Search Agent LLAP Problem

Project 1 Report

Team 66

By

Yousef Mohamed Hassan 49-0560

Ziad Khalil 49-5767

Amr Mohammed 49-15440

Omar Muhammed 49-11483

Artificial Intelligence

Haitham O. Ismail

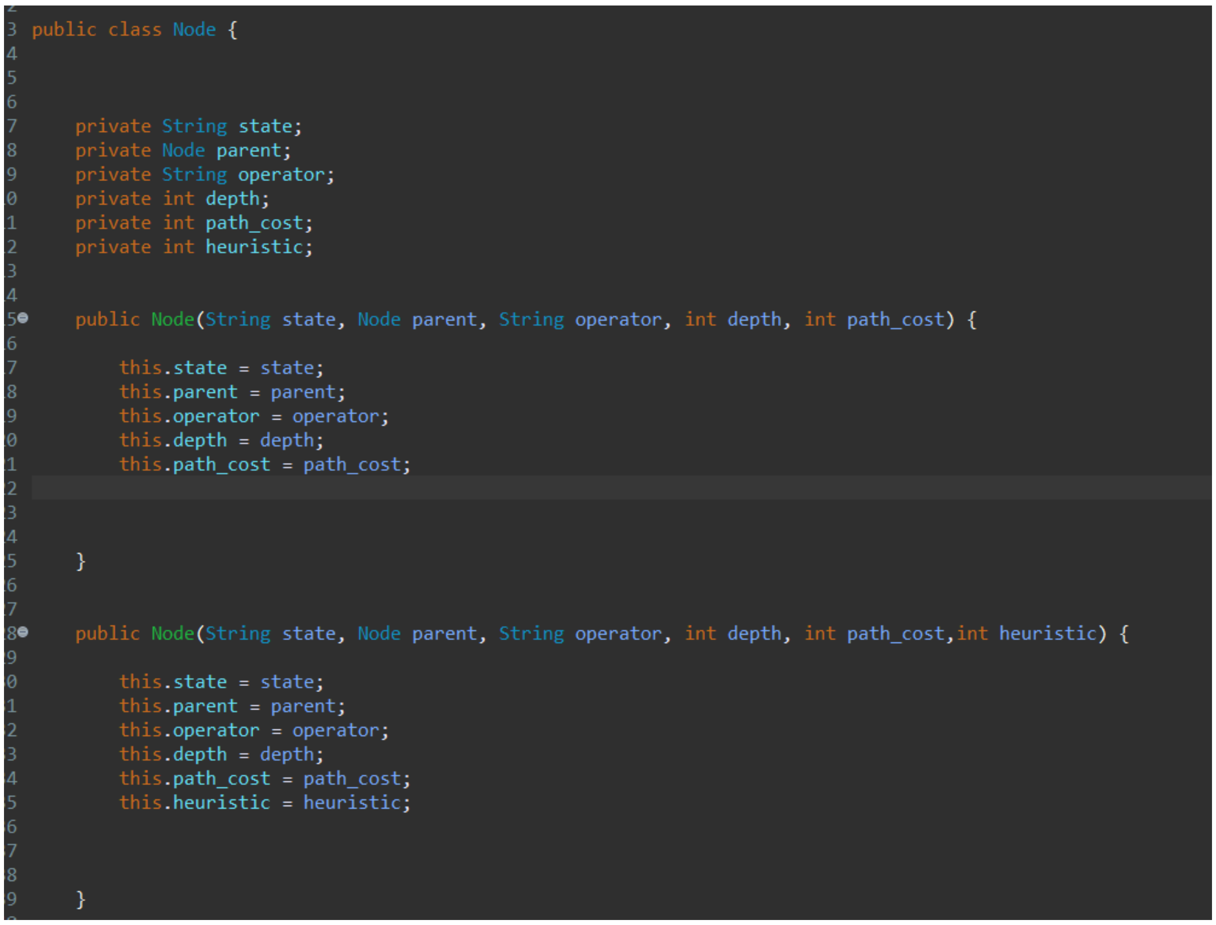
14-11-2023

Introduction

The problem involves designing a search agent to help a town reach a prosperity level of 100 by efficiently managing resources within a budget of 100,000. The town requires food, materials, and energy for its citizens and new buildings. Buildings contribute to prosperity but require resources. The agent must consider actions that affect prosperity and resources differently, manage a resource storage limit, and account for the cost of resource consumption. The agent operates with a fixed budget and has no additional income sources. The goal is to find a plan that optimally utilizes the budget to achieve the desired prosperity level. In addressing the “Live Long and Prosper” problem, a comprehensive set of six search algorithms has been employed: Breadth-First Search, Depth-First Search, Iterative Deepening Search, Uniform Cost Search, Greedy Best-First Search, and A* Search. Each algorithm operates within a complex state space comprising six distinct states, each demanding specific conditions for expansion.

Search Tree Node ADT Implementation

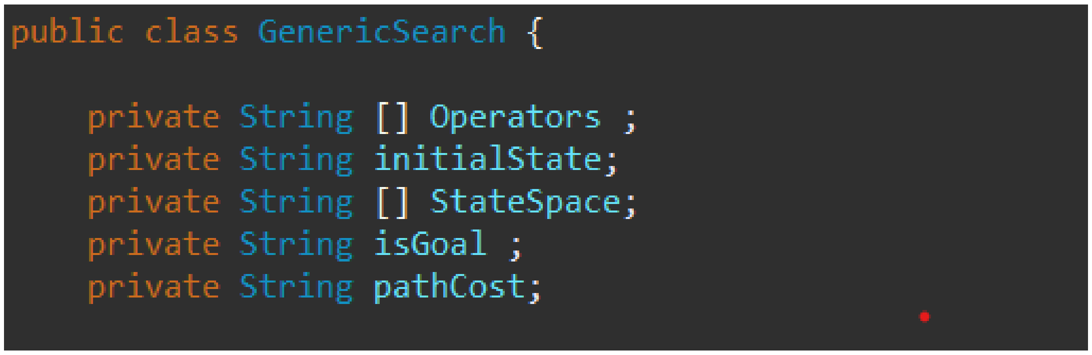
We used a search tree node comprising of 5 and 6 attributes by creating a class called "Node". The class has 2 different constructors , first one initializes 5 attributes that are: state – which represents the state of the node , parent – which refers to the parent of the node of type "Node" as well, operator – which represents the name of the operator leading to the this state, depth – which represents the depth of the node in the tree and path cost- which represents the cost from the root to reaching the current node/state. The second one has the same attributes in additional to an extra one called heuristic- which represents the heuristic value according to the heuristic function used, it is used for A* and Greedy Algorithms. This is shown in the below figure. Moreover, we created corresponding getters and setters for each attribute.



```
public class Node {



    private String state;
    private Node parent;
    private String operator;
    private int depth;
    private int path_cost;
    private int heuristic;

    public Node(String state, Node parent, String operator, int depth, int path_cost) {

        this.state = state;
        this.parent = parent;
        this.operator = operator;
        this.depth = depth;
        this.path_cost = path_cost;



    }


    public Node(String state, Node parent, String operator, int depth, int path_cost,int heuristic) {

        this.state = state;
        this.parent = parent;
        this.operator = operator;
        this.depth = depth;
        this.path_cost = path_cost;
        this.heuristic = heuristic;



    }
```

Search Problem ADT Implementation

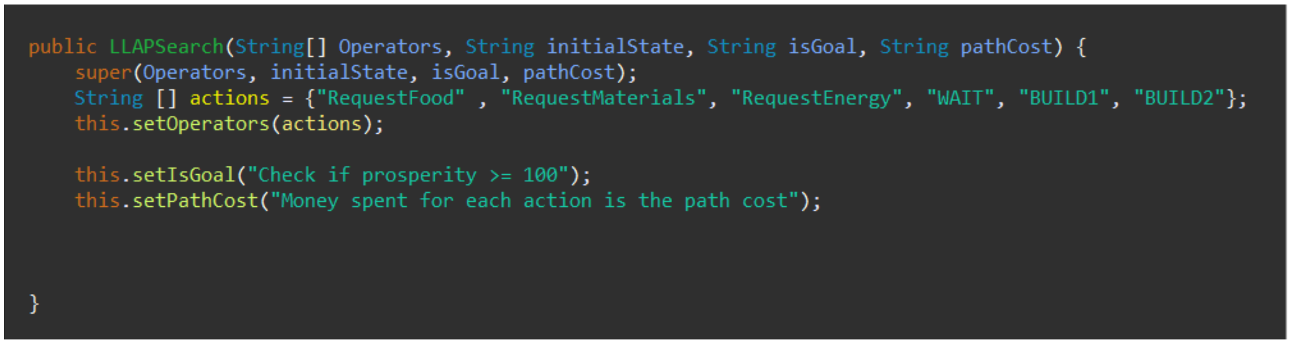
We used the following approach for the search problem representation, by creating a class called "GenericSearch" , which represents generic search problem by creating 5 attributes that are: Operators – which represents a set of all the operators/actions that can be invoked in the search problem , Initial state – which represents the initial state of the search problem indicating where it starts , State space – which represents the set of states that are reachable from the initial state of the search , Goal Test – which identifies the goal test carried out on each state to determine whether it is a goal or not , Path cost – which identifies the function used to calculate the path cost of each state/node. This is shown in the below figure. Moreover, we have created getters and setters for each attribute in order to edit and view them according to the search problem.



```java
public class GenericSearch {

    private String [] Operators ;
    private String initialState;
    private String [] StateSpace;
    private String isGoal ;
    private String pathCost;
```

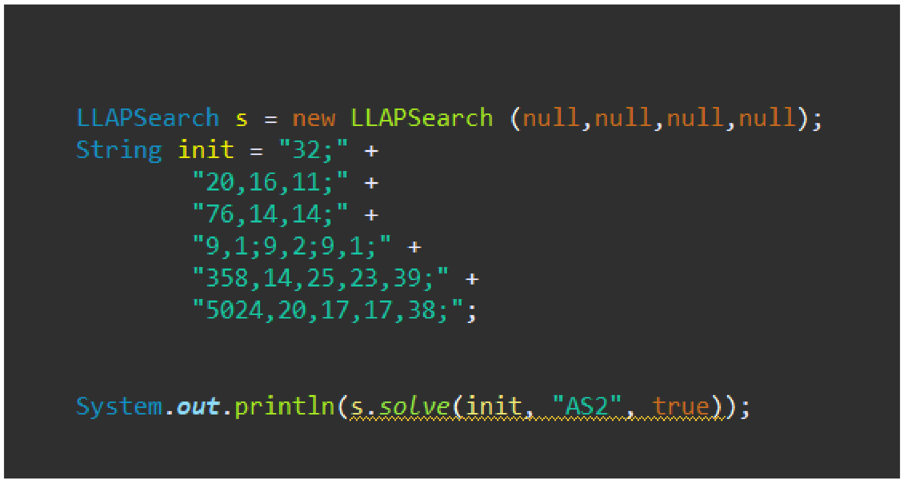
LLAP Problem Implementation

In this problem, we used the following approach for the LLAP problem Implementation. First, we have created a class called “LLAPSearch” which is a subclass of “GenericSearch” inheriting the same attributes and initializing additional new attributes that are used in the LLAP search problem. Then, some of the attributes of the search problem are changed as shown in the below figure.



```java
public LLAPSearch(String[] Operators, String initialState, String isGoal, String pathCost) {
    super(Operators, initialState, isGoal, pathCost);
    String [] actions = {"RequestFood" , "RequestMaterials", "RequestEnergy", "WAIT", "BUILD1", "BUILD2"};
    this.setOperators(actions);

    this.setIsGoal("Check if prosperity >= 100");
    this.setPathCost("Money spent for each action is the path cost");


}
```

We then created a static method called “solve” which takes three inputs - the initial state as a String which represents the state the search problem starts in , strategy as a String which represents the search algorithm to be used in the problem , and visualize as Boolean which controls whether to print state information at each state or not . The “solve” function calls different helper methods according to the input which will be discussed in the following sections. The search problem is tested by creating a new Object of type “LLAPSearch” and calling the solve function on it as shown in the figure below.



```java
LLAPSearch s = new LLAPSearch (null,null,null,null);
String init = "32;" +
        "20,16,11;" +
        "76,14,14;" +
        "9,1;9,2;9,1;" +
        "358,14,25,23,39;" +
        "5024,20,17,17,38;";


System.out.println(s.solve(init, "AS2", true));
```

In our implementation we represent the state as a String in the following format:

“Food;Materials;Energy;Prosperity;Money_spent;Waiting_for;Remaining_delivery_time”

Where Waiting_for represents the name of the resource the state is waiting its delivery if any and Remaining_delivery_time represents the time left for the delivery.

Main Functions

In this section , we will describe the main functions implemented , how they are implemented and what they output.

ParseInitials function

This function takes as an input the state as a String and Parse it accordingly into previously initialized variables at the creation of search problem.

Solve function

This function is the main search problem function , it takes as an input three inputs as stated before . According to the strategy , it calls the corresponding helper function that the search agent applies. Each helper function will be discussed in the following section as each one represents a search strategy.

canRequestFood function

This function takes as an input an Array of Strings which has the information of the state splitted in this array. What this function does is that it checks if there are enough resources to request a food delivery and checks if the current state is not waiting for a delivery. It outputs a boolean value , as in yes you can request or no you can't.

canRequestMaterials function

This function takes as an input an Array of Strings which has the information of the state splitted in this array. What this function does is that it checks if there are enough resources to request materials delivery and checks if the current state is not waiting for a delivery. It outputs a boolean value , as in yes you can request or no you can’t.

canRequestEnergy function

This function takes as an input an Array of Strings which has the information of the state splitted in this array. What this function does is that it checks if there are enough resources to request Energy and checks if the current state is not waiting for a delivery. It outputs a boolean value , as in yes you can request or no you can’t.

canWait function

This function takes as an input an Array of Strings which has the information of the state splitted in this array. What this function does is that it checks if there is an order requested previously or not . It outputs a boolean value , as in yes you can wait or no you can’t.

canBuild1 function

This function takes as an input an Array of Strings which has the information of the state splitted in this array. What this function does is that it checks if there are enough resources to perform Build1 action It outputs a boolean value , as in yes you can build or no you can’t.

canBuild2 function

This function takes as an input an Array of Strings which has the information of the state splitted in this array. What this function does is that it checks if there are enough resources to perform Build2 action It outputs a boolean value, as in yes you can build or no you can't.

Search Algorithms Implementation

In this section, we will be discussing how each algorithm was implemented.

The Basic Process of all the Algorithms included the following :

- Create an initial node with initial values given
- Create a Stack/Queue/PriorityQueue according to the algorithm being implemented
- Create an ArrayList containing States as Strings only to check for uniqueness
- Insert the initial node first
- Keep extracting nodes as long as the Stack/Queue/PriorityQueue not empty
- For each node expanded , we do the following:
    - We parse the state value into variable for ease of access
    - We check if the current prosperity is >= 100 , if yes we break as goal is reached if not continue
    - We check the waiting_for and remaining_time values, if the current state is waiting for a delivery , we pass a decremented value of it to a variable that will be passed to the new children
    - We check all of the actions available to us from the current state and saving those to boolean variables in order to know what actions can we make
    - For each possible action
        - if the corresponding boolean value is true, we update the resources and the state according to each action's effect
        - We check if the new child has the remaining_time = 0 , which means delivery time so we update the corresponding resource accordingly
        - In the case of Greedy and A* , we perform the heuristic and assign it to a variable and then assign this variable to the node we are creating

    - We create a new Node passing the corresponding values , and passing the parent as the current state/node and passing the depth of the parent + 1
    - We check then if this node's cost is >100000 which is the budget and we check if the new resources will not pass 50 , if both of those conditions are satisfied :
        - We check if the state of this node is present in the ArrayList unique_states , if not then we add the node to the Stack/Queue/PriorityQueue and add the state to unique_states ArrayList to avoid repeated states and redundancy

The above steps were common between all the algorithms what differentiated each algorithm from the other is the inserting and Data structure used for remaining nodes , so in the following we will mention that for each algorithm.

1) **BFS: A Queue is used accomplishing FIFO**
2) **DFS: A Stack is used accomplishing LIFO**
3) **UCS: A PriorityQueue is used comparing the path cost of each Node upon inserting and organizing the queue according to that.**
4) **ID: A Stack is used , but there was an additional structure here , where an outer loop was created to achieve the ID algorithm , This outer loop could be exited if a goal found or if the level we are in is the last/max level meaning there are no additional children**
5) **ASi: A PriorityQueue is used comparing the heuristic (i) + path cost of each Node upon inserting and organizing the queue according to that.**
6) **GRi : A PriorityQueue is used comparing the heuristic (i) of each Node upon inserting and organizing the queue according to that.**

Then, we check if the goal_state is null , which means no goal found we return “NOSOLUTION” String . If a goal was found , we track the solution’s parent and their parents and so on in order to get the plan and sequences of actions leading to this goal. Moreover , we have initialized a variable that increments upon every expansion of any node to save the value of expanded nodes and finally, we get the cost to goal by simply getting the path_cost attribute of the goal node as upon creating any new node , it inherits the cost of its parent and adds to it the action cost.

Heuristic Functions

Heuristic Function 1:

$$\frac{100 - CurrentProsperity}{Min_{build_{cost_{prosperity}}}} * Min_{build_{Price}}$$

Where :

CurrentProsperity: The current prosperity value of the state.

Min_build_cost_prosperity: The prosperity of the BUILD operator with the minimum cost

Min_build_price: The total cost of performing the BUILD operator with the minimum cost.

It has successfully passed the tests and it is an admissible function where we basically calculate the remaining prosperity divided by how much we can gain extra from the minimum BUILD multiplied by the price of that build , It is admissible as the other BUILD will have higher cost so we chose the minimum cost of BUILD to avoid Overestimation.

Heuristic Function 2:

$$\frac{100 - CurrentProsperity}{Current\ Total\ Resources + 1} * Min_{build_{Price}}$$

Where :

CurrentProsperity: The current prosperity value of the state.

Current Total Resources: The sum of all current resources which are the sum of Energy,Food and Materials

Min_build_price: The total cost of performing the BUILD operator with the minimum cost.

It has successfully passed the tests and it is an admissible function where we basically calculate the remaining prosperity divided by how much total resources we have multiplied by the total price of performing the BUILD with the minimum cost, It is admissible because the remaining prosperity can not increase but the resources are able to increase which will result in the number becoming smaller and not possible to get bigger , so there will never be an overestimation. Moreover + 1 was added in order to avoid Exceptions of Dividing by Zero .

Search Algorithms Implementation Comparison

The following comparisons were implemented on tests 0 and 4 as a base to evaluate the differences between the various algorithms at hand.

1. Breadth-First Search (BFS):

   Completeness: BFS is complete when applied to finite state spaces, ensuring that it will find a solution if one exists within the explored state space. However, its completeness may be compromised in infinite state spaces.

   Optimality: If every step cost is the same, BFS ensures optimality. It investigates nodes one at a time, making sure that the initial answer is the simplest and least expensive.

   CPU Time: 46.875 ms (test0), 250.000 ms (test4)

   Memory Usage: 18.88 MB (test0), 5.93 MB (test4)

   Nodes Expanded: 24 nodes (test0), 7272 nodes (test4)

   Analysis: BF explores all neighbors of the initial state before moving on to deeper levels. It's very memory-intensive and expands nodes in a manner that highly depends on branching factor and depth.

2. Depth-First Search (DFS):

   Completeness: DFS is not complete in the general case. It can get stuck in infinite loops or fail to find a solution if the search space is infinite. In finite state spaces, it is complete as it explores all possible paths.

Optimality: Optimality is not guaranteed by DFS. While it might discover a solution fast, it might not always find the cheapest one. The sequence in which nodes are investigated determines the path that is selected.

CPU Time: 0.000 ms (test0), 15.625 ms (test4)

Memory Usage: 6.65 MB (test0), 7.25 MB (test4)

Nodes Expanded: 4 nodes (test0), 381 nodes (test4)

Analysis: DF goes deeply into the search tree. It uses less memory compared to BF but might not find the optimal solution quickly. It explores fewer nodes in shallow searches but more nodes in deeper ones.

3. Iterative Deepening Search (IDS):

   Completeness: IDS is a complete algorithm, guaranteeing that it will find a solution if one exists. It combines the strengths of DFS and BFS, gradually increasing the depth of search until a solution is found.

   Optimality: When it comes to locating the least-cost solution, IDS is ideal. It blends DFS's memory efficiency with BFS's completeness, progressively refining the search until the ideal answer is discovered.

   CPU Time: 15.625 ms (test0), 609.375 ms (test4)

   Memory Usage: 8.26 MB (test0), 14.51 MB (test4)

   Nodes Expanded: 984 nodes (test0), 9766 nodes (test4)

   Analysis: UC expands nodes based on the cost of the path. It balances optimality and memory but can explore more nodes compared to other algorithms due to varying path costs.

4. Uniform Cost Search (UCS):

   Completeness: UCS is complete when applied to finite state spaces with non-decreasing path costs. It explores the least-cost paths first, ensuring optimality within the given constraints.

Optimality: Because UCS looks first at the least-cost pathways, it is ideal. By giving priority to nodes with lower path costs, it ensures that the lowest-cost solution is found.

CPU Time: 0.000 ms (test0), 5359.375 ms (test4)

Memory Usage: 14.71 MB (test0), 141.49 MB (test4)

Nodes Expanded: 34 nodes (test0), 265934 nodes (test4)

Analysis: ID combines advantages of DFS and BFS. It's complete and uses less memory than BFS but can expand a large number of nodes, especially in deeper searches.

5. Greedy Best-First Search:

   Completeness: Greedy Best-First Search is not generally complete. It may fail to find a solution if it gets stuck in local minima due to its myopic approach of selecting the most promising path at each step.

   Optimality: There is no certainty that Greedy Best-First Search is the best option. It may result in less-than-ideal solutions since it only takes into account heuristic data while making decisions, ignoring the total cost of the road.

   GR1 CPU Time: 0.000 ms (test0), 0.000 ms (test4)

   GR1 Memory Usage: 20.44 MB (test0), 21.32 MB (test4)

   GR1 Nodes Expanded: 5 nodes (test0), 561 nodes (test4)

   GR2 CPU Time: 0.000 ms (test0), 15.625 ms (test4)

   GR2 Memory Usage: 21.76 MB (test0), 4.56 MB (test4)

   GR2 Nodes Expanded: 10 nodes (test0), 1270 nodes (test4)

   Analysis: Greedy search uses heuristic information to guide the search. GR2 shows lower memory usage but expanded more nodes in test4.

6. A* Search:

Completeness: A* Search is complete given certain conditions, such as an admissible heuristic. If the heuristic is admissible (never overestimates the true cost), A* is guaranteed to find the optimal solution if one exists.

Optimality: Under some circumstances, namely when employing an acceptable heuristic, A* is the best option. Assuming the heuristic is valid, A* will invariably combine the advantages of greedy best-first search and uniform cost search to discover the best solution.

AS1 CPU Time: 31.250 ms (test0), 546.875 ms (test4)

AS1 Memory Usage: 6.05 MB (test0), 12.65 MB (test4)

AS1 Nodes Expanded: 984 nodes (test0), 9207 nodes (test4)

AS2 CPU Time: 15.625 ms (test0), 671.875 ms (test4)

AS2 Memory Usage: 13.65 MB (test0), 20.45 MB (test4)

AS2 Nodes Expanded: 984 nodes (test0), 9766 nodes (test4)

Analysis: A* combines the advantages of UC and Greedy by using heuristic information. AS2, in particular, shows higher memory usage but expanded fewer nodes compared to other algorithms in test4.

Further elaboration,

Breadth-First Search (BF):

Memory: Uses substantial memory by storing all nodes at each level. Tends to exhaust memory quickly as it explores the entire breadth of the tree before moving deeper.

CPU Time: Generally lower CPU time for shallower searches but increases exponentially with the branching factor and depth.

Depth-First Search (DF):

Memory: Uses less memory compared to BF as it explores deeply before backtracking. However, it can still be memory-intensive for deep searches due to its linear exploration.
CPU Time: Typically faster than BF in reaching deeper solutions but might not find the optimal one quickly.

Uniform Cost Search (UC):
Memory: Balances memory usage and optimality. Explores nodes based on the cost of the path, leading to more efficient memory utilization compared to BF and DF but might still expand a considerable number of nodes.
CPU Time: Generally slower than DF and BF due to the added cost calculation but tends to find optimal solutions.
Iterative Deepening (ID):

Memory: Uses less memory than BF but sacrifices memory for completeness. Clears the explored nodes after each depth iteration, making it less memory-intensive than BF but still expanding a large number of nodes.
CPU Time: Increases significantly with depth due to repeated depth-limited searches but is still complete and finds the optimal solution in the search space.

Greedy Search (GR1 and GR2):
Memory: Depends highly on the heuristic used. GR2, with a more informed heuristic, exhibits lower memory usage but might expand more nodes than GR1, which focuses solely on heuristic-guided exploration.
CPU Time: Usually faster than UC and ID due to the heuristic's influence, but may not guarantee an optimal solution.

A* Search (AS1 and AS2):

Memory: Combines characteristics of UC and Greedy approaches, employing an informed heuristic for more efficient exploration. AS2, though slightly higher in memory usage, expands fewer nodes than AS1 in test4.

CPU Time: Shows a balance between UC and Greedy approaches, utilizing heuristic information to guide the search and often finding optimal solutions.

In conclusion, the differences in memory usage and CPU time among these strategies align with their inherent trade-offs. Algorithms like BF and UC prioritize optimality, which leads to higher memory usage but potentially optimal solutions. Conversely, DF, GR, and some variations of A* sacrifice optimality for speed and memory efficiency but may not always find the best solution. ID strikes a balance between optimality and memory, making it suitable for certain scenarios where memory is a concern but the optimal solution is crucial.

The code runs without any errors and all tests in LLAPPublicGrading file passed successfully.