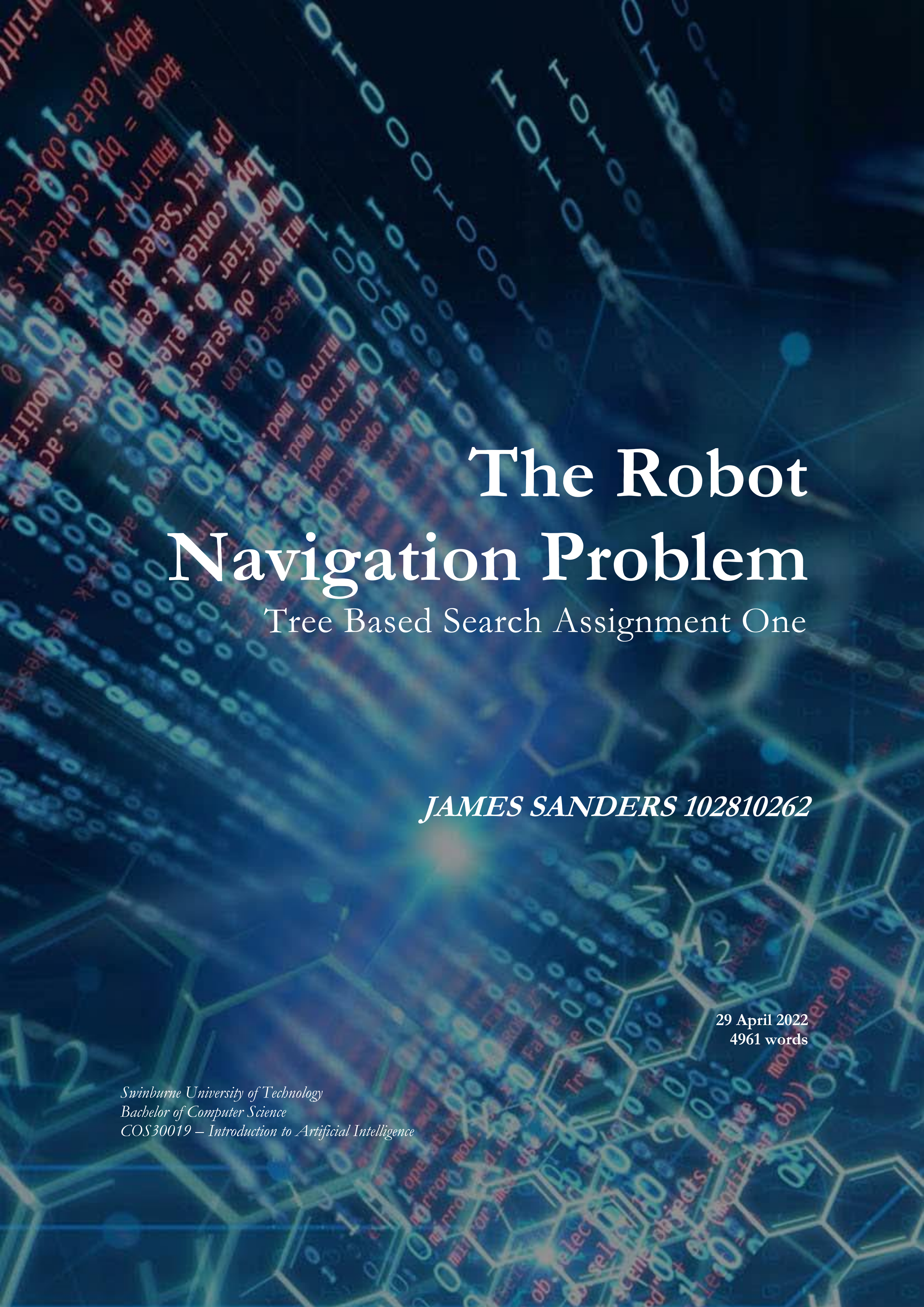


The Robot Navigation Problem

Tree Based Search Assignment One

JAMES SANDERS 102810262

29 April 2022
4961 words

Swinburne University of Technology
Bachelor of Computer Science
COS30019 – Introduction to Artificial Intelligence

1 CONTENTS

FIGURES

1 Instructions

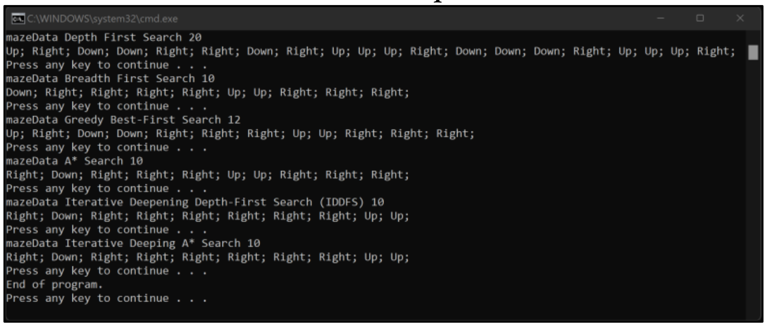
To run the executable program, search.exe, just double click the batch file, search.bat, contained in the same directory as the executable. However, if different maze data is desired, just open the bat file in a text editor, and there replace the value of the `_MAZEDATA` variable to the relative path name of the new maze data file as shown below in the batch code. Then, save and re-run the search.bat file by double clicking it. The executable will launch a command prompt console and iteratively run the search.exe program with all search methods, pausing to display the output of each as shown below in the console output.

Batch Code

```
@ECHO OFF
:The relative path name of maze data.
SET _MAZEDATA=mazeData
FOR %%G IN (DFS BFS GBFS AS CUS1 CUS2) DO (CALL :SUBROUTINE %%G)
ECHO End of program.
PAUSE
GOTO :EOF
:SUBROUTINE
 search.exe %_MAZEDATA% %1
 ECHO(
 PAUSE
 GOTO :EOF
```

Figure 1. Batch code of serach.bat.

Console Output



```
mazeData Depth First Search 20
Up; Right; Down; Down; Right; Right; Down; Right; Up; Up; Up; Right; Down; Down; Down; Right; Up; Up; Up; Right;
Press any key to continue . . .
mazeData Breadth First Search 10
Down; Right; Right; Right; Right; Up; Up; Right; Right; Right;
Press any key to continue . . .
mazeData Greedy Best-First Search 12
Up; Right; Down; Down; Right; Right; Right; Up; Up; Right; Right; Right;
Press any key to continue . . .
mazeData A* Search 10
Right; Down; Right; Right; Right; Up; Up; Right; Right; Right;
Press any key to continue . . .
mazeData Iterative Deepening Depth-First Search (IDDFS) 10
Right; Down; Right; Right; Right; Right; Right; Right; Up; Up;
Press any key to continue . . .
mazeData Iterative Deeping A* Search 10
Right; Down; Right; Right; Right; Right; Right; Right; Up; Up;
Press any key to continue . . .
End of program.
Press any key to continue . . .
```

Figure 2. Console output of search.exe.

2 Introduction

When it comes to intelligent agents (IA) and comparing the performance of algorithms they deploy, researchers use example problems. The robot navigation problem is among one of the best known, along with the grid world problem, each of these two problems can be distinguished as either a real-world problem or a standardised problem. A real-world problem, for instance robot navigation, according to Russell & Norvig (2020) solutions to this problem are widely used, while the problem formulation is idiosyncratic, not standardized as each robot can have unique sensors, generating different data. A standardised problem, for instance grid world, can be given a concise, precise narrative, therefore, making it ideal to demonstrate or practice many different problem-solving methods for comparing the performance of algorithms.

This robot navigation problem is in fact a grid world problem, being a two-dimensional rectangular array of square cells in which IAs can move from cell to cell. Typically, the IA can move to any obstacle-free adjacent cell—horizontally or vertically, each cell can be reached only from one of its four neighbours.

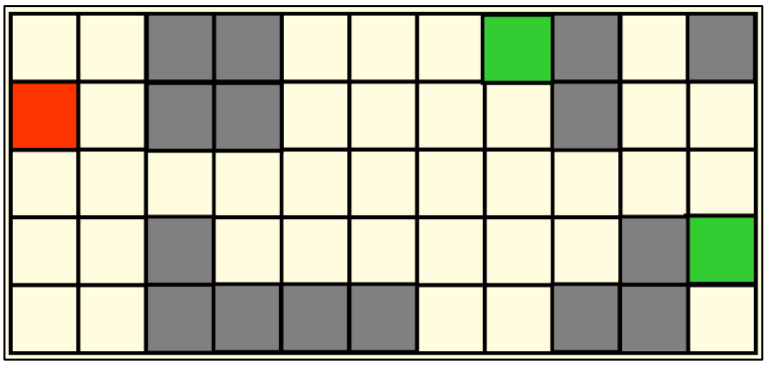


Figure 3. One possible grid world environment.

Figure 1 shows one possible grid world environment, the red cell represents the initial location of the IA, the green cells represent the goal states, and the grey cells represent walls or impassable cells. The IA can travel to any adjacent yellow cell. This grid world can be considered an undirected weighted graph, an abstract data structure (ADS). Figure 2 shows the graph with each edge having a weight of one, and each vertex any cell but a grey cell. In reference to figure 1, vertex A is the same as the red cell while vertices Z & AL are the same as the green cells. For this standardised grid world, the graph has not been explicitly created as the two-dimensional array can be represented as objects, therefore, each cell object can hold information about its neighbour.

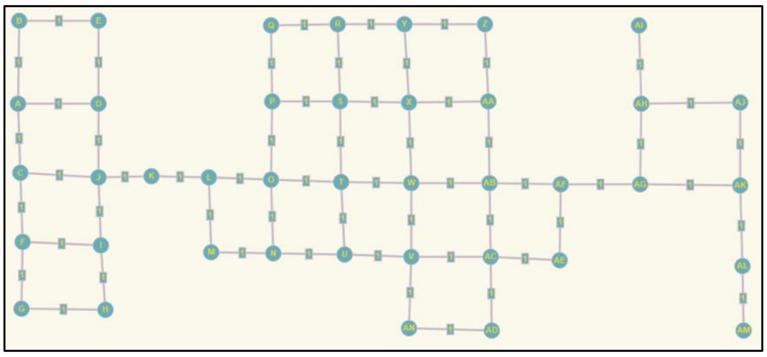


Figure 4. Undirected weighted graph of above's grid world problem.

Furthermore, the graph in figure 2 has no cycles making it an acyclic graph, and as it is an undirected graph where every pair of vertices are connected by a path, it is called connected. This brings us to another type of ADS called trees, starting at the root node, trees expand out along edges to any number of child nodes, all the way to leaf nodes. Additionally, a tree can be an acyclic connected graph like the one in figure 2, this type of tree is called a spanning tree, spans all vertices.

Taking the above standardised problem, six different search algorithms have been used to find the solution, hereby making it possible to compare the performance of each algorithm. Half of which are uninformed search methods while the other half are informed search methods.

2.1 UNINFORMED

An uninformed search algorithm is given only the problem statement and no indication of how near a state is to the goal(s). Uninformed searches cannot solve exponential-complexity search problems except for the simplest of cases. Uninformed search algorithms generate a search tree in an attempt to discover a solution and vary significantly depending on which node they expand first after the root node:

2.1.1 Depth-First Search (DFS)

Depth-first search always advances to the deepest level of the search tree, in which the nodes have no further children. The search then returns to the parent node one level back, before moving on to unexplored children. Because depth-first search is not a cost-optimal search, it returns the first solution encountered, even if it is not the cheapest.

It is efficient and complete for finite state spaces that are trees, however, for acyclic state spaces, it may end up expanding the same state numerous times along different paths, but it will systematically search the full space. Also, because it is possible to get caught in an infinite loop in cyclic state spaces, certain depth-first search implementations verify each new node for cycles. On top of that, depth-first search is not systematic in infinite state spaces, it can become stuck travelling down an infinite path even when there are no cycles. As a result, depth-first search is incomplete.

After evaluating the properties stated above, it appears that depth-first search has numerous flaws. However, since there is no need to maintain a record of reached nodes, depth-first search has substantially lower memory requirements for scenarios where a tree-like search is practical, and the frontier is much smaller than the frontier of a breadth-first search, for example.

A depth-first search in a finite graph-like state-space, such as the one shown above in figure 4, DFS takes time proportional to the total number of states $O(|V|)$, where $|V|$ is the number of vertices and has a space complexity of just $O(bd)$, where b is the branching factor and d is the depth of the shallowest solution in the graph, giving it linear space complexity.

Some search problems that may take zettabytes of memory with breadth-first search can be solved with depth-first search in just kilobytes. Due to its minimal use of memory, depth-first search has been recognised as the fundamental core of various branches of AI, including constraint satisfaction, propositional satisfiability, and logic programming. (Russell & Norvig, 2020)

2.1.2 Breadth-First Search (BFS)

Breadth-first search is an appropriate strategy when all actions have the same cost making it cost-optimal in this case. However, BFS is not cost-optimal for problems where all actions do not have the same cost, although, it is complete in either case. As always, the root node is expanded first, followed by all its successors, followed by their successors, and so forth. A breadth-first search's frontier is likened to a circle's ever-expanding circumference. As a result, this is a methodical search approach that is complete even for infinite state spaces, also known as a brute-force search. Time and space complexity are both $O(b^d)$, signifying that it has exponential time and space complexity. (Russell & Norvig, 2020)

2.1.3 Iterative Deepening Depth-First Search (IDDFS)

To prevent depth-first search from searching perpetually, we can implement DFS as an iterative deepening search or depth-limited search, in which we give a depth limit, ℓ, and treat all nodes at depth ℓ as though they had no successors, with $O(b^{\ell})$ for time complexity and $O(b\ell)$ for space complexity. Unfortunately, if the depth limit is inaccurately specified, the algorithm will never reach an actual solution, rendering it incomplete once more. Although, based on the problem knowledge, an appropriate depth limit can be specified. Unfortunately, for the majority of problems, the appropriate depth limit will not be known until it is solved. The difficulty of accurately specifying the depth limit value is resolved by iterative deepening search, which tests all potential values until either a solution is found, or the search returns the failure value rather than the cut-off value.

The many strengths of depth-first and breadth-first search are combined in iterative deepening. It has much the same time and space requirements as depth-first search, when a solution is found, it has space complexity of O(bd), and time complexity of O(b^d). Or, when no solution is found in finite state spaces, it has space complexity of O(bm) where m is the maximum depth, with time complexity of O(b^m). Giving it linear space complexity. Iterative deepening, like breadth-first search, it is optimal for problems where all actions have the same cost and is complete on finite acyclic state spaces, or on any finite state space when nodes are tested for cycles.

When the search state space is too big to fit into memory and the depth of the solution is uncertain, iterative deepening is the ideal uninformed search method. (Russell & Norvig, 2020)

2.2 INFORMED

Informed search methods have access to a heuristic function h(n) which calculates the estimated cost of a solution from n, or simply, the estimated distance from a given state to the closest goal state(s). Therefore, informed search algorithms can locate solutions more efficiently than uninformed search algorithms. However, the quality of the heuristic function determines the performance of heuristic search algorithms.

2.2.1 Greedy Best-First Search (GBFS)

Greedy best-first search is a type of a best-first search algorithm as it evaluates potential paths using f(n)=h(n), where the lowest h(n) nodes are expanded first, always choosing the path with the lowest heuristic value and f(n) is the path-based evaluation function. It is not always perfect, while it is typically efficient; yet, in the worst-case scenario, it may never locate a solution at all, comparable to depth-first search. Furthermore, greedy search is not optimal and can lead to extremely expensive paths. This may occur when the algorithm embarks on what it perceives to be the shortest expected path to the nearest goal state(s), even if the goal state proves to be more expensive in the end. (Coppin, 2004)

In finite state spaces, greedy best-first search is complete, but not in infinite ones. O(|V|) is the worst-case time and space complexity. However, with a decent heuristic function, the complexity can be decreased dramatically, with some cases nearing O(bm). (Russell & Norvig, 2020)

2.2.2 A*

A* algorithms are similar to greedy best-first search algorithms, but A* uses a slightly more advanced evaluation function to choose a path. The greedy search method always selects the next node that appears to be closest to the goal, regardless of the cost of the journey to that node thus far.

The A* algorithm works on much the same principle as this heuristic search method, but adds the cost of the journey thus far, scoring nodes using the following path evaluation function to determine appropriate paths through the search space: f(n)=g(n)+h(n). Where g(n) is the cost of the journey thus far, and h(n) is the current node's underestimated distance to the nearest goal state(s), also known as an admissible heuristic. If a heuristic never overestimates the cost of reaching the goal, it is admissible. If the heuristic is admissible, A* is optimal and complete. Therefore, it is guaranteed that a solution will be found, and that solution will be the shortest path.

When A* is implemented, the function f(n) for successor nodes is calculated, and the nodes with the lowest f(n) values are expanded first. A* is said to be optimally efficient in the sense that it will expand the fewest possible paths to the goal node. Once again, this property is dependent on h(n) always being admissible. In most circumstances, A* is a heuristic search method that is optimal and complete. (Coppin, 2004)

A* space and time complexity are exponential O(b^d) and for many problems, the space complexity of A* remains a challenge. (Russell & Norvig, 2020)

2.2.3 Iterative Deepening A* (IDA*)

IDA* is an iterative deepening variation of A* that eliminates the space complexity constraint, thus, yielding enhanced performance. Integrating iterative-deepening with A* yields an algorithm that is optimal and complete (like A*) while consuming little memory, as depth-first search achieves. IDA* is a type of iterative-deepening search in which each iteration increases the limit on f(n) rather than the depth of a node. IDA* works well in instances where the heuristic value f(n) has a small number of alternative possibilities.

For example, when applying the Manhattan distance as a heuristic to solve this robot navigation problem, the value of f(n) can only have 1 through 11 potential values. In this instance, the IDA* algorithm requires only 11 iterations and has a time complexity similar to that of A*, but with a substantially enhanced space complexity due to the fact that it is essentially performing depth-first search. (Coppin, 2004)

3 Implementation

This clause describes and explains how each search method has been implemented in the executable program search.exe which has been developed using the programming language Java. Since Java is an object-oriented programming (OOP) language, each search program has been implemented as a distinct class, essentially making them objects in the main program driver.

Each method class has a constructor that takes in an initial start node as its only argument. This constructor builds the new object and stores its vital data in private variables that can later be accessed by getter methods. One such vital data, is the search path taken by the method.

3.1 Depth-First Search (DFS)

The depth-first search implementation is of non-recursive implementation as per the pseudocode shown in figure 6. Also, as shown by the class diagram in figure 5, it utilises a stack data structure which represents a last-in-first-out queue of node objects. However, in this implementation there is no possibility of duplicate objects on the stack as there is a check for cycles. In addition, there is a map data structure which is used to keep track of all paths travel by the frontier, therefore, the final function call in the constructor is to the reconstruct path method. This method accepts two arguments, one is the map data structure, the other is the goal node, finally, it returns a list of nodes which represent the real path taken from start to end.

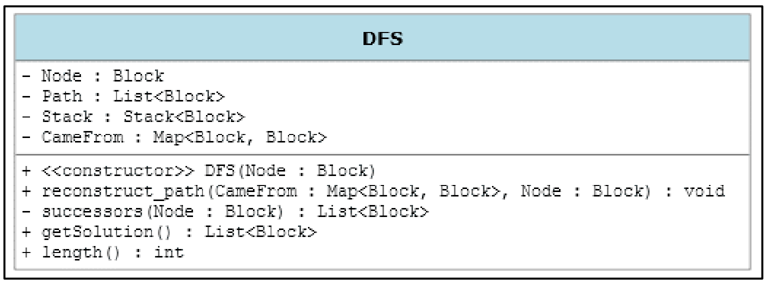

DFS
- Node : Block
- Path : List<Block>
- Stack : Stack<Block>
- CameFrom : Map<Block, Block>
+ <<constructor>> DFS(Node : Block)
+ reconstruct_path(CameFrom : Map<Block, Block>, Node : Block) : void
- successors(Node : Block) : List<Block>
+ getSolution() : List<Block>
+ length() : int

Figure 5. DFS Class Diagram.

Pseudocode

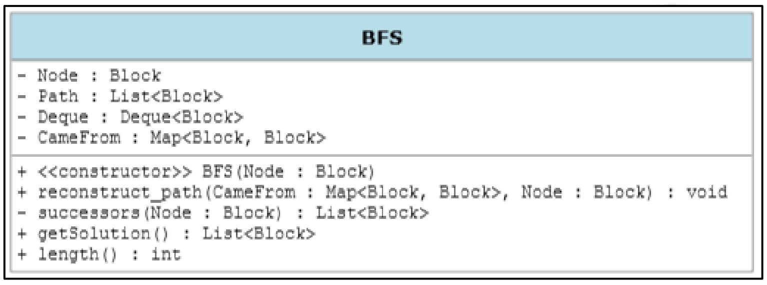
```
procedure DFS(G.v) is
  let cameFrom be map
  let S be stack
  S.push( G.v ) (Inserting G.v in stack)
  while ( S is not empty & G.v in not goal):
    Pop a vertex from stack to visit next
    v  =  S.pop( )
    mark v as visited.
    Push all the neighbours of v in stack that are not visited
    for all neighbours w of v in Graph G:
      if w is not visited :
        S.push( w )
        cameFrom.put( w, v)
      end if
    end for
  end while
  call reconstruct_path to find path taken.
end of DFS procedure

procedure reconstruct_path(cameFrom, goal) is
  let p be list
  let t be temp node
  while current in cameFrom.Keys:
    t := {current}
    p.prepend(current)
    current := cameFrom[current]
    cameFrom.remove[t]
  end while
  add final node.
  p.prepend(current)
  return reverse.p
end of reconstruct_path procedure

procedure successors(v) is
  let r be list of true successors
  let t be list of all successors
```

```
    for all neighbours w of v in Graph G:
      t.add( w )
    end for
    for all v of t
      if v is not a wall
        r.add( v )
      end if
    end for
    return r
  end of successors procedure
```

Figure 6. Pseudocode of DFS Class. (Wikipedia, 2022)

3.2 Breadth-First Search (BFS)

BFS is implemented as a procedural constructor, like DFS, however, it uses a first-in-first-out (FIFO) queue. Therefore, BFS pops the node that appears first in the queue. Modifications to the original pseudocode have been add which allow path retrieval. Using a map, the reconstruct path method back-tracks from the goal node to the start node and adds the path taken to path list, which can then be retrieved using the get solution method.



```
BFS
- Node : Block
- Path : List<Block>
- Deque : Deque<Block>
- CameFrom : Map<Block, Block>
+ <<constructor>> BFS(Node : Block)
+ reconstruct_path(CameFrom : Map<Block, Block>, Node : Block) : void
- successors(Node : Block) : List<Block>
+ getSolution() : List<Block>
+ length() : int
```

Figure 7. BFS Class Diagram.

Pseudocode

```
procedure BFS(G.v) is
    let cameFrom be map
    let Q be a queue
    label root as explored
    Q.enqueue(root)
    while Q is not empty do
      v := Q.dequeue()
      if v is the goal then
        return reconstruct_path()
      end if
      for all edges from v to w in G.adjacentEdges(v) do
        if w is not labeled as explored then
          label w as explored
          Q.enqueue(w)
          cameFrom.put( w, v)
        end if
       end for
    end while
end of BFS procedure

procedure reconstruct_path(cameFrom, goal) is
  let p be list
  let t be temp node
  while current in cameFrom.Keys:
    t := {current}
    p.prepend(current)
    current := cameFrom[current]
    cameFrom.remove[t]
  end while
  add final node.
  p.prepend(current)
  return reverse.p
end of reconstruct_path procedure

procedure successors(v) is
  let r be list of true successors
  let t be list of all successors
  for all neighbours w of v in Graph G:
    t.add( w )
  end for
  for all v of t
    if v is not a wall
      r.add( v )
    end if
  end for
  return r
end of successors procedure
```

Figure 8. Pseudocode of BFS Class. (Wikipedia, 2022)

3.3 Iterative Deepening Depth-First Search (IDDFS)

The IDDFS Class utilises a stack data structure to maintain a record of the nodes in the final path. The main procedure of IDDFS is implemented in the IDDFS Class constructor, which enters a loop starting from zero to infinite, where zero to infinite represents the depth limit. In this loop the current depth and node is passed by argument to the depth-limited-search (DLS) which returns a custom-built object of ResponseObject. The reason there is a need for this new return object, is that it returns two different objects, the current node, and a Boolean value.

The method DLS contains a recursive call, with depth as the recursion check. If depth is zero it stops the recursion, as the depth in the constructor loop grows larger the recursion call deepens, essentially, the recursion works backwards as the loop works forwards.

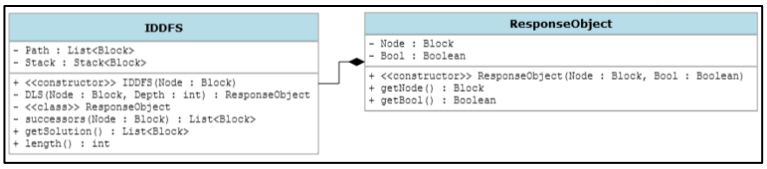

IDDFS
- Path : List<Block>
- Stack : Stack<Block>
+ <<constructor>> IDDFS(Node : Block)
- DLS(Node : Block, Depth : int) : ResponseObject
- <<class>> ResponseObject
- successors(Node : Block) : List<Block>
+ getSolution() : List<Block>
+ length() : int
ResponseObject
- Node : Block
- Bool : Boolean
+ <<constructor>> ResponseObject(Node : Block, Bool : Boolean)
+ getNode() : Block
+ getBool() : Boolean

Figure 9. IDDFS Class Diagram.

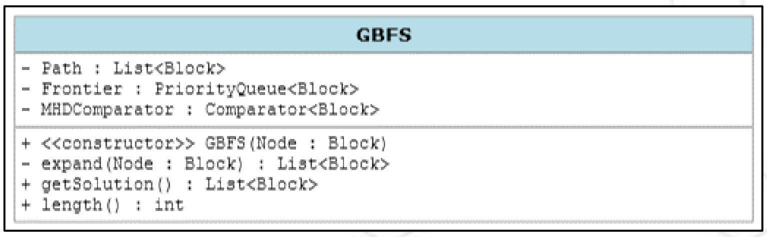
Pseudocode

```
procedure IDDFS(root) is
  let S be stack
  for depth from 0 to infinite do
    found, remaining = DLS(root, depth)
    if found != null then
      return found
    else if not remaining then
      return null
  end for
  S.push( node )
end of IDDFS procedure

procedure DLS(node, depth) is
  if depth == 0 then
    if node is a goal then
      return (node, true)
    else
      return (null, true)     (Not found, but may have children)
  else if depth > 0 then
    any_remaining = false
    for each child of node do
      found, remaining = DLS(child, depth-1)
        if found != null then
          S.add( child )
          return (found, true)
        end if
    if remaining then
(At least one node found at depth, let IDDFS deepen)
      any_remaining = true
    end for
    return (null, any_remaining)
  end else if
end of DLS procedure

procedure successors(v) is
  let r be list of true successors
  let t be list of all successors
  for all neighbours w of v in Graph G:
    t.add( w )
  end for
  for all v of t
    if v is not a wall
      r.add( v )
    end if
  end for
  return r
end of successors procedure
```

Figure 10. Pseudocode of IDDFS Class. (Wikipedia, 2021)

3.4 Greedy Best-First Search (GBFS)

The implementation of GBFS requires a priority queue, the queue is sorted by the h(n) value of each node. To do this, the queue must be instantiated with a comparator passed in as an argument, therefore, it requires a custom comparator which compares the heuristic h(n), in this case that represents the Manhattan distance to the closest goal. As a result, the search algorithm will follow the first path it finds and believes to be the shortest path to the goal, even if it is not.



```
GBFS
- Path : List<Block>
- Frontier : PriorityQueue<Block>
- MHDComparator : Comparator<Block>
+ <<constructor>> GBFS(Node : Block)
- expand(Node : Block) : List<Block>
+ getSolution() : List<Block>
+ length() : int
```

Figure 11. GBFS Class Diagram.

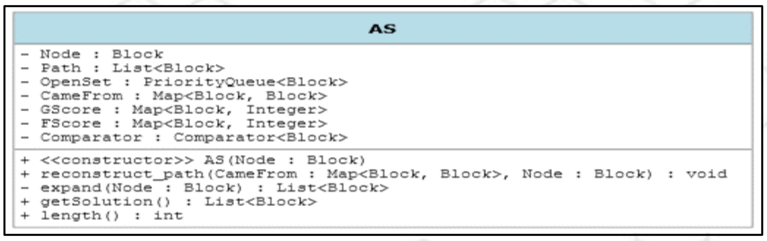
Pseudocode

```
procedure GBS(start, target) is
  let P be list of path nodes
  let F be priority queue
  mark start as visited
  add start to queue
  while queue is not empty do:
    current_node = vertex of queue with min distance to target
    remove current_node from queue
    add current_node to P
    foreach neighbor n of current_node do:
      if n not in visited then
        mark n as visited
        add n to queue
      end if
    end for
  end while
end of GBFS procedure

procedure expand(v) is
  let r be list of true successors
  let t be list of all successors
  for all neighbours w of v in Graph G:
    t.add( w )
  end for
  for all v of t
    if v is not a wall
      r.add( v )
    end if
  end for
  return r
end of expand procedure
```

Figure 12. Pseudocode of GBFS Class. (Wikipedia, 2022)

3.5 A*

The implementation of A* has seven private fields as shown in figure 13, like GBFS, a priority queue is required to sort nodes. However, as outlined in clause 2.2.2, the evaluation function f(n) will equate to f(n)=g(n)+h(n), as a result, the comparator passed to the queue will evaluate each node based on the Manhattan distance to the closest goal plus the path cost thus far. In addition, there are three maps, came from can be used to reconstruct the path as describe in clause 3.2, the two additional maps keep track of the current g(n) and h(n) values for each node in the tree. The reason being, if there are two or more instances of one node in the tree, they will have different g(n) and h(n) values. This is important as it makes it possible to find the correct cheapest path.



```
AS
- Node : Block
- Path : List<Block>
- OpenSet : PriorityQueue<Block>
- CameFrom : Map<Block, Block>
- GScore : Map<Block, Integer>
- FScore : Map<Block, Integer>
- Comparator : Comparator<Block>
+ <<constructor>> AS(Node : Block)
+ reconstruct_path(CameFrom : Map<Block, Block>, Node : Block) : void
- expand(Node : Block) : List<Block>
+ getSolution() : List<Block>
+ length() : int
```

Figure 13. A Class Diagram.*

Pseudo Code

```
procedure AStar(start) is
  openSet := {start}
  cameFrom := an empty map
  gScore := map with default value of Infinity
  gScore[start] := 0
  fScore := map with default value of Infinity
  fScore[start] := h(start)
  while openSet is not empty
    current := the node in openSet having the lowest f(n) value
    if current = goal
      return reconstruct_path(CameFrom, Current)
    end if
    openSet.Remove(current)
    for each neighbor of current
      test_gScore := gScore[current] + cost(current, neighbor)
      if test_gScore < gScore[neighbor]
This path to neighbor is better than any previous one. Record it!
        cameFrom[neighbor] := current
        gScore[neighbor] := test_gScore
        fScore[neighbor] := test_gScore + h(neighbor)
        if neighbor not in openSet
          openSet.add(neighbor)
        end if
    end for
  end while
  return failure
end procedure

procedure reconstruct_path(cameFrom, goal) is
  let p be list
  let t be temp node
  while current in cameFrom.Keys:
    t := {current}
    p.prepend(current)
    current := cameFrom[current]
    cameFrom.remove[t]
  end while
  add final node.
  p.prepend(current)
  return reverse.p
end of AS procedure

procedure expand(v) is
  let r be list of true successors
  let t be list of all successors
  for all neighbours w of v in Graph G:
    t.add( w )
  end for
  for all v of t
    if v is not a wall
      r.add( v )
    end if
  end for
  return r
end of expand procedure
```

Figure 14. Pseudocode of A Class. (Wikipedia, 2022)*

3.6 Iterative Deepening A* (IDA*)

Like IDDFS discussed in clause 2.1.3 and the implementation of it discussed in clause 3.3, the implementation of IDA* is a recursive call the search method. The search method takes in two arguments, both of which are related to the start node, g(n), and h(n). As seen in the clause 3.3, the procedure starts with a loop from zero to infinite, however, IDA* can only have max iterations of the start nodes h(n) value passed into the search method as the bound. Hence, the recursive call stops and returns the value of f(n) if it is larger than the bound as there are better paths. Finally, the method will return minus one when it reaches the goal, and the procedures will terminate. The path to the goal will be stored in the one private field which is accessible through the getter called get solution.

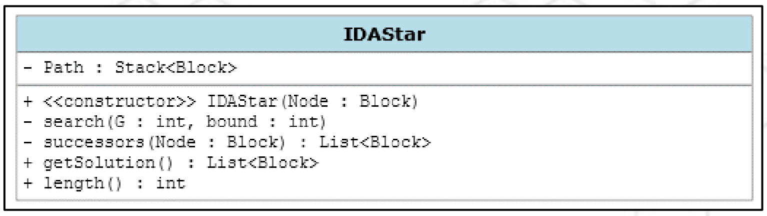

IDAStar
- Path : Stack<Block>
+ <<constructor>> IDAStar(Node : Block)
- search(G : int, bound : int)
- successors(Node : Block) : List<Block>
+ getSolution() : List<Block>
+ length() : int

Figure 15. IDAStar Class Diagram.

Pseudocode

```
procedure IDAStar(root)
    bound := h(root)
    path := [root]
    loop
        t := search(0, bound)
        if t = FOUND then return (path, bound)
        if t = ∞ then return NOT_FOUND
        bound := t
    end loop
end of IDAStar procedure

procedure search(g, bound)
    node := path.last
    f := g + h(node)
    if f > bound then return f
    if is_goal(node) then return FOUND
    min := ∞
    for succ in successors(node) do
        if succ not in path then
            path.push(succ)
            t := search(g + cost(node, succ), bound)
            if t = FOUND then return FOUND
            if t < min then min := t
            path.pop()
        end if
    end for
    return min
end search of procedure

procedure successors(v) is
  let r be list of true successors
  let t be list of all successors
  for all neighbours w of v in Graph G:
    t.add( w )
  end for
  for all v of t
    if v is not a wall
      r.add( v )
    end if
  end for
  return r
end of successors procedure
```

Figure 16. Pseudocode of IDA Class. (Wikipedia, 2021)*

4 FEATURES/BUGS/MISSING

4.1 FEATURES

4.1.1 Classes

- Search (Main driver).
- Block (Maze cell).
- Vector (Block coordinates).
- <<Enum>> Status (Block status, e.g., start, path, wall, goal).
- Neighbour (Maps maze cell to its neighbouring cells, e.g., north, east, south, west).
- Maze (Maze itself).

4.1.2 Methods

- getDirections(list of nodes) returns list of strings as directions.
- setWallStates(BufferedReader aFileData) adds all walls to maze, including outer boundary.
- setGoalStates(BufferedReader aFileData) adds all goals to maze.
- setInitialState(BufferedReader aFileData) adds start location to maze.
- generateMaze() adds paths to maze.
- setMazeDimensions(BufferedReader aFileData) sets maze dimensions.
- buildNeighbours() adds all neighbouring cells to every cell in the maze.
- buildDistance() sets the Manhattan distance to closest goal for all cells in the maze.
- getStart() finds the starting cell in the maze.

4.2 BUGS

- If there are no goal states specified in the maze data supplied to the program, it will detect and throw an exception while terminating the program. As opposed to, the program continuing to run and the search methods failing to find a goal state.

4.3 MISSING

- No required feature has been un-implemented.

5 CONCLUSION

In this report, I have documented how I have implemented tree-based search algorithms in software using Java (from scratch). Furthermore, I investigated six path-finding algorithms to search for a solution to the Robot Navigation problem. Both informed and uninformed methods have been implemented.

The wide variety of algorithms tested all had strengths and weaknesses. Thus, concludes that when planning which path-finding algorithm is the best type of algorithm to implement for solving the Robot Navigation problem, or any path-finding problem, one must first determine what to prioritize. If path optimality is of importance, one might choose to go with the A* search algorithm. If the map has a small maze structure, therefore, space constraints are not an issue, one might prefer to choose a grid-based search algorithm like BFS, which can handle brute-forcing the solution. Or, if the problem does not fit into memory and path optimality is of importance, one might prefer to choose IDA* which at a cost of visiting some states on numerous occasions, it gives the benefits of A* minus the overhead of storing in memory all reached states, therefore providing optimal performance.

On top of every, I learnt and taught myself several new search methods not covered in the lectures.

6 Acknowledgements/Resources

All pseudocode for every search method was sourced from Wikipedia, as listed in clause 8 of this document. The pseudocode from Wikipedia assisted as a great guide on how to code all the search methods outlined in clause 3. However, all instances of the pseudocode required modification to achieve the desired result of my program. In addition, cross-modification is also present, where pseudocode from one search algorithm may have been used to help modify another search algorithm.

7 Glossary

7.1 Definitions

The following definitions establish meaning in the text of this document. For other definitions of terms not found in this clause, consult *ISO/IEC 2382:2015*, or latest revision thereof.

artificial intelligence: 1) branch of computer science devoted to developing data processing systems that perform functions normally associated with human intelligence, such as reasoning, learning, and self-improvement. (ISO/IEC JTC 1, 2015)

7.2 Acronyms

The following acronyms appear within the text of this report:

ADS	ABSTRACT DATA STRUCTURE
AI	ARTIFICIAL INTELLIGENCE
BFS	BREADTH-FIRST SEARCH
DFS	DEPTH-FIRST SEARCH
GBFS	GREEDY BEST-FIRST SEARCH
IA	INTELLIGENT AGENT
IDA*	ITERATIVE DEEPENING A*
IDDFS	ITERATIVE DEEPENING DEPTH-FIRST SEARCH
LIFO	LAST-IN-FIRST-OUT
OOP	OBJECT-ORIENTED PROGRAMMING

8 References

Coppin, B., 2004. *Artificial Intelligence Illuminated.* 1st ed. Sudbury: Jones and Bartlett Publishers.

ISO/IEC JTC 1, 2015. ISO/IEC 2382:2015(en). *ISO/IEC 2382:2015,* Issue 1, pp. 1-4.

Russell, S. J. & Norvig, P., 2020. *Artificial Intelligence, A Modern Approach.* Fourth ed. Hoboken: Prentice Hall.

Wikipedia, 2021. *Iterative Deepening A*.* [Online]
Available at: https://en.wikipedia.org/wiki/Iterative_deepening_A*
[Accessed 29 April 2022].

Wikipedia, 2021. *Iterative Deepening Depth-First Search.* [Online]
Available at: https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search
[Accessed 29 April 2022].

Wikipedia, 2022. *A* Search Algorithm.* [Online]
Available at: https://en.wikipedia.org/wiki/A*_search_algorithm
[Accessed 29 April 2022].

Wikipedia, 2022. *Best-First Search.* [Online]
Available at: https://en.wikipedia.org/wiki/Best-first_search
[Accessed 29 April 2022].

Wikipedia, 2022. *Breadth-First Search.* [Online]
Available at: https://en.wikipedia.org/wiki/Breadth-first_search
[Accessed 28 April 2022].

Wikipedia, 2022. *Depth-First Search.* [Online]
Available at: https://en.wikipedia.org/wiki/Depth-first_search
[Accessed 28 April 2022].