Student Declaration of Authorship

HERIOT WATT UNIVERSITY
UK | DUBAI | MALAYSIA

Course code and name:	**F29AI - Artificial Intelligence and Intelligent Agents**
Type of assessment:	**Group**
Coursework Title:	**Coursework 1: A* Search and Automated Planning**
Student Name:	**SYED ARIF ALI**
Student ID Number:	**H00484788**

Declaration of authorship. By signing this form:

- **I declare** that the work I have submitted for individual assessment OR the work I have contributed to a group assessment, is entirely my own. I have NOT taken the ideas, writings or inventions of another person and used these as if they were my own. My submission or my contribution to a group submission is expressed in my own words. Any uses made within this work of the ideas, writings or inventions of others, or of any existing sources of information (books, journals, websites, etc.) are properly acknowledged and listed in the references and/or acknowledgements section.
- I confirm that I have read, understood and followed the University's Regulations on plagiarism as published on the University's website, and that I am aware of the penalties that I will face should I not adhere to the University Regulations.
- I confirm that I have read, understood and avoided the different types of plagiarism explained in the University guidance on Academic Integrity and Plagiarism

Student Signature (type your name): SYED ARIF ALI **Date**: 24/10/2024

Student Declaration of Authorship

HERIOT WATT UNIVERSITY
UK | DUBAI | MALAYSIA

Course code and name:	**F29AI - Artificial Intelligence and Intelligent Agents**
Type of assessment:	**Group**
Coursework Title:	**Coursework 1: A* Search and Automated Planning**
Student Name:	**FAISAL CHAUDHRY**
Student ID Number:	**H00490254**

Declaration of authorship. By signing this form:

- **I declare** that the work I have submitted for individual assessment OR the work I have contributed to a group assessment, is entirely my own. I have NOT taken the ideas, writings or inventions of another person and used these as if they were my own. My submission or my contribution to a group submission is expressed in my own words. Any uses made within this work of the ideas, writings or inventions of others, or of any existing sources of information (books, journals, websites, etc.) are properly acknowledged and listed in the references and/or acknowledgements section.
- I confirm that I have read, understood and followed the University's Regulations on plagiarism as published on the University's website, and that I am aware of the penalties that I will face should I not adhere to the University Regulations.
- I confirm that I have read, understood and avoided the different types of plagiarism explained in the University guidance on Academic Integrity and Plagiarism

Student Signature (type your name): FAISAL CHAUDHRY **Date**: 24/10/2024

A* Search and Automated Planning

F29AI - Artificial Intelligence and Intelligent Agents

Syed Arif Ali and Faisal Chaudhry (Dubai PG Group-3)

F29AI - Artificial Intelligence and Intelligent Agents

Coursework 1 - Postgraduate

A* Search and Automated Planning

Part 1: A* Search

Part 1A - Manual Calculations

The 2 grids below (Fig.1 and Fig.2) represent two problem environments where an agent is trying to find a path from the start location (S) to the goal location (G). The agent can move to an adjacent square provided the square is white or grey. The agent cannot move to a black square which represents a wall. No diagonal movement is allowed. Moving into a white square has a cost of 1. Moving into a grey square has a cost of 2. We assume that ties on the fringe are broken by alphabetical ordering of the nodes.

The heuristic function used is the Manhattan function (Black, 2006) which can be represented by the formula :-

- 2-Dimensional Manhattan Distance $(\boldsymbol{L_m}) = |\boldsymbol{x_{goal}} - \boldsymbol{x_{initial}}| + |\boldsymbol{y_{goal}} - \boldsymbol{y_{initial}}|$
- 3-Dimensional Manhattan Distance $(\boldsymbol{L_m}) = |\boldsymbol{x_{goal}} - \boldsymbol{x_{initial}}| + |\boldsymbol{y_{goal}} - \boldsymbol{y_{initial}}| + |\boldsymbol{z_{goal}} - \boldsymbol{z_{initial}}|$

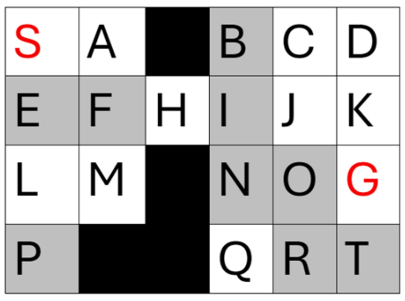
Steps for manual calculation

To perform the manual calculations for the A* search on both grids, we need to go step by step and calculate the **g-value** (cost from the start node), **h-value** (Manhattan distance heuristic), and **f-value** (g + h) for each node expanded during the search process.

The step-by-step breakdown of the procedure is:

1. **Find G-value**: The cumulative cost from the start node to the current node (incremented by the transition cost for each move). Here the transition costs are either 1(white tiles) or 2(black tiles).
2. **Find H-value** (Manhattan distance): The absolute difference between the goal node's coordinates and the current node's coordinates: H=| $x_{goal} - x_{current}$ |+ | $y_{goal} - y_{current}$ |
3. **Calculate F-value**: The sum of the g-value and the h-value: F = G + H
4. **Determine Expansion**: Using the calculated F-value we determine which of the Children Node states is expanded from the Parent Node states . Expansion is done for the state with the lowest F-value.

For Grid 1



S	A	■	B	C	D
E	F	H	I	J	K
L	M	■	N	O	G
P	■	■	Q	R	T

Fringe before expansion	Node Expanded	Neighbors of Node Expanded	Heuristic at that node (H)	Cost (T)	F (node) = H + T	Fringe After	Closed List (C)
	Initial node: S		**7**	**0**	**7**	**S(7)**	
S(7)	S	A E	6 6	1 2	7 8	A(7), E(8)	S
A(7), E(8)	A	F	5	3	8	E(8), F(8)	S, A,
E(8), F(8)	E *	F L	-- 5	-- 3	-- 8	F(8), L(8)	S, A, E
F(8), L(8)	F *	H M	4 4	4 4	8 8	H(8), L(8), M(8)	S, A, E, F
H(8), L(8)	H *	I	3	6	9	L(8), M(8), I(9)	S, A, E, F, H
L(8), M(8), I(9)	L	E P M	- 6 -	- 5 -	- 11 -	M(8), I(9), P(11)	S, A, E, F, H, L
M(8), I(9), P(11)	M	F L	- -	-	-	I(9), P(11)	S, A, E, F, H, L, M
I(9), P(11)	I	B J N	4 2 2	8 7 8	12 9 10	J(9), N(10), P(11), B(12)	S, A, E, F, H, L, M, I
J(9), N(10), P(11), B(12)	J	C K O	3 1 1	8 8 9	11 9 10	K(9),N(10), O(10), C(11), P(11), B(12)	S, A, E, F, H, L, M, I, J
K(9), N(10), O(10), C(11), P(11), B(12)	K	D G	2 0	9 9	11 9	G(9), N(10), O(10), C(11), D(11), P(11), B(12)	S, A, E, F, H, L, M, I, J, K

Fig.1 - (Grid 1 and its associated A* calculation table)

Heuristic calculations for Grid 1

Node	Coordinates (x, y)	H-value (Manhattan Distance to Goal (2, 5)
Start	(0, 0)	|2 - 0|+|5 - 0|= 7
A	(0, 1)	|2 - 0|+|5 - 1|= 6
B	(0, 3)	|2 - 0|+|5 - 3|= 4
C	(0, 4)	|2 - 0|+|5 - 4|= 3
D	(0, 5)	|2 - 0|+|5 - 5|= 2
E	(1, 0)	|2 - 1|+|5 - 0|= 6
F	(1, 1)	|2 - 1|+|5 - 1|= 5
H	(1, 2)	|2 - 1|+|5 - 2|= 4
I	(1, 3)	|2 - 1|+|5 - 3|= 3
J	(1, 4)	|2 - 1|+|5 - 4|= 2
K	(1, 5)	|2 - 1|+|5 - 5|= 1
L	(2, 0)	|2 - 2|+|5 - 0|= 5
M	(2, 1)	|2 - 2|+|5 - 1|= 4
N	(2, 3)	|2 - 2|+|5 - 3|= 2
O	(2, 4)	|2 - 2|+|5 - 4|= 1
P	(3, 0)	|2 - 3|+|5 - 0|= 6
Q	(3, 3)	|2 - 3|+|5 - 3|= 3
R	(3, 4)	|2 - 3|+|5 - 4|= 2
T	(3, 5)	|2 - 3|+|5 - 5|= 1
Goal	(2, 5)	|2 - 2|+|5 - 5|= 0

Summary of manual calculations for Grid 1

- **States expanded:**
  Start → A → E → F → H→L→M→ I → J → K → Goal.

- **Goal path:**
  Start → A → F → H → I → J → K → Goal.

- **Total Cost: 9**

For Grid 2

S	A	■	B	C
D	E	F	H	I
J	K	■	L	M
N	O	P	Q	R
T	■	■	G	U

Fringe before expansion	Node Expanded	Neighbors of Node Expanded	Heuristic at that node (H)	Cost (T)	F (node) = H + T	Fringe After	Closed List (C)
	Initial node: S		7	0	7	S(7)	
S(7)	S	A D	6 6	1 2	7 8	A(7), D(8)	S
A(7), D(8)	A	E	5	2	7	E(7), D(8)	S, A
E(7), D(8)	E	F K	4 4	3 3	7 7	F(7), K(7), D(8)	S, A, E
F(7), K(7) D(8)	F	H	3	5	8	H(8), K(7), D(8)	S, A, E, F

H(8), K(7), D(8)	K	O
		J
			3
			5
				4
				4
					O(7)
					J(9)
						H(8), D(8), O(7), J(9)	S, A, E, F, K

H(8), D(8), O(7), J(9)	O	P
		N
			2
			4
				6
				6
					P(8)
					N(10)
						H(8), D(8), J(9), P(8), N(10)	S, A, E, F, K, O

H(8), D(8), J(9), P(8), N(10)	D	E J	- -			H(8), J(9), P(8), N(10)	S, A, E, F, K, O, D

H(8), J(9), P(8), N(10)	H	B
		I
		L
			4
			4
			2
				7
				6
				6
					B(11)
					I(10)
					L(8)
						J(9), P(8), N(10), B(11), I(10), L(8)	S, A, E, F, K, O, D, H

J(9), P(8), N(10), B(11), I(10), L(8)	L	Q
		M
			1
			3
				7
				7
					Q(8)
					M(10)
						J(9), P(8), N(10), B(11), I(10), Q(8), M(10)	S, A, E, F, K, O, D, H, L

J(9), P(8), N(10), B(11), I(10), Q(8), M(10)	P	Q	-			J(9), N(10), B(11), I(10), Q(8), M(10)	S, A, E, F, K, O, D, H, L, P

J(9), N(10), B(11), I(10), Q(8), M(10)	Q	G
		R
			0
			2
				8
				9
					G(8)
					R(11)
							S, A, E, F, K, O, D, H, L, P, Q, G

Fig.2 - (Grid 2 and its associated A* calculation table)

Heuristic calculations for Grid 2

Node	Coordinates (x, y)	H-value (Manhattan Distance to Goal (4, 3))
Start	(0, 0)	|4 - 0|+|3 - 0|= 7
A	(0, 1)	|4 - 0|+|3 - 1|= 6
B	(0, 3)	|4 - 0|+|3 - 3|= 4
C	(0, 4)	|4 - 0|+|3 - 4|= 5
D	(1, 0)	|4 - 1|+|3 - 0|= 6
E	(1, 1)	|4 - 1|+|3 - 1|= 5
F	(1, 2)	|4 - 1|+|3 - 2|= 4
H	(1, 3)	|4 - 1|+|3 - 3|= 3
I	(1, 4)	|4 - 1|+|3 - 4|= 4
J	(2, 0)	|4 - 2|+|3 - 0|= 5
K	(2, 1)	|4 - 2|+|3 - 1|= 4
L	(2, 3)	|4 - 2|+|3 - 3|= 2
M	(2, 4)	|4 - 2|+|3 - 4|= 3
N	(3, 0)	|4 - 3|+|3 - 0|= 4
O	(3, 1)	|4 - 3|+|3 - 1|= 3
P	(3, 2)	|4 - 3|+|3 - 2|= 2
Q	(3, 3)	|4 - 3|+|3 - 3|= 1
R	(3, 4)	|4 - 3|+|3 - 4|= 2
T	(4, 0)	|4 - 4|+|3 - 0|= 3
U	(4, 4)	|4 - 4|+|3 - 4|= 1
Goal	(4, 3)	|4 - 0|+|3 - 0|= 0

Summary of manual calculations for Grid 2

- **States expanded:**

  Start → A → E → F → K→ O→ D → H →L→ P→ Q → Goal

- **Goal path:**

  - **(Path 1 - Start → A → E → F → H → L → Q → Goal) or**
  - **(Path 2 = Start → A → E → K → O → P → Q → Goal)**

  Path 1 is obtained using the manual calculation and the ties on the fringe are broken by alphabetical ordering of the nodes as shown in Figure 2.

  Path 2 is obtained when not account for alphabetic precedence in case of a tie on the fringe

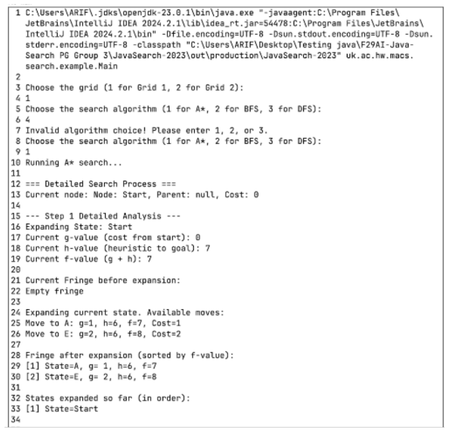
- **Total Cost:** **(Path 1 = 8) or (Path 2 = 8)**

Part 1B - Java Coding

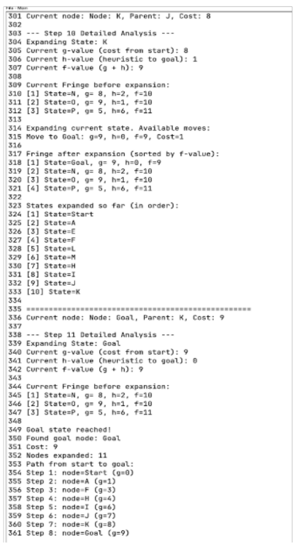
For this section we are presenting a basic code comparison between our implementations for A* search and the starter code (Sierra and Bates, 2005) that was provided. Here is a breakdown of the main differences:

1. **AStarSearchOrder Implementation:** In the Java Starter code provided , the AStarSearchOrder class is absent and needs to be implemented whereas in our Search Algorithm implementation the class has been fully implemented. Its main function is to sort the frontier by the f-values (g + h) after adding the children. It adds children to the fringe and sorts the frontier based on F-values (G + H). This enables A* search to prioritize nodes with the lowest estimated cost to the goal.
2. **BreadthFirstSearchOrder and DepthFirstSearchOrder Classes:** Both classes are present in both files and are similarly implemented. However, in our implementation we have allowed the user to choose between algorithms dynamically during runtime.
3. **Enhanced Output in SearchProblem:** In our Search Algorithm implementation the SearchProblem class has been given additional output features, like detailed steps during the search process. For instance, it outputs the current node, g-value, h-value, and f-value. The fringe is printed before and after expansion, and a more detailed analysis is provided during each step. Whereas in the Java Starter code, these detailed print statements and step-by-step breakdowns are absent. The search process in Java Starter is simpler and only prints the current node and the goal state if found.
4. **Grids in Main.java:** Our Search Algorithm implementation includes two different grid structures (createGrid1 and createGrid2) for use in the search. It allows the user to select which grid to run searches on. In Java Starter code provided , the Main.java() file contains only a basic example with a small number of states. The nodes and transitions are hard-coded, with no choice of different grids.
5. **Interactive User Input:** Our Search Algorithm implementation prompts the user to select between different grids and search algorithms (A*, BFS, DFS) and provides error handling for invalid inputs. The Java Starter code does not include user input options for selecting grids or search algorithms; it only uses BFS as the search order.
6. **Expanded FringeNode toString Output:** In our implementation the FringeNode class has a more detailed toString method, which prints the node label, its parent, and the cost this helps in debugging and understanding the search progress. In the Java Starter Code, the toString method for FringeNode is simpler and does not provide as much information.
7. **Final goal path and Cost Calculation**

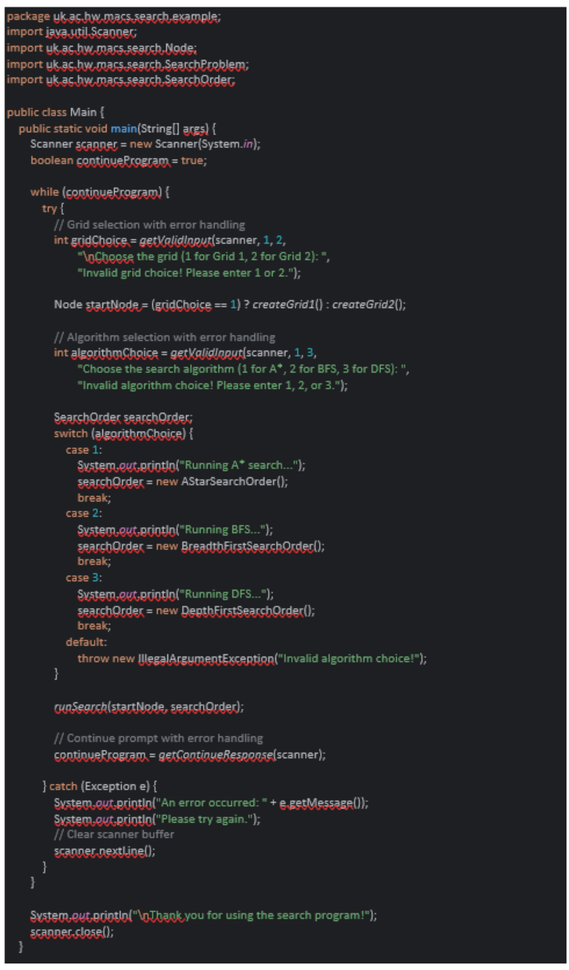
Code Snippets



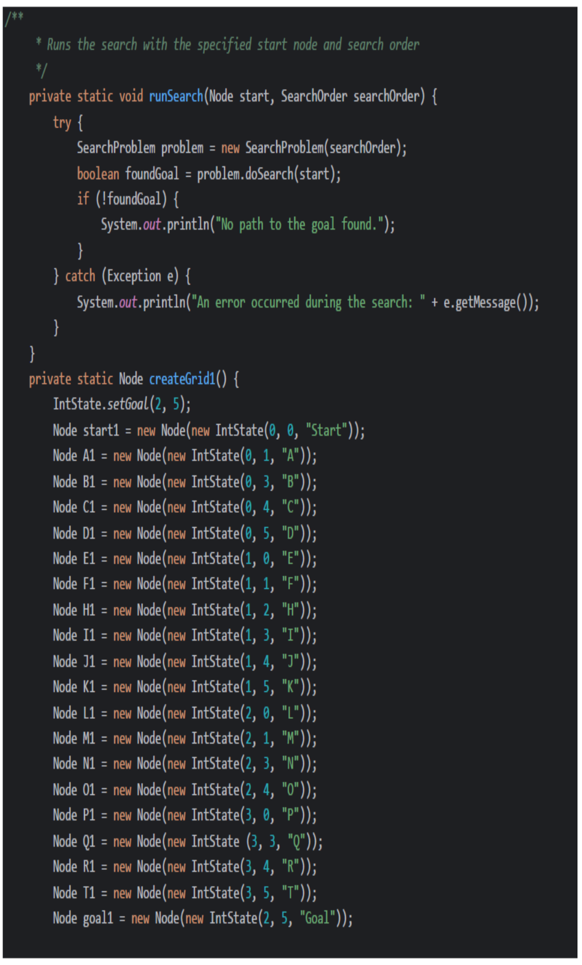
```
C:\Users\ARIF\.jdks\openjdk-23.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.2.1\lib\idea_rt.jar=54478:C:\Program Files\JetBrains\IntelliJ IDEA 2024.2.1\bin" -Dfile.encoding=UTF-8 -Dsun.stdout.encoding=UTF-8 -Dsun.stderr.encoding=UTF-8 -classpath "C:\Users\ARIF\Desktop\Testing java\F29AI-Java-Search PG Group 3\JavaSearch-2023\out\production\JavaSearch-2023" uk.ac.hw.macs.search.example.Main

Choose the grid (1 for Grid 1, 2 for Grid 2):
1
Choose the search algorithm (1 for A*, 2 for BFS, 3 for DFS):
4
Invalid algorithm choice! Please enter 1, 2, or 3.
Choose the search algorithm (1 for A*, 2 for BFS, 3 for DFS):
1
Running A* search...

=== Detailed Search Process ===
Current node: Node: Start, Parent: null, Cost: 0

--- Step 1 Detailed Analysis ---
Expanding State: Start
Current g-value (cost from start): 0
Current h-value (heuristic to goal): 7
Current f-value (g + h): 7

Current Fringe before expansion:
Empty fringe

Expanding current state. Available moves:
Move to A: g=1, h=6, f=7, Cost=1
Move to E: g=2, h=6, f=8, Cost=2

Fringe after expansion (sorted by f-value):
[1] State=A, g= 1, h=6, f=7
[2] State=E, g= 2, h=6, f=8

States expanded so far (in order):
[1] State=Start

```



```
Current node: Node: K, Parent: J, Cost: 8

--- Step 10 Detailed Analysis ---
Expanding State: K
Current g-value (cost from start): 8
Current h-value (heuristic to goal): 1
Current f-value (g + h): 9

Current Fringe before expansion:
[1] State=N, g= 8, h=2, f=10
[2] State=O, g= 9, h=1, f=10
[3] State=P, g= 5, h=6, f=11

Expanding current state. Available moves:
Move to Goal: g=9, h=0, f=9, Cost=1

Fringe after expansion (sorted by f-value):
[1] State=Goal, g= 9, h=0, f=9
[2] State=N, g= 8, h=2, f=10
[3] State=O, g= 9, h=1, f=10
[4] State=P, g= 5, h=6, f=11

States expanded so far (in order):
[1] State=Start
[2] State=A
[3] State=E
[4] State=F
[5] State=L
[6] State=M
[7] State=H
[8] State=I
[9] State=J
[10] State=K

==================================================
Current node: Node: Goal, Parent: K, Cost: 9

--- Step 11 Detailed Analysis ---
Expanding State: Goal
Current g-value (cost from start): 9
Current h-value (heuristic to goal): 0
Current f-value (g + h): 9

Current Fringe before expansion:
[1] State=N, g= 8, h=2, f=10
[2] State=O, g= 9, h=1, f=10
[3] State=P, g= 5, h=6, f=11

Goal state reached!
Found goal node: Goal
Cost: 9
Nodes expanded: 11
Path from start to goal:
Step 1: node=Start (g=0)
Step 2: node=A (g=1)
Step 3: node=F (g=3)
Step 4: node=H (g=4)
Step 5: node=I (g=6)
Step 6: node=J (g=7)
Step 7: node=K (g=8)
Step 8: node=Goal (g=9)
```

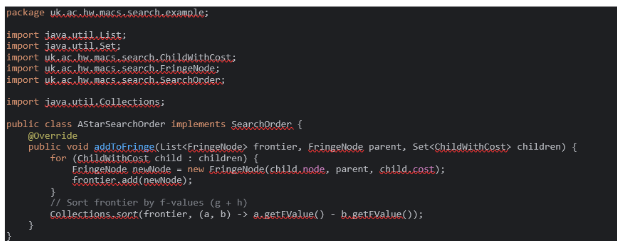


```java
package uk.ac.hw.macs.search.example;
import java.util.Scanner;
import uk.ac.hw.macs.search.Node;
import uk.ac.hw.macs.search.SearchProblem;
import uk.ac.hw.macs.search.SearchOrder;

public class Main {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    boolean continueProgram = true;

    while (continueProgram) {
      try {
        // Grid selection with error handling
        int gridChoice = getValidInput(scanner, 1, 2,
            "\nChoose the grid (1 for Grid 1, 2 for Grid 2): ",
            "Invalid grid choice! Please enter 1 or 2.");

        Node startNode = (gridChoice == 1) ? createGrid1() : createGrid2();

        // Algorithm selection with error handling
        int algorithmChoice = getValidInput(scanner, 1, 3,
            "Choose the search algorithm (1 for A*, 2 for BFS, 3 for DFS): ",
            "Invalid algorithm choice! Please enter 1, 2, or 3.");

        SearchOrder searchOrder;
        switch (algorithmChoice) {
          case 1:
            System.out.println("Running A* search...");
            searchOrder = new AStarSearchOrder();
            break;
          case 2:
            System.out.println("Running BFS...");
            searchOrder = new BreadthFirstSearchOrder();
            break;
          case 3:
            System.out.println("Running DFS...");
            searchOrder = new DepthFirstSearchOrder();
            break;
          default:
            throw new IllegalArgumentException("Invalid algorithm choice!");
        }

        runSearch(startNode, searchOrder);

        // Continue prompt with error handling
        continueProgram = getContinueResponse(scanner);

      } catch (Exception e) {
        System.out.println("An error occurred: " + e.getMessage());
        System.out.println("Please try again.");
        // Clear scanner buffer
        scanner.nextLine();
      }
    }

    System.out.println("\nThank you for using the search program!");
    scanner.close();
  }
```

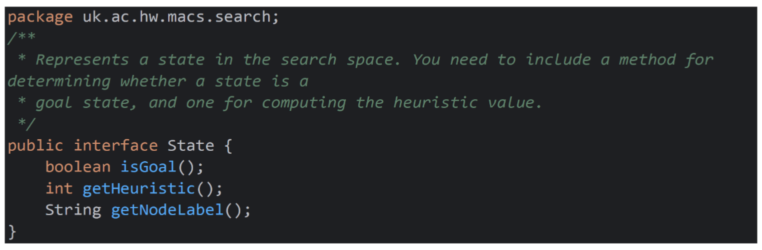


```java
/**
     * Runs the search with the specified start node and search order
     */
    private static void runSearch(Node start, SearchOrder searchOrder) {
        try {
            SearchProblem problem = new SearchProblem(searchOrder);
            boolean foundGoal = problem.doSearch(start);
            if (!foundGoal) {
                System.out.println("No path to the goal found.");
            }
        } catch (Exception e) {
            System.out.println("An error occurred during the search: " + e.getMessage());
        }
    }
    private static Node createGrid1() {
        IntState.setGoal(2, 5);
        Node start1 = new Node(new IntState(0, 0, "Start"));
        Node A1 = new Node(new IntState(0, 1, "A"));
        Node B1 = new Node(new IntState(0, 3, "B"));
        Node C1 = new Node(new IntState(0, 4, "C"));
        Node D1 = new Node(new IntState(0, 5, "D"));
        Node E1 = new Node(new IntState(1, 0, "E"));
        Node F1 = new Node(new IntState(1, 1, "F"));
        Node H1 = new Node(new IntState(1, 2, "H"));
        Node I1 = new Node(new IntState(1, 3, "I"));
        Node J1 = new Node(new IntState(1, 4, "J"));
        Node K1 = new Node(new IntState(1, 5, "K"));
        Node L1 = new Node(new IntState(2, 0, "L"));
        Node M1 = new Node(new IntState(2, 1, "M"));
        Node N1 = new Node(new IntState(2, 3, "N"));
        Node O1 = new Node(new IntState(2, 4, "O"));
        Node P1 = new Node(new IntState(3, 0, "P"));
        Node Q1 = new Node(new IntState (3, 3, "Q"));
        Node R1 = new Node(new IntState(3, 4, "R"));
        Node T1 = new Node(new IntState(3, 5, "T"));
        Node goal1 = new Node(new IntState(2, 5, "Goal"));
```

Fig.3 (Output and Main.java)

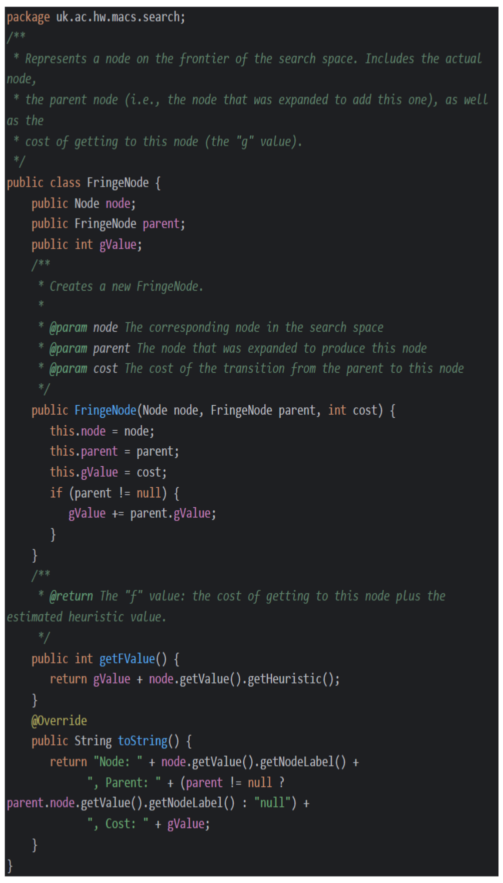


```java
package uk.ac.hw.macs.search.example;

import java.util.List;
import java.util.Set;
import uk.ac.hw.macs.search.ChildWithCost;
import uk.ac.hw.macs.search.FringeNode;
import uk.ac.hw.macs.search.SearchOrder;

import java.util.Collections;

public class AStarSearchOrder implements SearchOrder {
    @Override
    public void addToFringe(List<FringeNode> frontier, FringeNode parent, Set<ChildWithCost> children) {
        for (ChildWithCost child : children) {
            FringeNode newNode = new FringeNode(child.node, parent, child.cost);
            frontier.add(newNode);
        }
        // Sort frontier by f-values (g + h)
        Collections.sort(frontier, (a, b) -> a.getFValue() - b.getFValue());
    }
}
```

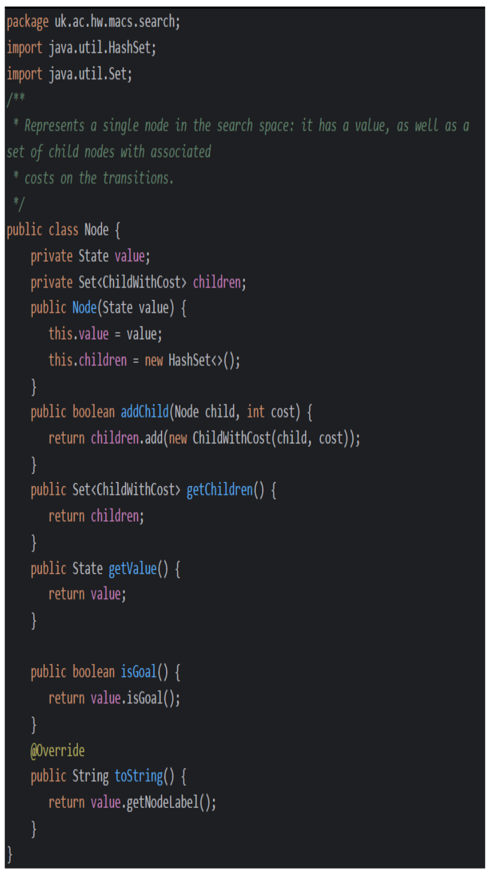


```java
package uk.ac.hw.macs.search;
/**
 * Represents a state in the search space. You need to include a method for
determining whether a state is a
 * goal state, and one for computing the heuristic value.
 */
public interface State {
    boolean isGoal();
    int getHeuristic();
    String getNodeLabel();
}
```

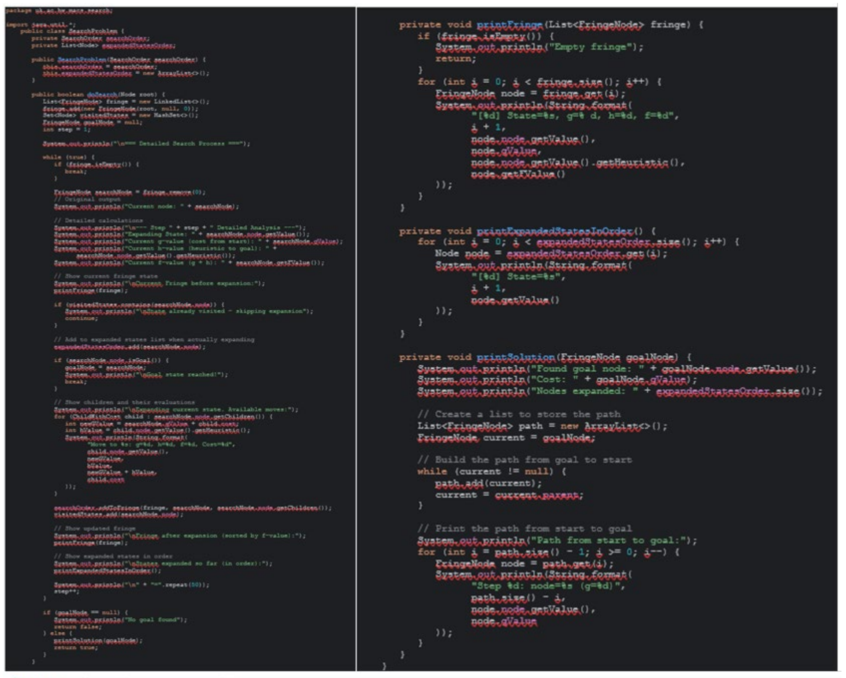
Fig.4 (AStarSearchOrder. Java() and State. Java())



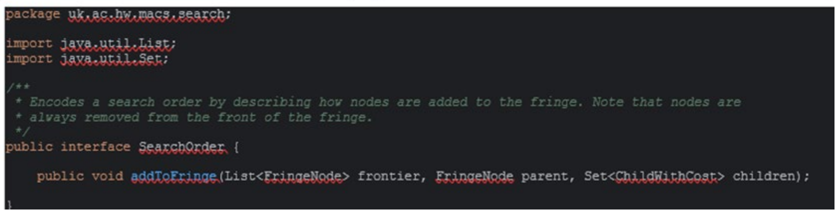
```java
package uk.ac.hw.macs.search;
/**
 * Represents a node on the frontier of the search space. Includes the actual
node,
 * the parent node (i.e., the node that was expanded to add this one), as well
as the
 * cost of getting to this node (the "g" value).
 */
public class FringeNode {
    public Node node;
    public FringeNode parent;
    public int gValue;
    /**
     * Creates a new FringeNode.
     *
     * @param node The corresponding node in the search space
     * @param parent The node that was expanded to produce this node
     * @param cost The cost of the transition from the parent to this node
     */
    public FringeNode(Node node, FringeNode parent, int cost) {
        this.node = node;
        this.parent = parent;
        this.gValue = cost;
        if (parent != null) {
            gValue += parent.gValue;
        }
    }
    /**
     * @return The "f" value: the cost of getting to this node plus the
estimated heuristic value.
     */
    public int getFValue() {
        return gValue + node.getValue().getHeuristic();
    }
    @Override
    public String toString() {
        return "Node: " + node.getValue().getNodeLabel() +
                ", Parent: " + (parent != null ?
parent.node.getValue().getNodeLabel() : "null") +
                ", Cost: " + gValue;
    }
}
```



```java
package uk.ac.hw.macs.search;
import java.util.HashSet;
import java.util.Set;
/**
 * Represents a single node in the search space: it has a value, as well as a
set of child nodes with associated
 * costs on the transitions.
 */
public class Node {
    private State value;
    private Set<ChildWithCost> children;
    public Node(State value) {
        this.value = value;
        this.children = new HashSet<>();
    }
    public boolean addChild(Node child, int cost) {
        return children.add(new ChildWithCost(child, cost));
    }
    public Set<ChildWithCost> getChildren() {
        return children;
    }
    public State getValue() {
        return value;
    }
    public boolean isGoal() {
        return value.isGoal();
    }
    @Override
    public String toString() {
        return value.getNodeLabel();
    }
}
```



```java
package uk.ac.hw.macs.search;

import java.util.*;
    public class SearchProblem {
        private SearchOrder searchOrder;
        private List<Node> expandedStatesOrder;

        public SearchProblem(SearchOrder searchOrder) {
            this.searchOrder = searchOrder;
            this.expandedStatesOrder = new ArrayList<>();
        }

        public boolean doSearch(Node root) {
            List<FringeNode> fringe = new LinkedList<>();
            fringe.add(new FringeNode(root, null, 0));
            Set<Node> visitedStates = new HashSet<>();
            FringeNode goalNode = null;
            int step = 1;

            System.out.println("\n=== Detailed Search Process ===");

            while (true) {
                if (fringe.isEmpty()) {
                    break;
                }

                FringeNode searchNode = fringe.remove(0);
                // Original output
                System.out.println("Current node: " + searchNode);

                // Detailed calculations
                System.out.println("\n--- Step " + step + " Detailed Analysis ---");
                System.out.println("Expanding State: " + searchNode.node.getValue());
                System.out.println("Current g-value (cost from start): " + searchNode.gValue);
                System.out.println("Current h-value (heuristic to goal): " +
                        searchNode.node.getValue().getHeuristic());
                System.out.println("Current f-value (g + h): " + searchNode.getFValue());

                // Show current fringe state
                System.out.println("\nCurrent Fringe before expansion:");
                printFringe(fringe);

                if (visitedStates.contains(searchNode.node)) {
                    System.out.println("\nState already visited - skipping expansion");
                    continue;
                }

                // Add to expanded states list when actually expanding
                expandedStatesOrder.add(searchNode.node);

                if (searchNode.node.isGoal()) {
                    goalNode = searchNode;
                    System.out.println("\nGoal state reached!");
                    break;
                }

                // Show children and their evaluations
                System.out.println("\nExpanding current state. Available moves:");
                for (ChildWithCost child : searchNode.node.getChildren()) {
                    int newGValue = searchNode.gValue + child.cost;
                    int hValue = child.node.getValue().getHeuristic();
                    System.out.println(String.format(
                            "Move to %s: g=%d, h=%d, f=%d, Cost=%d",
                            child.node.getValue(),
                            newGValue,
                            hValue,
                            newGValue + hValue,
                            child.cost
                    ));
                }

                searchOrder.addToFringe(fringe, searchNode, searchNode.node.getChildren());
                visitedStates.add(searchNode.node);

                // Show updated fringe
                System.out.println("\nFringe after expansion (sorted by f-value):");
                printFringe(fringe);

                // Show expanded states in order
                System.out.println("\nStates expanded so far (in order):");
                printExpandedStatesInOrder();

                System.out.println("\n" + "=".repeat(50));
                step++;
            }

            if (goalNode == null) {
                System.out.println("No goal found");
                return false;
            } else {
                printSolution(goalNode);
                return true;
            }
        }

        private void printFringe(List<FringeNode> fringe) {
            if (fringe.isEmpty()) {
                System.out.println("Empty fringe");
                return;
            }
            for (int i = 0; i < fringe.size(); i++) {
                FringeNode node = fringe.get(i);
                System.out.println(String.format(
                        "[%d] State=%s, g=% d, h=%d, f=%d",
                        i + 1,
                        node.node.getValue(),
                        node.gValue,
                        node.node.getValue().getHeuristic(),
                        node.getFValue()
                ));
            }
        }

        private void printExpandedStatesInOrder() {
            for (int i = 0; i < expandedStatesOrder.size(); i++) {
                Node node = expandedStatesOrder.get(i);
                System.out.println(String.format(
                        "[%d] State=%s",
                        i + 1,
                        node.getValue()
                ));
            }
        }

        private void printSolution(FringeNode goalNode) {
            System.out.println("Found goal node: " + goalNode.node.getValue());
            System.out.println("Cost: " + goalNode.gValue);
            System.out.println("Nodes expanded: " + expandedStatesOrder.size());

            // Create a list to store the path
            List<FringeNode> path = new ArrayList<>();
            FringeNode current = goalNode;

            // Build the path from goal to start
            while (current != null) {
                path.add(current);
                current = current.parent;
            }

            // Print the path from start to goal
            System.out.println("Path from start to goal:");
            for (int i = path.size() - 1; i >= 0; i--) {
                FringeNode node = path.get(i);
                System.out.println(String.format(
                        "Step %d: node=%s (g=%d)",
                        path.size() - i,
                        node.node.getValue(),
                        node.gValue
                ));
            }
        }
    }
```



```java
package uk.ac.hw.macs.search;

import java.util.List;
import java.util.Set;

/**
 * Encodes a search order by describing how nodes are added to the fringe. Note that nodes are
 * always removed from the front of the fringe.
 */
public interface SearchOrder {

    public void addToFringe(List<FringeNode> frontier, FringeNode parent, Set<ChildWithCost> children);
}
```

Fig.5 (FringeNode.Java() and Node. Java ())

Fig.6 (SearchProblem.Java() and SearchOrder. Java ())

Part 2 : Automated Planning Solution for Underwater Robotic Missions

Introduction

PDDL is a human-readable format for problems in automated planning that gives a description of the possible states of the world, a description of the set of possible actions, a specific initial state of the world, and a specific set of desired goals (McCluskey, 2003).

The modelling and solving of underwater robotic inspection of wind farms missions using the Planning Domain Definition Language (PDDL) is the focus of this task. The Unmanned Underwater Vehicle (UUV) is tasked with navigating through a series of waypoints, capturing images, performing sonar scans, and collecting samples.

The two main components of PDDL (Höller, and Alford, 2020) are:

- **Domain Modelling:** Creating a representation of the environment, the objects involved, and the actions the UUV can perform.
- **Problem Modelling**: Specifying the initial conditions, the mission goals, and generating the plan to achieve those goals.

Domain Structure

There are five (5) domain files that align with the respective problems required as part of this report. Here is a quick summary of those five domains:

1. **Domain 1** focuses on simple UUV operations such as deployment, movement, and data capture.
2. **Domain 2** extends the functionality to include sample collection and storage.
3. **Domain 3** introduces multiple UUVs and their assignment to specific ships.
4. **Domain 4** adds engineers to manage UUV deployment and data transmission from control centers.
5. **Domain 5** includes environmental obstacles like algae that can hinder UUV movements, requiring engineer intervention.

Types and Predicates

The domain model uses types to define categories of objects and predicates involved in the mission. These include:

1. **UUV:** The unmanned vehicle responsible for navigating underwater and performing tasks.
2. **Ship:** The vehicle that carries and deploys the UUV.
3. **Location:** This includes both underwater waypoints and the ship.
4. **Data:** The various types of data the UUV can collect, including images, sonar scans, and samples.

The key predicates used in the domain to represent the state of the system are:

1. **(at ?u - uuv ?l - location):** Specifies the location of the UUV.
2. **(on-ship ?u - uuv ?s - ship):** Indicates that the UUV is on the ship.
3. **(connected ?from - location ?to - location):** Describes connectivity between locations, allowing the UUV to move between connected waypoints.
4. **(memory-empty ?u - uuv):** Checks if the UUV's memory is empty, allowing it to collect new data.
5. **(has-data ?u - uuv ?d - data):** Specifies that the UUV has collected a specific piece of data (image, sonar, or sample).
6. **(data-saved ?d - data ?l - location):** Indicates that a piece of data has been successfully transmitted to the ship or saved at a specific location.

These predicates ensure that the planner has a clear understanding of the UUV's position, memory status, and the tasks completed at each waypoint.

Locations

The domain defines several types of locations for the UUVs and engineers to interact with. These include waypoints, ships, bays, and control centers. Each type of location serves a specific purpose in the mission.

- **Waypoints**, such as `(image-at ?img - image ?w - waypoint)`,are underwater locations where the UUV can perform tasks such as capturing images, performing sonar scans or collecting samples from specific waypoints.
- In **domain 4** and **domain 5**, **bays** and **control centers** are introduced on the ships. UUVs can only be deployed or returned to the ship when the engineer is at the bay. While data transmission from the UUV to the ship can only occur when the engineer is at the control centre. Here are examples for these settings in PDDL:
  ```
  (bay-for-ship ?b - bay ?s - ship)
  (control-for-ship ?c - control-centre ?s - ship)
  ```
- Here is an example of control center action that allows engineers to monitor UUV operations and transmit data:

```
(:action transmit-data
    :parameters (?u - uuv ?d - data ?l - location ?e - engineer ?c - control-center ?s - ship)
    :precondition (and
        (at ?u ?l)
        (has-data ?u ?d)
        (engineer-at ?e ?c)
        (control-for-ship ?c ?s)
        (assigned-to ?u ?s)
    )
    :effect (and
        (not (has-data ?u ?d))
        (data-saved ?d ?l)
    )
)
```

Task 1.2: Defining Actions

The domain defines several actions that describe the UUV's behaviour and capabilities during the mission. Each action has specific preconditions that must be met before the action can be executed and effects that describe how the state of the world changes because of the action. The main actions defined are:

- **Deploy UUV:** (deploy-uuv) moves the UUV from the ship to a specified waypoint, provided the UUV is on the ship and the ship is connected to the waypoint. Preconditions: The UUV must be on the ship, the ship must be connected to the waypoint, and the UUV must be available for deployment. Effects: The UUV is no longer on the ship, it is at the specified waypoint, and it can no longer be redeployed.
- **Move UUV:** (move-uuv) moves the UUV between connected waypoints. Preconditions: The UUV must be at the starting waypoint, and the waypoints must be connected. Effects: The UUV moves to the destination waypoint.
- **Capture Image:** (capture-image) allows the UUV to take an image at a specific waypoint. Preconditions: The UUV must be at the waypoint, the waypoint must have an image to capture, and the UUV's memory must be empty. Effects: The UUV stores the image in its memory.
- **Perform Sonar Scan:** (perform-sonar-scan) allows the UUV to perform a sonar scan at a specific waypoint. Preconditions: The UUV must be at the waypoint, the waypoint must have a sonar scan available, and the UUV's memory must be empty. Effects: The UUV stores the sonar scan data in its memory.
- **Transmit Data:** (transmit-data) allows the UUV to send captured data (image or sonar scan) back to the ship. Preconditions: The UUV must be at the correct location, have data stored in its memory, and be able to transmit the data to the ship. Effects: The data is transmitted, and the UUV's memory is cleared.
- **Return to Ship:** (return-to-ship) moves the UUV back to the ship from a connected waypoint. Preconditions: The UUV must be at a waypoint connected to the ship. Effects: The UUV is back on the ship, ready for redeployment if needed.

Part 2B: Problem Modelling

The problem files define the specific missions the UUV must implement , including the initial state of the system, the waypoints the UUV must visit, and the tasks it must accomplish.

Problem 1:

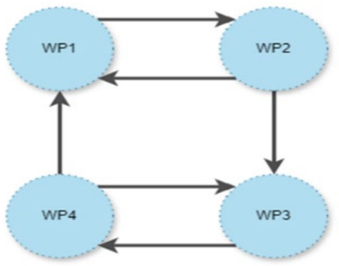
Initial State: The UUV starts on the ship, with its memory empty and ready for deployment. The map includes waypoints connected by bidirectional paths**.**

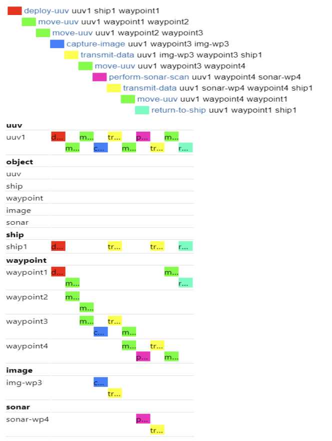
Goals: The UUV must:

- Capture an image at waypoint 3.
- Perform a sonar scan at waypoint 4.
- Return to the ship.

PDDL Files:

- Domain name is Windfarm-1 (Domain-1.pddl)
- Problem name is Windfarm-mission-1(Problem-1.pddl)





```
deploy-uuv uuv1 ship1 waypoint1
 move-uuv uuv1 waypoint1 waypoint2
  move-uuv uuv1 waypoint2 waypoint3
   capture-image uuv1 waypoint3 img-wp3
    transmit-data uuv1 img-wp3 waypoint3 ship1
     move-uuv uuv1 waypoint3 waypoint4
      perform-sonar-scan uuv1 waypoint4 sonar-wp4
       transmit-data uuv1 sonar-wp4 waypoint4 ship1
        move-uuv uuv1 waypoint4 waypoint1
         return-to-ship uuv1 waypoint1 ship1
```

Fig.7 Plan for Problem 1

Problem 2:

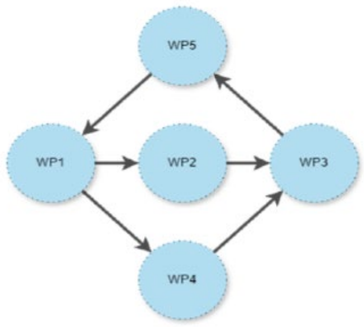
Initial State: The same as Problem 1, but with an additional waypoint and new tasks.

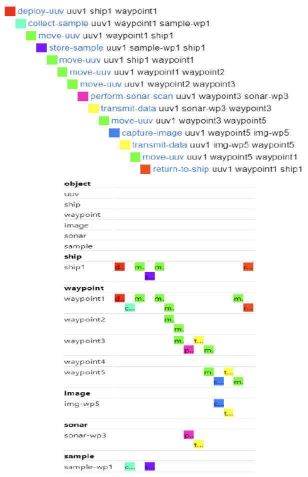
Goals: The UUV must:

- Capture an image at waypoint 5.
- Perform a sonar scan at waypoint 3.
- Collect a sample from waypoint 1.
- Return to the ship.

PDDL Files:

- Domain name is Windfarm-2 (Domain-2.pddl)
- Problem name is Windfarm-mission-2(Problem-2.pddl)





```
deploy-uuv uuv1 ship1 waypoint1
 collect-sample uuv1 waypoint1 sample-wp1
  move-uuv uuv1 waypoint1 ship1
   store-sample uuv1 sample-wp1 ship1
    move-uuv uuv1 ship1 waypoint1
     move-uuv uuv1 waypoint1 waypoint2
      move-uuv uuv1 waypoint2 waypoint3
       perform-sonar-scan uuv1 waypoint3 sonar-wp3
        transmit-data uuv1 sonar-wp3 waypoint3
         move-uuv uuv1 waypoint3 waypoint5
          capture-image uuv1 waypoint5 img-wp5
           transmit-data uuv1 img-wp5 waypoint5
            move-uuv uuv1 waypoint5 waypoint1
             return-to-ship uuv1 waypoint1 ship1
```

Fig.8 Plan for Problem 2

Problem 3:

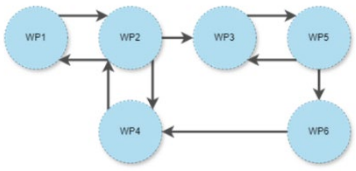
Initial State: Two UUVs are used in this problem, with UUV 1 starting at waypoint 2 and UUV 2 on a secondary ship at waypoint 3. The map involves six waypoints**.**

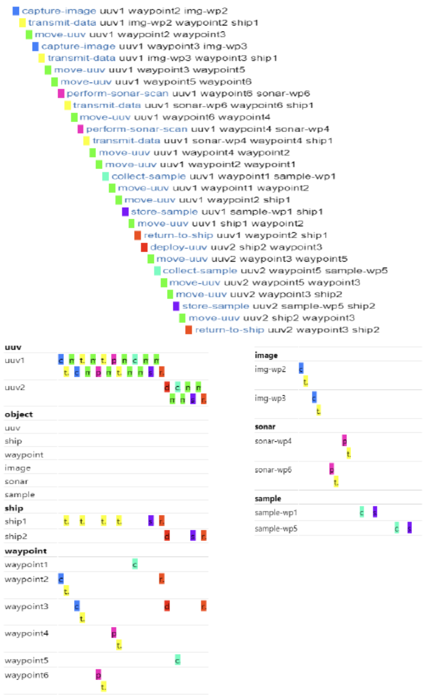
Goals: The tasks are distributed between the two UUVs:

- UUV 1 must capture an image at waypoints 2 and 3, perform sonar scans at waypoints 4 and 6, and collect a sample at waypoint 1.
- UUV 2 must collect a sample from waypoint 5 and return to its ship.

PDDL Files:

- Domain name is Windfarm-3 (Domain-3.pddl)
- Problem name is Windfarm-mission-3(Problem-3.pddl)





```
capture-image uuv1 waypoint2 img-wp2
 transmit-data uuv1 img-wp2 waypoint2 ship1
  move-uuv uuv1 waypoint2 waypoint3
   capture-image uuv1 waypoint3 img-wp3
    transmit-data uuv1 img-wp3 waypoint3 ship1
     move-uuv uuv1 waypoint3 waypoint5
      move-uuv uuv1 waypoint5 waypoint6
       perform-sonar-scan uuv1 waypoint6 sonar-wp6
        transmit-data uuv1 sonar-wp6 waypoint6 ship1
         move-uuv uuv1 waypoint6 waypoint4
          perform-sonar-scan uuv1 waypoint4 sonar-wp4
           transmit-data uuv1 sonar-wp4 waypoint4 ship1
            move-uuv uuv1 waypoint4 waypoint2
             move-uuv uuv1 waypoint2 waypoint1
              collect-sample uuv1 waypoint1 sample-wp1
               move-uuv uuv1 waypoint1 waypoint2
                move-uuv uuv1 waypoint2 ship1
                 store-sample uuv1 sample-wp1 ship1
                  move-uuv uuv1 ship1 waypoint2
                   return-to-ship uuv1 waypoint2 ship1
                    deploy-uuv uuv2 ship2 waypoint3
                     move-uuv uuv2 waypoint3 waypoint5
                      collect-sample uuv2 waypoint5 sample-wp5
                       move-uuv uuv2 waypoint5 waypoint3
                        move-uuv uuv2 waypoint3 ship2
                         store-sample uuv2 sample-wp5 ship2
                          move-uuv uuv2 ship2 waypoint3
                           return-to-ship uuv2 waypoint3 ship2
```

Fig.9 Plan for Problem 3

Part 2C: Extended Problems

Problem 4:

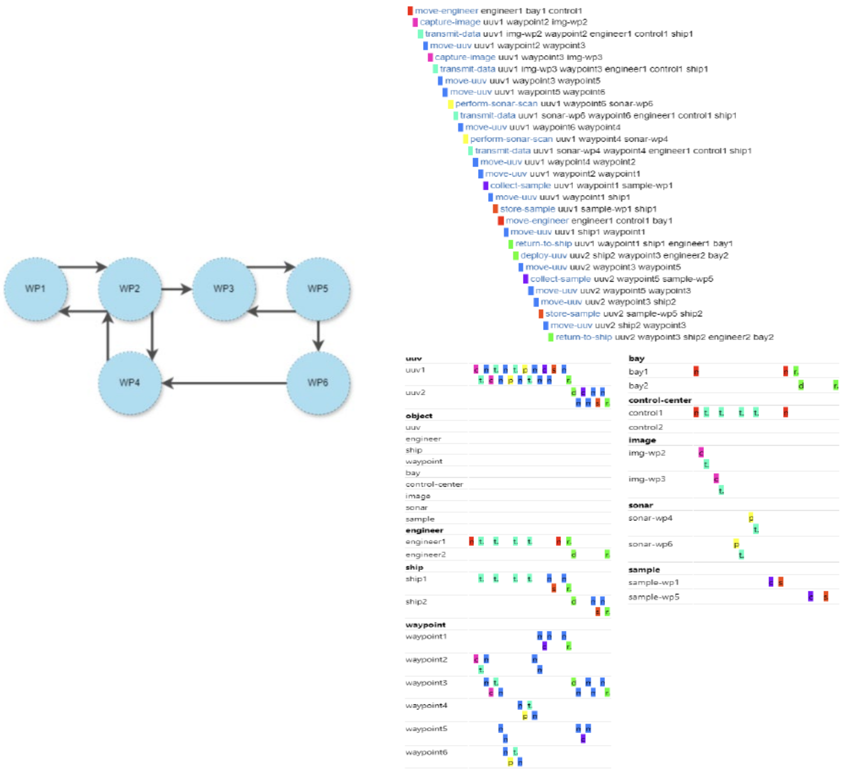
Initial State: This problem introduces an engineer who assists with the deployment of the UUV and the transmission of data. The UUV can only be deployed if the engineer is at the bay, and data can only be transmitted when the engineer is in the control centre. The setting of the environment is same as that of Problem-3 with the actions needing to be adjusted to account for the preconditions.

Goals: The tasks are distributed between the two UUVs:

- UUV 1 must capture an image at waypoints 2 and 3, perform sonar scans at waypoints 4 and 6, and collect a sample at waypoint 1.
- UUV 2 must collect a sample from waypoint 5 and return to its ship.

PDDL Files:

- Domain name is Windfarm-4 (Domain-4.pddl)
- Problem name is Windfarm-mission-4(Problem-4.pddl)



```
move-engineer engineer1 bay1 control1
capture-image uuv1 waypoint2 img-wp2
transmit-data uuv1 img-wp2 waypoint2 engineer1 control1 ship1
move-uuv uuv1 waypoint2 waypoint3
capture-image uuv1 waypoint3 img-wp3
transmit-data uuv1 img-wp3 waypoint3 engineer1 control1 ship1
move-uuv uuv1 waypoint3 waypoint5
move-uuv uuv1 waypoint5 waypoint6
perform-sonar-scan uuv1 waypoint6 sonar-wp6
transmit-data uuv1 sonar-wp6 waypoint6 engineer1 control1 ship1
move-uuv uuv1 waypoint6 waypoint4
perform-sonar-scan uuv1 waypoint4 sonar-wp4
transmit-data uuv1 sonar-wp4 waypoint4 engineer1 control1 ship1
move-uuv uuv1 waypoint4 waypoint2
move-uuv uuv1 waypoint2 waypoint1
collect-sample uuv1 waypoint1 sample-wp1
move-uuv uuv1 waypoint1 ship1
store-sample uuv1 sample-wp1 ship1
move-engineer engineer1 control1 bay1
move-uuv uuv1 ship1 waypoint1
return-to-ship uuv1 waypoint1 ship1 engineer1 bay1
deploy-uuv uuv2 ship2 waypoint3 engineer2 bay2
move-uuv uuv2 waypoint3 waypoint5
collect-sample uuv2 waypoint5 sample-wp5
move-uuv uuv2 waypoint5 waypoint3
move-uuv uuv2 waypoint3 ship2
store-sample uuv2 sample-wp5 ship2
move-uuv uuv2 ship2 waypoint3
return-to-ship uuv2 waypoint3 ship2 engineer2 bay2
```

Fig.10 Plan for Problem-4

Problem 5:

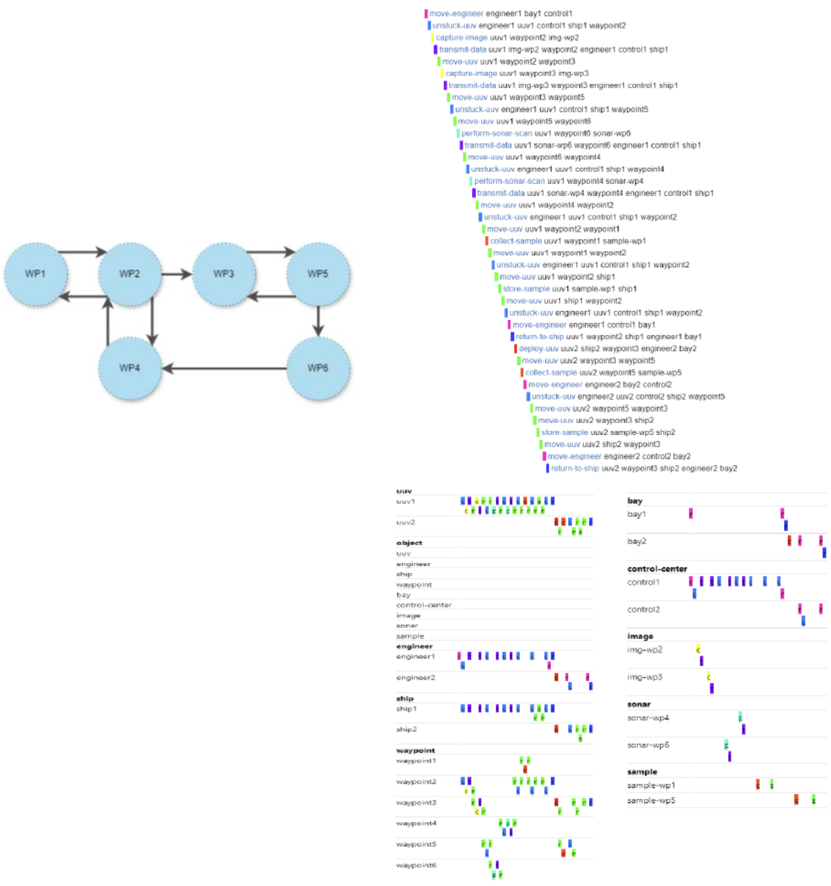
Initial State: In this scenario, algae present at certain waypoints block the UUV's movement, requiring the engineer to execute an "unstuck routine" from the control centre. The UUV will get stuck if it enters a waypoint with algae, halting its movement until the algae is cleared.

Goals: The tasks are distributed between the two UUVs:

- UUV 1 must capture an image at waypoints 2 and 3, perform sonar scans at waypoints 4 and 6, and collect a sample at waypoint 1.
- UUV 2 must collect a sample from waypoint 5 and return to its ship.

PDDL Files:

- Domain name is Windfarm-5 (Domain-5.pddl)
- Problem name is Windfarm-mission-5(Problem-5.pddl)



```
move-engineer engineer1 bay1 control1
unstuck-uuv engineer1 uuv1 control1 ship1 waypoint2
capture-image uuv1 waypoint2 img-wp2
transmit-data uuv1 img-wp2 waypoint2 engineer1 control1 ship1
move-uuv uuv1 waypoint2 waypoint3
capture-image uuv1 waypoint3 img-wp3
transmit-data uuv1 img-wp3 waypoint3 engineer1 control1 ship1
move-uuv uuv1 waypoint3 waypoint5
unstuck-uuv engineer1 uuv1 control1 ship1 waypoint5
move-uuv uuv1 waypoint5 waypoint6
perform-sonar-scan uuv1 waypoint6 sonar-wp6
transmit-data uuv1 sonar-wp6 waypoint6 engineer1 control1 ship1
move-uuv uuv1 waypoint6 waypoint4
unstuck-uuv engineer1 uuv1 control1 ship1 waypoint4
perform-sonar-scan uuv1 waypoint4 sonar-wp4
transmit-data uuv1 sonar-wp4 waypoint4 engineer1 control1 ship1
move-uuv uuv1 waypoint4 waypoint2
unstuck-uuv engineer1 uuv1 control1 ship1 waypoint2
move-uuv uuv1 waypoint2 waypoint1
collect-sample uuv1 waypoint1 sample-wp1
move-uuv uuv1 waypoint1 waypoint2
unstuck-uuv engineer1 uuv1 control1 ship1 waypoint2
move-uuv uuv1 waypoint2 ship1
store-sample uuv1 sample-wp1 ship1
move-uuv uuv1 ship1 waypoint2
unstuck-uuv engineer1 uuv1 control1 ship1 waypoint2
move-engineer engineer1 control1 bay1
return-to-ship uuv1 waypoint2 ship1 engineer1 bay1
deploy-uuv uuv2 ship2 waypoint3 engineer2 bay2
move-uuv uuv2 waypoint3 waypoint5
collect-sample uuv2 waypoint5 sample-wp5
move-engineer engineer2 bay2 control2
unstuck-uuv engineer2 uuv2 control2 ship2 waypoint5
move-uuv uuv2 waypoint5 waypoint3
move-uuv uuv2 waypoint3 ship2
store-sample uuv2 sample-wp5 ship2
move-uuv uuv2 ship2 waypoint3
move-engineer engineer2 control2 bay2
return-to-ship uuv2 waypoint3 ship2 engineer2 bay2
```

Fig.11 Plan for Problem-5

References

Black, P.E., 2006. Manhattan distance"" Dictionary of algorithms and data structures. *http://xlinux. nist. gov/dads//.*

Höller, D., Behnke, G., Bercher, P., Biundo, S., Fiorino, H., Pellier, D. and Alford, R., 2020, April. HDDL: An extension to PDDL for expressing hierarchical planning problems. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 34, No. 06, pp. 9883-9891).

McCluskey, T.L., 2003. PDDL: A Language with a Purpose?

Sierra, K. and Bates, B., 2005. *Headfirst Java: A Brain-Friendly Guide*. " O'Reilly Media, Inc.".

VIDEO LINKS

1) https://heriotwatt.sharepoint.com/:v:/s/F29AIGroup3/EQZK-E-9Uc9OnUkUfrIX3fEBTtej-VZrbACctpFk9oOkug?e=uNOQxu
2) https://youtu.be/F-IIFzk6B6U
3) https://heriotwatt.sharepoint.com/:v:/r/sites/F29AIGroup3/Shared%20Documents/General/Recordings/Video%20Recording%20Coursework-1%20PG%20Dubai%20-%20Group3.mp4?csf=1&web=1&e=ZdEube