This is the format of the directory of the package (followed by the working code)

-Main Folder -uk.ac.hw.macs

-Subfolder 1 - Search

File 1.1 – ChildWithCost.java

File 1.2 – FringeNode.java

File 1.3 – Node.java

File 1.4 – SearchOrder.java

File 1.5 – SearchProblem.java

File 1.6 – State.java

Sub-Sub-folder 1.7 – Example

File 1.7.1 – AStarSearchOrder.java

File 1.7.2 – BreadthFirstSearchOrder.java

File 1.7.3 – DepthFirstSearchOrder.java

File 1.7.4 – IntState.java

File 1.7.5 – Main.java

---

Subfolder 1 - Search

File 1.1 – ChildWithCost.java

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

/**
 * Represents a connection in the state graph: a node, and the cost of getting from the parent to this node.
 */
public class ChildWithCost  {

    public final Node node;
    public final int cost;

    public ChildWithCost(Node child, int cost) {
       this.node = child;
       this.cost = cost;
    }

}
```

---

File 1.2 – FringeNode.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;
    }
}
```

File 1.3 – Node.java

```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();
    }
}
```

File 1.4 – SearchOrder.java

```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);

}
```

File 1.5 – SearchProblem.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
              ));
          }
      }
  }
```

File 1.6 – State.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();
}
```

Sub-Sub-folder 1.7 – Example

File 1.7.1 – AStarSearchOrder.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());
    }
}
```

File 1.7.2 – BreadthFirstSearchOrder.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;
/**
 * A simple search order that adds all nodes to the end of the fringe.
 */
public class BreadthFirstSearchOrder implements SearchOrder {
    @Override
    public void addToFringe(List<FringeNode> frontier, FringeNode parent,
Set<ChildWithCost> children) {
       // Add them at the end, ignoring the cost
       for (ChildWithCost child : children) {
          frontier.add(new FringeNode(child.node, parent, child.cost));// Add
to end of the fringe
       }
    }
}
```

File 1.7.3 – DepthFirstSearchOrder.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;

/**
 * A simple search order that adds all nodes to the start of the fringe.
 */
public class DepthFirstSearchOrder implements SearchOrder {

    @Override
    public void addToFringe(List<FringeNode> frontier, FringeNode parent,
Set<ChildWithCost> children) {
       // Add them at the start, ignoring the cost
       for (ChildWithCost child : children) {
          frontier.add(0, new FringeNode(child.node, parent, child.cost));//
Add to start of the fringe
       }
    }
}
```

---

File 1.7.4 – IntState.java

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

/**
 * A state representing a position in a 2D grid with coordinates (x, y).
 * It also includes goal-checking and heuristic (Manhattan distance).
 */
public class IntState implements State {
    private int x, y;
    private static int goalX, goalY;
    private String nodeLabel;

    public IntState(int x, int y, String nodeLabel) {
        this.x = x;
        this.y = y;
        this.nodeLabel = nodeLabel;
    }
    public static void setGoal(int goalX, int goalY) {
        IntState.goalX = goalX;
        IntState.goalY = goalY;
    }
    @Override
    public boolean isGoal() {
        return this.x == goalX && this.y == goalY;
    }
    @Override
    public int getHeuristic() {
        return Math.abs(this.x - goalX) + Math.abs(this.y - goalY);
    }
    @Override
    public String getNodeLabel() {
        return nodeLabel;
    }
    @Override
    public String toString() {
        return nodeLabel;
    }
}
```

File 1.7.5 – Main.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
    /**
     * Gets valid integer input within a specified range
     */
    private static int getValidInput(Scanner scanner, int min, int max, String prompt, String
errorMessage) {
        while (true) {
            try {
                System.out.println(prompt);
                if (!scanner.hasNextInt()) {
                    scanner.nextLine(); // Clear invalid input
                    throw new IllegalArgumentException(errorMessage);
                }
                int input = scanner.nextInt();
                scanner.nextLine(); // Clear buffer
                if (input < min || input > max) {
                    throw new IllegalArgumentException(errorMessage);
                }
                return input;
            } catch (IllegalArgumentException e) {
                System.out.println(e.getMessage());
            }
        }
    }
    /**
     * Gets valid yes/no response for continuing the program
     */
    private static boolean getContinueResponse(Scanner scanner) {
        while (true) {
            try {
                System.out.println("\nDo you want to run another search? (y/n): ");
                String response = scanner.nextLine().trim().toLowerCase();

                if (response.equals("y") || response.equals("yes")) {
                    return true;
                } else if (response.equals("n") || response.equals("no")) {
                    return false;
                } else {
                    throw new IllegalArgumentException("Invalid input! Please enter 'y' or
'n'");
                }

            } catch (IllegalArgumentException e) {
                System.out.println(e.getMessage());
            }
        }
    }
```

```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"));
```

```
        start1.addChild(A1, 1);
        start1.addChild(E1, 2);
        A1.addChild(F1, 2);
        B1.addChild(I1, 2);
        B1.addChild(C1, 1);
        C1.addChild(D1, 1);
        C1.addChild(J1, 1);
        E1.addChild(F1, 2);
        E1.addChild(L1, 1);
        F1.addChild(H1, 1);
        F1.addChild(M1, 1);
        H1.addChild(I1, 2);
        I1.addChild(J1, 1);
        I1.addChild(N1, 2);
        J1.addChild(K1, 1);
        J1.addChild(O1, 2);
        K1.addChild(goal1, 1);
        L1.addChild(M1, 1);
        L1.addChild(P1, 2);
        N1.addChild(O1, 2);
        N1.addChild(Q1, 1);
        O1.addChild(goal1, 1);
        O1.addChild(R1, 2);
        goal1.addChild(T1, 2);
        return start1;
    }
    private static Node createGrid2() {
        IntState.setGoal(4, 3);
        Node start2 = new Node(new IntState(0, 0, "Start"));
        Node A2 = new Node(new IntState(0, 1, "A"));
        Node B2 = new Node(new IntState(0, 3, "B"));
        Node C2 = new Node(new IntState(0, 4, "C"));
        Node D2 = new Node(new IntState(1, 0, "D"));
        Node E2 = new Node(new IntState(1, 1, "E"));
        Node F2 = new Node(new IntState(1, 2, "F"));
        Node H2 = new Node(new IntState(1, 3, "H"));
        Node I2 = new Node(new IntState(1, 4, "I"));
        Node J2 = new Node(new IntState(2, 0, "J"));
        Node K2 = new Node(new IntState(2, 1, "K"));
        Node L2 = new Node(new IntState(2, 3, "L"));
        Node M2 = new Node(new IntState(2, 4, "M"));
        Node N2 = new Node(new IntState(3, 0, "N"));
        Node O2 = new Node(new IntState(3, 1, "O"));
        Node P2 = new Node(new IntState(3, 2, "P"));
        Node Q2 = new Node(new IntState(3, 3, "Q"));
        Node R2 = new Node(new IntState(3, 4, "R"));
        Node T2 = new Node(new IntState(4, 0, "T"));
        Node U2 = new Node(new IntState(4, 4, "U"));
        Node goal2 = new Node(new IntState(4, 3, "Goal"));
```

```
        start2.addChild(A2, 1);
        start2.addChild(D2, 2);
        A2.addChild(E2, 1);
        B2.addChild(C2, 1);
        B2.addChild(H2, 2);
        C2.addChild(I2, 1);
        E2.addChild(F2, 1);
        E2.addChild(K2, 1);
        D2.addChild(E2, 1);
        D2.addChild(J2, 1);
        F2.addChild(H2, 2);
        H2.addChild(I2, 1);
        H2.addChild(L2, 1);
        I2.addChild(M2, 1);
        J2.addChild(K2, 1);
        J2.addChild(N2, 2);
        K2.addChild(O2, 1);
        L2.addChild(M2, 1);
        L2.addChild(Q2, 1);
        M2.addChild(R2, 2);
        N2.addChild(O2, 1);
        N2.addChild(T2, 2);
        O2.addChild(P2, 2);
        P2.addChild(Q2, 1);
        Q2.addChild(R2, 2);
        Q2.addChild(goal2, 1);
        R2.addChild(U2, 2);

        return start2;
    }
}
```

---