepening A Star
RBFS*	Recursive best first search (does not work as intended)
ModifiedGBFS*	Modified greedy best first search

**These are included in the source code as they were initially the intended custom searches to implement, however these were replaced by IDAS since the author believed that it was a more accurate implementation, and it performs better than the RBFS and ModifiedGBFS implemented. These were still included in the table as the reader may still want to try running them. Note RBFS is a faulty implementation as it should return an optimal solution, but this implementation does not.*

1.2 Extended Implementation

Chapter 6 discusses the intrusion to the additional features in details, however for quick reference the command line argument are as follows

Auto Tester:

test <gridRow> <gridColumn> <numberOfGoalStates> <numberOfWall> <numberOfTests>

Auto Test Generator:

AutogenTestFile <gridRow> <gridColumn> <numberOfGoalStates> <numberOfWall>

2 Introduction

The application is meant to solve the Robot Navigation Problem using specific search strategies. The application will utilise the specifications as mostly defined in the book *Artificial Intelligence a Modern Approach 4th Edition* with slight changes to adapts for the nuances posed by the programming language used and additional requirements for the output. The program will be implemented in Java with the following JDK version and runtime environment:

- OpenJDK version "1.8.0_322"
- OpenJDK Runtime Environment (Temurin) (build 1.8.0_322-b06)
- OpenJDK 64-Bit Server VM (Temurin) (build 25.322-b06, mixed mode)

The Robot Navigation Problem is formally presented to the author as follows:
Given an environment represented as an NxM grid where N and M are both greater than 1, walls occupying some cells in the grid, find a path to visit one of the given goal nodes from an initial location that is an empty cell in the grid. The grid will have always at least one goal node. The problem will be provided as a file, where the file will always contain a valid configuration. All things being equal, the cells should be reached in the order up, left, down, then right.

The search algorithms to be use are for the most part tree-based searches, and deviates in some implementation to a graph search that uses a that keeps track of the reached states to make the strategy complete. In both case each node corresponds to a state in the state space, exploring paths from an initial state to a possible goal state. Effectively, the root of the search tree is the initial state (Russel & Norvig 2018).

To summarise this and to help the reader of this document a glossary defining the terms used previously and a few other helpful terms is provided in **Table 2**. These terminologies are frequently used throughout the rest of the report. This also helps further establish the context of the subsequent discussions.

Table 2. Glossary of Important Terms (Russel & Norvig 2018)

Terminology	**Definition**
Search Algorithm	Generates a solution or failure to come up with a solution given a problem as input.
Initial State	The starting state in a search problem.
Goal State	The state that satisfies the condition in each problem.
Path	Sequence of intermediate states resulting from a sequence of actions.
Solution	A path from the initial state to a goal state.
Search Tree	Depicts various paths between from an initial state to a goal state, where each node may have multiple paths to get to, but each node only has one specific path back to a root node.
State Space	Depicts the set of states possible and the actions that results from one state to another.
Node, or Search Node	Corresponds to a state in the state space
Edges	Connection between nodes, corresponds to actions.
Action	Taken to result from one state to another
Expand	The process of arriving at new node through a set of actions.
Child Node, or Successor Node	Nodes resulting from an expansion of another node.
Parent Node	The node from which the child or successor nodes are expanded from.
Frontier	Organises which node should be considered and expanded next.
Reached	The term used for a state when a node corresponding to it has been created.
Repeated States	States that are encountered more than once in a search tree because of a cycle.
Optimal Algorithm	Solutions are of the lowest path cost
Complete Algorithm	The algorithm is guaranteed to find a solution when there is at least one and able to report if there is none. Certain strategies are incomplete because they can get stuck in loops or cycles.

3 Search Algorithms

This section discusses the search algorithms used in the submitted application namely: Depth First Search (DFS), Breadth First Search (BFS), Greedy Best First Search (GBFS), A Star Search (AS), Uniform Cost Search (UCS), Iterative Deepening A Star Search (IDAS). This section also shows a table comparing the average execution time in the same environment, and the number of nodes instantiated. A discussion of these search algorithms in the context of the Robot Navigation Problem is discussion **Chapter 4** of this document.

3.1 Depth First Search

DFS is a search algorithm that always looks to expand the deepest node possible. It is an incomplete in the context of the given problem, meaning it can get stuck going around a loop of cells if repeated states are not checked. Moreover, DFS is a non-optimal search algorithm meaning it does not come up with a solution that has the lowest possible path, it returns the very first solution it finds. DFS generally works better with problems with a finite search space, and it is a strategy that maintains a relatively small frontier. As such DFS runs with a time complexity of O(b^m) where b is the branching factor, and m is the maximum depth in implementations where repeated states are not checked, implying that it runs infinitely if the search tree has a branch with an infinite depth, and it runs with a space complexity of O(bm) which is great if the search space is finite, but is just as terrible otherwise (Russel & Norvig 2018).

3.2 Breadth First Search

BFS works by expanding nodes on a per depth basis, all nodes in a particular depth with be expanded, then the child nodes of those nodes will all be expanded, and so on until a goal node is reached. This results to getting to a goal node in the shallowest depth possible, hence coming up with an optimal solution. As all possible nodes in each depth is expanded, BFS is bound to encounter the goal state even if there are loops if one exists, hence also making it complete. However, despite the algorithms completeness and cost optimality of its solution, BFS suffers from an exponential complexity, O (b^d), in both time and space complexities where d is the depth of the shallowest solution (Russel & Norvig 2018).

3.3 Greedy Best First Search

GBFS is an informed search algorithm that prioritises the expansion of the node with the least estimated distance to the closest goal node. To do this, the algorithm relies on a priority queue to order the nodes by the evaluation function value, which in this case is just equal to h(n) or the distance to closest goal node. The worst case for time and space complexity for GBFS is O(V) where V is the number of vertices in the state space when repeated states are checked (graph approach) or O (b^d) equivalently (Why is the space-complexity of greedy best-first search is O(b^m) 2020), with the best case of O(bm) where b is the branching factor and m as the max depth. GBFS is not always cost-optimal and is only complete in finite state spaces (Russel & Norvig 2018).

3.4 A Star Search

A star or A* is an informed search algorithm that prioritises the expansion of the node with the least evaluation function value which is calculated by adding the estimated distance to the closest goal node and adding the path cost. To do this, the algorithm relies on a priority queue to order the nodes as well by their f(n) where f(n) = h(n) + (g) where g(n) is the path cost of node n. A Star is optimal and complete with an admissible heuristic. However, A Star suffers from potentially excessively using memory since the number of nodes expanded may be exponential relative to the length of the solution (Russel & Norvig 2018).

3.5 Custom Search Strategy 1

The author chose UCS for the customer search strategy 1. This search strategy is complete but only cost optimal if the branching factor is finite and if a solution exists or the state space is finite. The worst-case time and space complexity of UCS is O ($b^{1+[C^*/\varepsilon]}$) where b is the branching factor, C* is the cost of the optimal solution, and ε is the lower bound of the cost of each possible action (Russel & Norvig 2018). The main drawback of this algorithm is that it does blind search which may result to a waste in time and memory resources (Jaliparthi 2014).

3.6 Custom Search Strategy 2

The author chose IDAS for the customer search strategy 2. IDAS is an iterative deepening version of AS that relies on keeping track of a limit to the evaluation function (i.e., f(n) = h(n) + (g) as in AS) for cutting off an iteration. This f limit is updated whenever a slightly higher f value is encountered, effectively this exhaustively searches for an f-contour, inadvertently directing the search towards to an optimal solution to the goal node best (Russel & Norvig 2018). This is essentially a more space efficient variant of AS. Although it shares the same time complexity as AS contingent on the Heuristic, it does have a relatively better space complexity which is linear at (Korf 1985).

3.7 Final Comparison

Table 3 briefly shows the comparison of the different search algorithms in the context Robot Navigation Problem with all of them checking if the state has been reached before. With these preconditions, it is assumed that there will be loops and the state space is assumed finite. A more extensive version of this comparison via testing is shown in Chapter 6 of this report that considers more scenarios or test cases.

Table 3. Search Strategies Comparison

	DFS[1]	**BFS**[1]	**GBFS**[3,1]	**AS**[3,1]	**CUS1 (UCS[1])**	**CUS2 (IDAS[1])**
Optimal	No	Yes	No	Yes	Yes	Yes
Complete	Yes[1]	Yes	Yes	Yes	Yes	Yes
Time Complexity	O(b^m)	O (b^d)	O (b^d)	O (b^d)	O ($b^{1+[C^*/\varepsilon]}$)	O (b^d)
Space Complexity	O(bm)	O (b^d)	O (b^d)	O (b^d)	O ($b^{1+[C^*/\varepsilon]}$)	O (b)
Performance Comparison in Test File “RobotNav-test.txt”						
Number of Nodes Instantiated	77	86	21	29	35	22

Average Execution Time[2]	5.9041 ms	5.5653 ms	4.6922 ms	4.5383 ms	4.6199 ms	4.9471 ms

[1] The tested/implemented version uses checks for repeated states hence avoid loops. [2] Average Execution Time in 10 attempts in the same environment.[3] Uses Manhattan Distance to the Closest Goal Node as h(n).

Due to the numerous factors that affects execution time, it should be taken with a grain of salt when considered for comparing search algorithms. However, when discrepancies are large such as the with DFS and BFS compared to rest, it is not to hypothesize how their search efficiencies compare with the others. Both in theory and in testing with the given test case, IDAS and AS show to be the best search strategies amongst all the options for solving the Robot Navigation Problem

4 Implementation

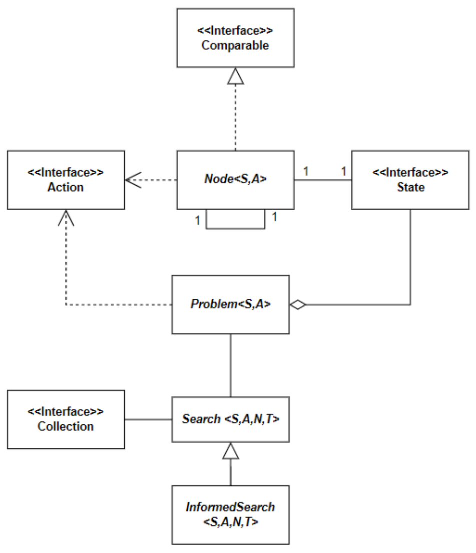
4.1 Design

Due to the size and number of the classes in the application I decided to break down it down by package. For a full UML class diagram please refer to the image named "COS30019-A1-UML-Class-Diagram.jpeg" that was submitted together with this document.

4.1.1 Search Package

This package includes the abstract class and interfaces that relies on generic types for specifying object associate, aggregation, and dependence. This package contains the interfaces Action, State, and the abstract classes Node which implement comparable and requires types that extend Action and State; Problem that requires types that extend Action and State as well; and lastly Search which includes an object that implements Collection which is used as the frontier, which also requires types that extend State, Action, and Node. Their relationship and interaction are shown in **Figure 2** the UML diagram below that excludes the fields and methods.

Figure 2. Search Package



The abstract class Node is general implementation of a search node that requires is to be able to be compared against other search nodes, it contains a state, it keeps track of its parent which is another node, and the action that resulted to it. The abstract class Problem is the general implementation of a problem referred to when searching, it contains initial the information regarding state, goal states, and the invalid states and performs the necessary information for referencing or managing them. Lastly, the abstract class Search, which is a general implementation of a search, it contains the abstract methods specifying the search, expand, and display solution. The Search Class is further extended by the abstract Class informed search which abstracts the ability to get the distance to the closest end goal that uses a heuristic that enables more efficient search.

4.1.2 Robot Navigation

Each abstract class and interface are extended to meet the to fit the context of the Robot Navigation Problem. **Figure 3** shows how the Action, State, Node, and Problems are extended.

Figure 3. Extending Abstract Classes and Interfaces Necessary for Search in the Context of the Robot Navigation Problem

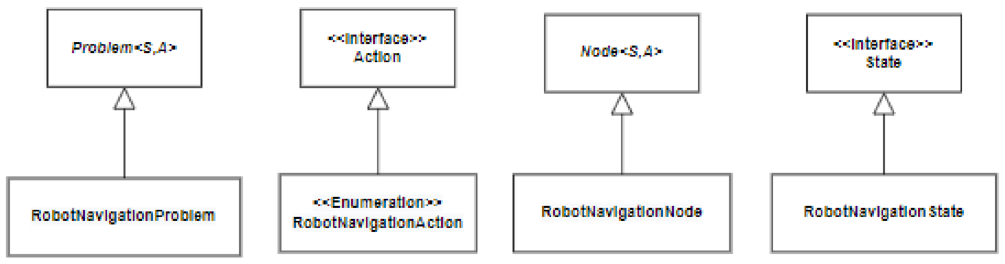
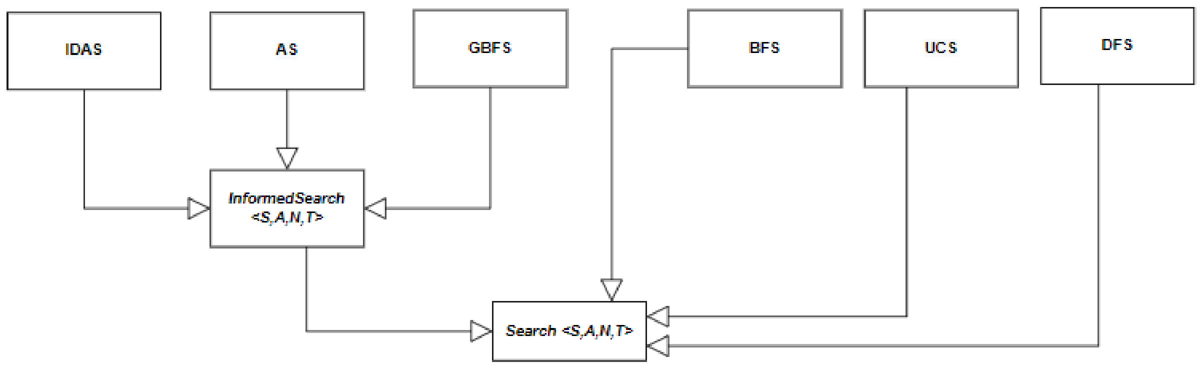


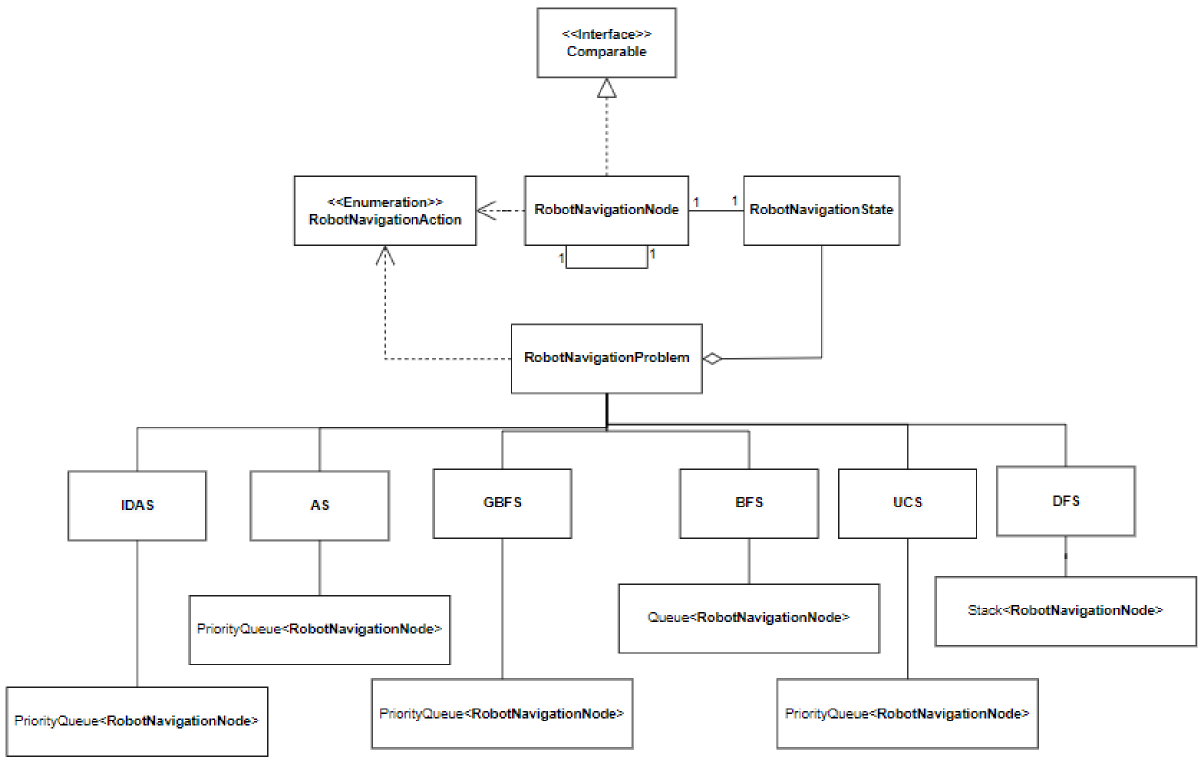
Figure 4 shows how the searches extend from the abstract Search classes. AS, GBFS, and IDAS are all informed searches hence they extend the Informed Search Abstract Class, while the rest extend from the base Search Abstract Class.

Figure 4. Extending Abstract Classes Search and Informed Search for the Searches used for solving the Robot Navigation Problem



The child classes now then effectively take up the position of their super classes in the **Figure 2** as shown in **Figure 5**.

Figure 5. Robot Navigation Package


<<Interface>> Comparable
<<Enumeration>> RobotNavigationAction
RobotNavigationNode
1
1
RobotNavigationState
1
1
RobotNavigationProblem
IDAS
AS
GBFS
BFS
UCS
DFS
PriorityQueue<RobotNavigationNode>
Queue<RobotNavigationNode>
Stack<RobotNavigationNode>
PriorityQueue<RobotNavigationNode>
PriorityQueue<RobotNavigationNode>
PriorityQueue<RobotNavigationNode>

The application is initialised through an object called *RobotNavigationFileReader* which creates the problem from a file path input. This problem is passed on the appropriate search implementation together with the Collection Implementation that it requires in the Main Class. The method or search implementation to be invoked and displayed will depend on the Method entered in the command line argument as shown in Chapter 1 of this report.

4.2 Search Pseudocode

As for the actual search, the implementations of these were made to abide as closely as possible to the available pseudocode provided in Artificial Intelligence a Modern Approach 4th Edition. Small changes were made to either slightly improve the search, fit with the design better, and keep track of the number of Nodes created. This section shows the pseudocode of how the searches were defined.

Table 4. Search Pseudocode

Method	Pseudocode for Implemented Search	Pseudocode for Expand

DFS	function DFS() returns a solution node or failure
	    node<-NODE(problem.initial)
	    NumberOfNodesCreated <- NumberOfNodesCreated + 1
	    if problem.isGoal(node.state) then return node
	    frontier<-priority queue ordered by f, with node as an element
	    while not isEmpty(frontier) do
	        node<-pop(frontier)
	        if node.state is not in reached then
	            reached[node.s]<-node
	            for each child in expand(node) do
	                s<-child.state
	                if problem.isGoal(child.state) then return node
	                if s is not in reached then
	                    add child to frontier
	    return failure
		function expand(node) yields nodes
		    s<-node.STATE
		    for each action in problem.actions(s) do
		        tempStack <- action
		    while not isEmpty(tempStack)do
		        action <- tempStack.pop
		        sc<-problem.result(s, action)
		        cost<-node.PATH-COST + problem.actionCost(s, action,sc)
		        NumberOfNodesCreated <- NumberOfNodesCreated + 1
		        yield NODE(sc, cost, action, node)

BFS	function BFS() returns a solution node or failure
	    node<-NODE(problem.initial)
	    NumberOfNodesCreated <- NumberOfNodesCreated + 1
	    if problem.isGoal(node.state) then return node
	    frontier<-priority queue ordered by f, with node as an element
	    reached<-hashmap with key problem.initial and value node
	    while not isEmpty(frontier) do
	        node<-pop(frontier)
	        for each child in expand(node) do
	            s<-child.state
	            if problem.isGoal(child.state) then return node
	            if s is not in reached then
	                reached[s]<-child
	                add child to frontier
	    return failure
		function expand(node) yields nodes
		    s<-node.STATE
		    for each action in problem.actions(s) do
		        sc<-problem.result(s, action)
		        cost<-node.PATH-COST + problem.actionCost(s, action,sc)
		        NumberOfNodesCreated <- NumberOfNodesCreated + 1
		        yield NODE(sc, cost, action, node)

GBFS	function GBFS() returns a solution node or failure
	    node<-NODE(problem.initial)
	    node.SequenceNumber <- Number of Nodes Created
	    node.f <- 0 + h(problem.initial)
	    NumberOfNodesCreated <- NumberOfNodesCreated + 1
	    frontier<-priority queue ordered by f, with node as an element
	    reached<-hashmap with key problem.initial and value node
	    while not isEmpty(frontier) do
	        node<-pop(frontier)
	        if problem.isGoal(node.STATE) then return node
	        for each child in expand(node) do
	            s<-child.state
	            if s is not in reached or child.f < reached[s].f then
	                reached[s]<-child
	                add child to frontier
	    return failure
		function expand(node) yields nodes
		    s<-node.STATE
		    for each action in problem.actions(s) do
		        sc<-problem.result(s, action)
		        distance <- H(sc)
		        if (sc is not reached or distance < reached[sc].g)
		        newNode <- NODE(sc, action, node)
		        newNode.f <- 0 + distance
		        newNode.SequenceNumber <- NumberOfNodesCreated
		        NumberOfNodesCreated <- NumberOfNodesCreated + 1
		        yield newNode

AS	function AS() returns a solution node or failure
	    node<-NODE(problem.initial)
	    node.SequenceNumber <- Number of Nodes Created
	    node.f <- 0 + h(problem.initial)
	    NumberOfNodesCreated <- NumberOfNodesCreated + 1
	    frontier<-priority queue ordered by f, with node as an element
	    reached<-hashmap with key problem.initial and value node
	    while not isEmpty(frontier) do
		function expand(node) yields nodes
		    s<-node.STATE
		    for each action in problem.actions(s) do
		        sc<-problem.result(s, action)
		        distance <- H(sc)
		        if (sc is not reached or distance + 1 + node.g < reached[sc].f)
		        newNode <- NODE(sc, action, node)

	node<-pop(frontier)
	    if problem.isGoal(node.state) then return node
	    for each child in expand(node) do
	      s<-child.state
	      if s is not in reached or child.f < reached[s].f then
	        reached[s]<-child
	        add child to frontier
	  return failure
		newNode.f <- node.g + 1 + distance
		    newNode.SequenceNumber <- NumberOfNodesCreated
		    NumberOfNodesCreated <- NumberOfNodesCreated + 1
		    yield newNode

CUS1	function UCS() returns a solution node or failure
	  node<-NODE(problem.initial)
	  node.SequenceNumber <- Number of Nodes Created
	  node.f <- 0 + h(problem.initial)
	  NumberOfNodesCreated <- NumberOfNodesCreated + 1
	  frontier<-priority queue ordered by f, with node as an element
	  reached<-hashmap with key problem.initial and value node
	  while not isEmpty(frontier) do
	    node<-pop(frontier)
	    if problem.isGoal(node.state) then return node
	    for each child in expand(node) do
	      s<-child.state
	      if s is not in reached or child.f < reached[s].f then
	        reached[s]<-child
	        add child to frontier
	  return failure
		function expand(node) yields nodes
		  s<-node.STATE
		  for each action in problem.actions(s) do
		    sc<-problem.result(s, action)
		    if (sc is not reached or node.g + 1 < reached[sc].f)
		    newNode <- NODE(sc, action, node)
		    newNode.f <- node.g + 1
		    newNode.SequenceNumber <- NumberOfNodesCreated
		    NumberOfNodesCreated <- NumberOfNodesCreated + 1
		    yield newNode

CUS2	function IDAS() returns a solution node or failure
	  node<-NODE(problem.initial)
	  node.SequenceNumber <- Number of Nodes Created
	  node.f <- 0 + h(problem.initial)
	  NumberOfNodesCreated <- NumberOfNodesCreated + 1
	  flimit<-inf
	  while true do
	    sol , flimit <- DFSC(node, flimit)
	    if sol is non-null then return sol
	    else return failure
	
	function DFSC(node, flimit) returns a solution node or failure
	  if node.f > flimit then return null, node.f
	  if problem.isGoal(node) then return node, flimit
	  for each child in expand(node)
	    sol, new flimit <- DFSC(child, flimit)
	    if sol is non-null then return sol, flimit
	    nextF <- Min(flimit, new flimit)
	  return null nextF
		function expand(node) yields nodes
		  s<-node.STATE
		  for each action in problem.actions(s) do
		    sc<-problem.result(s, action)
		    distance <- H(sc)
		    if (sc is not reached or distance + 1 + node.g < reached[sc].f)
		    newNode <- NODE(sc, action, node)
		    newNode.f <- node.g + 1 + distance
		    newNode.SequenceNumber <- NumberOfNodesCreated
		    NumberOfNodesCreated <- NumberOfNodesCreated + 1
		    yield newNode

The main difference this implementation has from the specifications in the reference materials (Artificial Intelligence a Modern Approach 4th Edition) is that it includes a check for previous reached nodes and evaluation function comparison between the previously reached state and the current state that has the same hash code during expansion, this allows the algorithm to instantiate less nodes than necessary. For comparison the expand function for BFS and DFS does not do this, hence they each generate 77 and 86 nodes respectively.

4.3 Method Invocation Pseudocode

For reading the problem from the file, executing the search, and displaying the solution, the following pseudocode is executed:

```
file <- FILEREADER("filepath")
problem <- file.readRobotNavigationFile()
frontier <- FRONTIER()
search <- SEARCH(problem, frontier)
node <- search.performSearch()
search.displaySolution(node, "filepath")
```

5 Features/Bug/Missing

The application meets the base requirements as set in the project brief. The application can take in a file with the specified format, read the problem from it, perform the search, and return a solution. It returns an optimal solution for search algorithms that it is expected to return an optimal solution. The searches arrive at a solution with a reasonable amount of node created. The application also returns "No solution found" when there is none. The author also believes that the appropriate data structures were used. However, there are several improvements that can be made on the both the design and code.

List of design limitations, inefficiencies, and unnecessary redundancies:

1. The node object contains information that is not needed (i.e., visited) - given that a HashMap for keeping track of reached states has been provisioned in the search, the visited field is not necessary and is just dead weight in terms of space consumption.
2. The HashMap was unnecessary - the author intended the use of a HashMap for keeping record of repeated states as it makes it fast and easy to retrieve instantiated nodes given a state as key. However, the retrieval of nodes from the HashMap was never used. A simpler collection object that requires less memory would have been a better option, or the HashMap could have been implemented in a way that it only stores states as keys with the evaluation function as value, since the Nodes are not needed with most of the searches.
3. Repeated Code in the Search – Upon assessing the pseudocode and actual code, there are a lot of repeated code in the expand function and perform search function where the only implementation that shows a substantial difference is the IDAS. These could have been implemented as default methods, overridden only when necessary.
4. Inefficient Reached Nodes Check in DFS and BFS – Although the implementations of these two are strongly based on the specifications in the reference material, it seems that the author may have misunderstood the reference as these two instantiate more search nodes than the total possible states in the grid, meaning some states have been instantiated more than once. This means that there is an inefficiency in recording child nodes or checking if states have already been created. The author could have chosen to add a reached state check in the expand method, however the author wanted to at least have the BFS and DFS implemented as close as possible to the pseudocode provided in the reference material.
5. Ineffective use of Access Modifies – There are several methods and properties that could have been given more appropriate and effective access modifiers as to improve the security of the design.
6. Unnecessary Method, Properties, and Constructors – The author tested and experimented earlier on in the project, this led to a lot of getters and setter, methods, and constructors that the application could do without.
7. Coupling – From chapter 4, the design shows that the search algorithms are implementations specific to the Robot Navigation Problem, however for improve reusability it would have been better to have a base or general implementation of the search algorithms that is only then extended to meet the requirements of the Robot Navigation Problem.
8. IDAS does not need a priority queue - As IDAS was the last search algorithm to be implemented, the author was not able to consider such algorithms that does not relay on a frontier, but due to the design of the application had to instantiate one regardless. Search algorithms such as IDAS and RBFS both relay on the call stack as their way of organising nodes to expand.
9. Better design patterns could have been used – The design could have benefitted from a NodeFactory object that is solely responsible for expanding a given node, and a EvaluationFunction object that is solely responsible for managing F, G, and H values. For the research initiative, a PerformanceMetrics object could have been made and integrated with the search algorithms to make testing and tracking performance better and more accurate.
10. Using int – although it is convenient to use ints in things such as the coordinates, it does limit the application. For example, since most all other values are int, using linear distance as a heuristic may require the use of Double, which would result to comparison on unmatched type which will require an additional step or two. Using ints also limits the size of the grid, although Integer.MAX_VALUE is a relatively big value, using bigger values through double would have allowed the author to explore further the limitations of the search algorithms.
11. The author attempted to implement RBFS but could not resolve a bug that prohibits nodes from having a G value greater than 0. As other search algorithms can do this, the author hypothesizes that it is a design issue, where setting the g value of a node in a recursive process hinders the comparison of objects.

6 Testing Research Initiative

6.1 Overview

As the project brief states that the input files will always be in a valid configuration as seen in **Figure 1**. The focus of this testing would be to a larger range of problems that each search algorithm implemented is meant to solve. Each algorithm will be tested for their solutions optimality, execution time, and nodes instantiated.

6.2 Authors Idea

The author’s idea is to create many random valid problems for the Robot Navigation Problem programmatically, test the problem against the search algorithms implemented, write the data of the test into a csv file, simplify the data in excel, and validate if the authors assumptions regarding the optimality, execution time, and nodes instantiated are correct.

Here are the author’s assumptions regarding the algorithms implemented:

1. All solutions return by all the search algorithms except for DFS and GBFS will be optimal as shown in **Table 3**.
2. A start will have the fastest average execution time (time of search).
3. GBFS will on average create the least number of nodes.

6.3 Set Up Plan

As BFS is the simplest to implement but at the same time is complete and optimal, it will serve as the benchmark for testing for optimality. The number of times it returns a solution equal to the shortest path will also be counted and used a cross reference for validating that it is really finding an optimal path. There will be 200 autogenerated problems as to hopefully normalise the distribution of different metrics tracked. Each problem will follow the base specification of being a 10x10 grid with 3 goal nodes randomly placed in an empty grid cell, and 50 walls randomly placed in an empty grid cell.

6.4 Results

Table 5. Cross Checking for Optimality

Column1	**DFS**	**AS**	**UCS**	**GBFS**	BFS	**IDAS**
Number of solutions that matches the solution of the benchmark (BFS)	122	200	200	193	200	177
Number of times the solution is equal to the minimum among all search algorithms	122	200	200	193	200	177

Table 5 shows that only AS, UCS, and BFS returned an optimal solution in all the autogenerated problems. The authors assumptions are right that DFS and GBFS will not perform optimally. However, IDAS raises a concern as it is supposed to be optimal in the context of the Robot Navigation Problem. This possibly indicates there something wrong in terms of implementing the IDAS.

Table 6. Number of Nodes, Solution Length, Duration

	DFS			**AS**			**UCS**			**GBFS**			**BFS**			**IDAS**		
	NoN	**SL**	**D**	**NoN**	**SL**	**D**	**NoN**	**SL**	**D**	**NoN**	**SL**	**D**	**NoN**	**SL**	**D**	**NoN**	**SL**	**D**
Min	2	0	200	2	0	300	2	0	200	2	0	1200	2	0	200	2	0	500
Average	33.405	8.17	40062	12.945	4.22	18547	17.835	4.22	23863	12.01	4.315	20851.5	29.565	4.22	24350.5	13.71	4.745	22551
Max	144	38	1068300	57	21	331300	57	21	312300	57	21	359900	144	21	642800	87	23	872700

Table 6 shows that AS executes search the fastest on average while DFS is significantly slower. Generally, AS, UCS, GBFS, IDAS, and BFS executed the search in relatively close timespans on average. In terms of the Nodes created, DFS appears to have instantiated the most on average, and GBFS instantiates the least. This contradicts **Table 3** once again as DFS was expected to perform the best in terms of nodes instantiated.

6.5 Notes on the Extended Features

There are essentially two extensions to this project. There is the auto tester that creates Robot Navigation Problems in Memory and directly passes it to an object that will run the search algorithms implements on the problems generated, which then outputs a csv file containing the performance metrics of the search (i.e., length of solution, duration of search, and the number of nodes it instantiated) although the author understands that these metrics are crude ways of evaluation an algorithms performance, the volume of tests is meant to compensate for the variability of the machine and the problem.

For the user to interface with the auto tester, they should be at the root directory of the project (i.e., “COS30019-2022 S1-A1”) and run the following commands with the following arguments:

```
test <gridRow> <gridColumn> <numberOfGoalStates> <numberOfWall> <numberOfTests>
```

the file output will be named "*rawPerformanceTestResult.csv*" an example out is available in the root directory of this project. The format of the output file be of the following:

"

<method1>, <Number of nodes in problem 1>, <Number of nodes in problem 2>…

<method1>, <solution length in problem 1>, < solution length in problem 2>…

<method1>, <search duration in problem 1>, < search duration in problem 2>…

<method2>, <Number of nodes in problem 1>, <Number of nodes in problem 2>…

…

"

Note: the duration will be nanoseconds

The second extension which is a subcomponent of the first extension, is the auto generation of a test file that specifies a Robot Navigation Problem. To interface with this feature, users should be at the root directory of the project (i.e., "COS30019-2022 S1-A1") and run the following commands with the following arguments:

AutogenTestFile <gridRow> <gridColumn> <numberOfGoalStates> <numberOfWall>

This will produce a file called "autogen_testfile" that follows the format specified in the project brief (see **Figure 1**).

The source code for these features are in the package ExtensiveTester, exposed to the users through the files Tester.java GenerateTestCaseFile.java with their own respective batch files.

7 Conclusion

This concludes the report for this project submission. This report outlined the Robot Navigation Problem to be solved, the search algorithms implemented in the project, the design of the application, the pseudocode, or the logic flow of the search algorithms, and then chapter 5 was an evaluation of the features, bugs, and missing components of the project. Then the research initiative section focused on how to do large amounts of automated testing that produces data that will allow us to assess the search algorithms to some extent of validity. This section also covered how the user can utilise or invoke these extensions. It is in this section as well that the search implementations of IDAS must have been incorrect or inaccurate given that in theory it should return an optimal solution but testing states otherwise. Chapter 6 also dug deeper into the average performance of the search algorithms by recording and evaluating the number of nodes each search creates and the time these to complete. Ultimately it is showed in Chapter 3 and 6 that A star is the best search in theory, and in the author's implementation. GBFS was efficient as seen in chapter 3 but non-optimal as shown in Chapter 6. UCS performed well achieving an optimal solution in all 200 test problems whilst substantially creating less nodes on average. IDAS could have been better or just as good as AS if it was implemented correctly as it would in theory reduce the number of nodes it creates. The BFS and DFS implementations shows to be the strategies that could use some improvements, mainly earlier checking of previously reached states as to prevent more use of memory. The author also realised how much redundancies are there in the code, with some dead weight unused variable, that when cleaned up would result in less memory use overall.

8 Acknowledgements/Resources

The author of this report would like to thank the unit Convenor and Tutor as the lectures and classrooms sessions really aided me in fostering my interest in the field and the explanations improved my understanding of the subject matter. The author also acknowledges the writer of the book Artificial Intelligence a Modern Approach 4th Edition as it is a really well written resource with helpful diagrams and thorough discussions.

9 References

Heap, D 2002, *Depth-first search (DFS)*, www.cs.toronto.edu, viewed 21 April 2022,

<http://www.cs.toronto.edu/~heap/270F02/node36.html#:~:text=Here>.

Jaliparthi, R 2014, *PATH FINDING -Dijkstra's Algorithm*, viewed 23 November 2020,

<http://cs.indstate.edu/~rjaliparthive/dijkstras.pdf>.

Korf, R 1985, *Depth-First Iterative-Deepening: i z An Optimal Admissible Tree Search**.

Russel, S & Norvig, P 2018, *Artificial Intelligence: a modern approach.*, Prentice Hall.

Why is the space-complexity of greedy best-first search is O(b^m)? 2020, Artificial Intelligence Stack Exchange, viewed 21 April 2022, <https://ai.stackexchange.com/questions/17971/why-is-the-space-complexity-of-greedy-best-first-search-is-mathcalobm>.