Project 1: Live Long and Prosper

Project report

November 14, 2023

1 Problem Description

A town is in need of essential resources (food, materials, and energy) to support its citizens and establish new buildings. The town's prosperity level reflects the well-being of its residents, and the key to enhancing prosperity lies in strategically establishing buildings that require specific resources. The goal of this project is to design a search agent tasked with finding a plan to elevate the town's prosperity level to **100**. The search agent operates within a budget of **100,000** units and must carefully manage the available resources. There is no additional source of income beyond the initial budget, and the town has a storage limit of **50 units** per resource. Resources are consumed with each action taken by the agent, whether it be establishing buildings or requesting deliveries. The agent must consider the varying impacts of different actions on both prosperity and resources. In addition, using resources requires spending money, which is what these resources cost.
The agent's objective is to search for a plan, if available, that can lead to achieving the target prosperity level, based on the initial state description and a set of potential actions.

2 Search Tree Node

Search tree nodes serve as representations of the state space of a problem. Each node encapsulates a set of variables that mirrors the state of the problem at that specific node. Each node consists of 5 main components:

1. **Node parent**: which is the parent of the current node.
2. **String operator**: This represents the operator applied to generate this node. In our problem, an operator could be one of these: **RequestFood**, **RequestMaterials**, **RequestEnergy**, **WAIT**, **BUILD1**, **BUILD2**.
3. **int depth**: which is the depth of the current node.
4. **int pathCost**: from the root, In our problem, The path cost is the total money spent from the root to the current node.
5. **State state**: A current state that includes the necessary variables representing the node, In our problem, The state keeps track of this information:

- **prosperity**: The current prosperity.
- **food**: The current amount of food.
- **materials**: The current amount of materials.
- **currBudget**: The current budget.
- **delay**: The remaining delay for delivery and has a maximum value of 2.
- **pendingType**: The type of resource for which this state is pending, represented by an Integer value from 0 to 3. Here, 0 indicates the idle state, 1 corresponds to food, 2 to materials, and 3 to energy.
- **h**: The heuristic value from the current node to the goal, changes depending on the strategy.

3 Generic Search Problem

In this class, we implemented the general search procedure, which takes a search problem and a strategy as input and returns a solution if it exists. In addition, abstract functions such as isGoalTest, getInitalState, and method to retrieve operators of the problem (getOperators) and method to expand etc. and They are discussed below.

3.1 Main Functions

1. `public String generalSearch(String problem , String strategy){}`

   - This function takes a string representing the search problem and a strategy as input. Depending on the specified strategy, the function implements the assigned search technique.
   - In each of the following algorithms, we handle repeated states by utilizing a HashSet of visited nodes. Within each algorithm, we check in the loop whether the node has been added to the HashSet before. If it has, we continue looping without expanding; otherwise, we add it to the HashSet.

   The following functions discuss how we implement the different search strategies.

2. `public String BFS(Node root) {}`

   - This function implements the breadth-first search strategy, taking the root node as input, We utilize a **Queue** to store nodes. Initially, we add the root node, The function iterates through a loop examining the nodes at each level first, At any point if the queue is empty, indicating there is no solution, in which case it returns **"No solution"**. Otherwise, it removes a node from the front of the queue (the current node), checks if it is a goal test then returns it if not expands it using operators (discussed in Section 2), and adds its children to the end of the queue.
   - The order of node arrangement in the queue is determined by the order of operators, In our implementation, requests are enqueued first then wait then builds.

3. `public String DFS(Node root) {}`

   - This function implements the Depth-first search strategy, taking the root node as input. We utilize a **Stack** to store nodes. Initially, we push the root node, The function iterates through a loop exploring as far as possible along each branch before backtracking, At any point if the queue is empty, indicating there is no solution, in which case it returns **"No solution"**. Otherwise, it pops a node from the top of the stack (the current node), checks if it is a goal test then returns it if not expands it using operators (discussed in Section 2), and pushes its children to the top of the stack.
   - The order of node arrangement in the stack is determined by the order of operators. In our implementation, builds are placed at the top of the stack so that they are the first to be popped then the wait then the requests.

4. `public String UCS(Node root) {}`

   - This function implements the Uniform-Cost search strategy, taking the root node as input, We utilize a **PriorityQueue** to store nodes. Initially, we add the root node, The function iterates through a loop, At any point if the priority queue is empty, indicating there is no solution, in which case it returns **"No solution"**. Otherwise, it removes a node from the front of the priority queue (the current node), checks if it is a goal test then returns it if not expands it using operators (discussed in Section 2), and adds its children to the end of the queue.
   - The arrangement of added nodes in the priority queue is implemented in a way that caters to three main search strategies: uniform cost, greedy, and A*. where, in the case of **uniform cost**, nodes are arranged in terms of their pathCost (i.e., money spent), which favours nodes with a lower pathCost to the front of the priority queue. In the case of the **greedy search**, nodes are arranged in terms of their heuristic values. Last, In the case of the **A* search**, nodes are arranged in terms of their heuristic values in addition to the pathCost. Essentially, the UCS function serves as an implementation that encapsulates the three search strategies within the same function.

5.
   ```
   public String IDS(Node root) {}
   public String depthLimited(Node root, int depth) {}
   ```

   - The following functions implement Iterative Deepening Search, a combination of DFS and BFS. The loop iterates from depth zero to the maximum possible depth. In our problem, the worst possible branch depth is determined by the initial budget divided by the consumption cost. At each depth, we perform a depth-limited search, essentially a regular DFS with the condition that if the depth of the current node exceeds the current depth level, we backtrack.

3.2 Helper Functions

These are abstract functions (discussed in Section 4) that are tailored to the specific problem at hand. They are initially declared as abstract functions within a generic search class. The body of these functions is then implemented in the LLAP search class, as their specifics are unique to each problem.

```
public abstract Node getInitialState(String problem,String strategy);
public abstract String [] getOperators()
public abstract boolean isGoalTest(Node node);
public abstract ArrayList<Node> expand(Node node);
public abstract String getPrint(Node node);
public abstract String getPlan(Node node);
public abstract int getMaxDepth();
public abstract boolean isVisual();
public abstract void incrementTotalNum();
```

4 LLAP Problem

The LLAPSearch class serves as the main class for our problem, extending the GenericSearch class. Within this class, we define initial values specific to our problem and extend functions inherited from the superclass. Additionally, a static solve method is implemented, which executes the tests.

4.1 Main Functions

1.
   ```
   public static String solve(String initialState
   ,String strategy, boolean visualize){}
   ```

   - This function utilizes a search algorithm to discover a sequence of steps enabling the town to achieve prosperity, provided such a sequence exists. The function takes a string representing the initial state of the problem, a string indicating the strategy, and a boolean value "visualize," which results in side-effecting display of state information. Initially, the function invokes the interpreter() (see 2), which parses the initial state string into the variables of the problem. Subsequently, the function calls the general search method with the problem and the specified strategy.

2.
   ```
   private static void interpreter(String initialState){}
   ```

   - This function accepts a string, initialState, which the interpreter parses by splitting it at [,;]. It then initializes the constants of the problem using the values obtained from each index as described in the string.

3.
   ```
   public boolean isGoalTest (Node node ){}
   ```

   - This function is an extension of the abstract method in the superclass GenericSearch and has been customized in the LLAPSearch class to suit the unique requirements of our problem. In this context, a node is considered a goal node if the prosperity of the input node is greater than or equal to 100.

4.
   ```
   public Node getInitialState(String problem,String strategy) {}
   ```

   - This function is an extension of the abstract method in the superclass GenericSearch takes a string as input and returns a created node object with the parameter initialized from the interpreter, which is the root of the problem.

5. `public ArrayList<Node> expand(Node currNode){}`

   - This function is an extension of the abstract method in the superclass GenericSearch. In this function, a node is taken as input and expanded with the possible operators (discussed in Section 2). For each operator, if it is feasible to perform at this node, a child node is created and added to an ArrayList. Finally, the function returns an ArrayList of nodes, which is then added to the storage unit of the utilized search strategy.

6. `private static Node handleBuild(Node currNode, State currState,int idx){}`

   - If a build is possible, the function updates the state of the node with consequences on money, prosperity, and resources.

7. `private static void handleDelivery(State currState){}`

   - if a delivery arrives, the function updates the corresponding resource to the request of the delivery based on the pendingType value.

8. `public static int getH(State s){}`

   - The function calculates the heuristic value of the current node based on the chosen heuristic function (either one or two).

9.
   ```
   public String getPrint(Node node){}
   public String getPlan(Node node){}
   ```

   - This function takes the goal node as input and backtracks to the root to obtain the plan leading to the solution.

10.
    ```
    public int getMaxDepth() {}
    public boolean isVisual() {}
    public void incrementTotalNum() {}
    ```

4.2 Constants

In addition to the main functions, we maintain a set of constant values that are initialized at the beginning of the run through the 'interpreter()'. These constants serve as parameters used by the main functions.
* initialProsperity, initialFood, initialMaterials, initialEnergy, unitPriceFood, unitPriceMaterials, unitPriceEnergy, amountRequestFood, amountRequestMaterials, amountRequestEnergy , delayRequestFood, delayRequestMaterials, delayRequestEnergy, consumptionCost, priceBuild [], foodUseBuild [], materialsUseBuild [], energyUseBuild [], prosperityBuild []

5 Heuristic Functions

5.1 Heuristic 1

Math Formula

$h(n) = \min\left(\frac{\text{BUILDCOST1}}{\text{prosberityBUILD1}}, \frac{\text{BUILDCOST2}}{\text{prosberityBUILD2}}\right) \cdot \max(0, 100 - \text{prosb}(n))$ where:

- $h(n)$: The heuristic value of node n.
- BUILDCOST1 and BUILDCOST2: the cost of each build which is the sum of the price of the build in addition to the price of the resources consumed.
- prosberityBUILD1 and prosberityBUILD2: The increase in prosperity resulting from BUILDi.
- prosb(n) is a function representing the level of prosperity at this node n.

Discussion
In this heuristic, we estimate the cost remaining from the current node to the goal by calculating the cost of one unit of prosperity for each build, take the lower cost and then multiply this cost by the remaining prosperity from the current node to the goal (i.e., where prosb=100).
To get this heuristic, we relaxed the problem so that we only considered the restrictions made by the cost of building, with no restrictions on the requests or waiting. This heuristic is admissible because it only considers the minimum building cost without the cost of waiting or requesting, which is correct because you can't reach a goal state without doing at least this build sequence so it always gives a value less than or equal to the real cost.

5.2 Heuristic 2

Math Formula

$h(n) = \min\left(\frac{\text{BUILDCOST1}+(\text{NeededRequests1}*\text{consumptionCOST})}{\text{prosberityBUILD1}}, \frac{\text{BUILDCOST2}+(\text{NeededRequests2}*\text{consumptionCOST})}{\text{prosberityBUILD2}}\right) \cdot$
$\max(0, 100 - \text{prosb}(n))$
where:

- $h(n)$: The heuristic value of node n.
- BUILDCOST1 and BUILDCOST2: the cost of each build which is the sum of the price of the build in addition to the price of the resources consumed.
- prosberityBUILD1 and prosberityBUILD2: The increase in prosperity resulting from BUILDi.
- NeededRequests1 and NeededRequests2: The number of requests needed to make BUILD1 or BUILD2, considering the amount of resources at node n, the amount of resources needed by each build, and the amount of resources given by the delivery

of each request, can be expressed as follows:
Number of requests needed $= \max\left(0, \left\lceil \frac{\text{buildResources}-\text{Resources at node } n}{\text{deliveryResources}} \right\rceil\right)$

- consumptionCOST(n) The cost of the consumption of resources at each step, which is the sum of food price, material price, and energy price.

- prosb(n) is a function representing the level of prosperity at this node n.

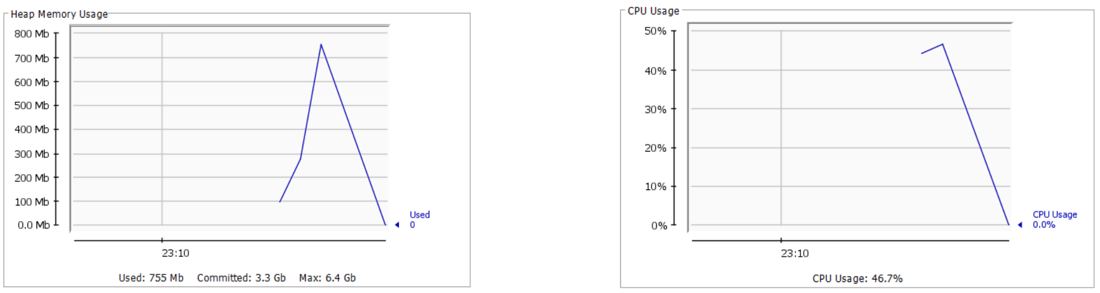
Discussion
Similar to the first heuristic, this one calculates the minimum cost for a prosperity unit but with a better estimate because it considers the cost of the required food, material and energy requests along with the cost of the build, leading to a less-relaxed version of the problem.
This heuristic is admissible because it doesn't consider the cost of waiting and only considers the cost of building and requesting the resources required for building which is definitely less than the actual cost to the goal. This heuristic dominates the first one because it considers the cost of resources required for building.
So at any node h1(n) $\leq$ h2(n) $\leq$ h*(n).

6 Performance Comparison

Measuring the performance of the algorithms was done by the Jconsole tool from the JDK. It is a GUI that shows monitoring information for the chosen running process. Performance for each search algorithm is measured by its RAM usage and CPU utilization for all of the 11 test cases.

6.1 BFS

all the 11 test cases were completed in 12.754 seconds, consuming 755Mb and 46.7% of the CPU.

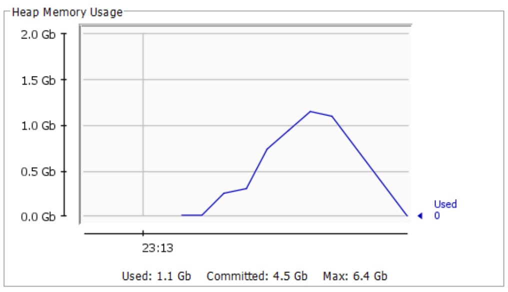


(a) RAM usage (b) CPU utilization

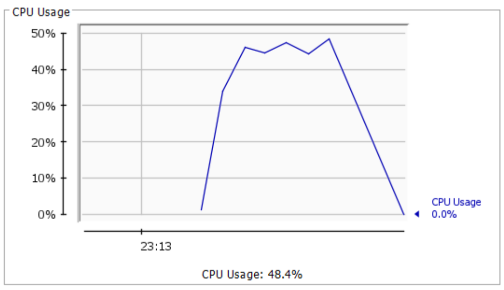
Figure 1: Stats for BFS

6.2 DFS

all the 11 test cases were completed in 33.679 seconds, consuming 1.1Gb and 48.4% of the CPU.



(a) RAM usage

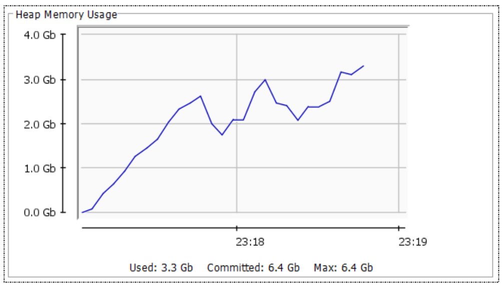


(b) CPU utilization

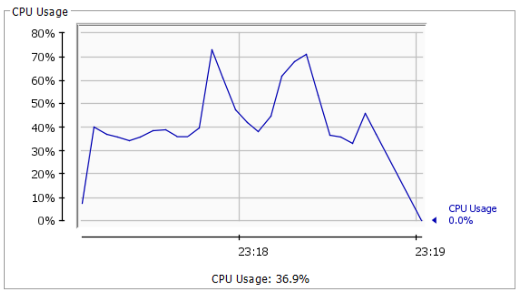
Figure 2: Stats for DFS

6.3 UCS

all the 11 test cases were completed in 1 minute, consuming 3.3Gb and 36.9% of the CPU.



(a) RAM usage

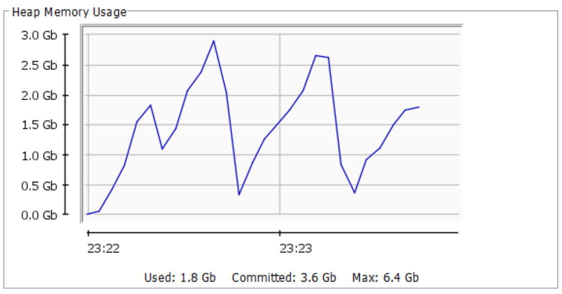


(b) CPU utilization

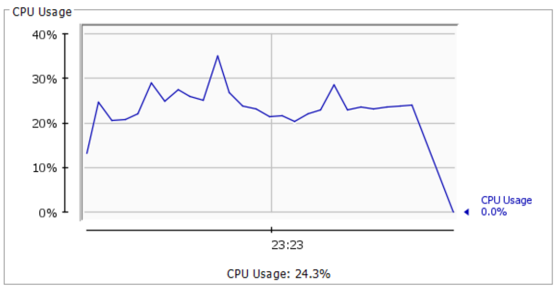
Figure 3: Stats for UCS

6.4 IDS

all the 11 test cases were completed in 1 minute, consuming 1.8Gb and 24.3% of the CPU.



(a) RAM usage

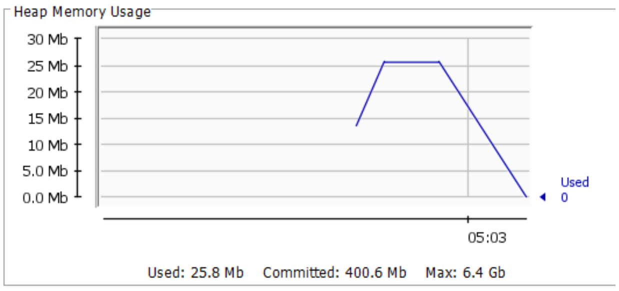


(b) CPU utilization

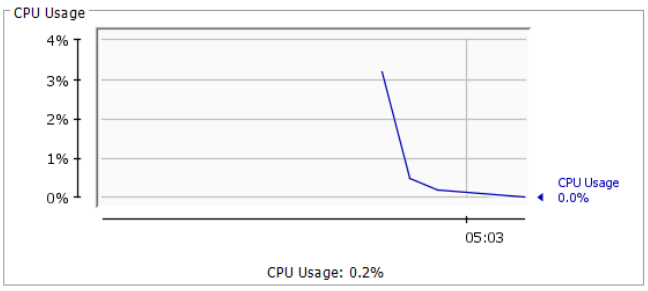
Figure 4: Stats for IDS

6.5 GR1

all the 11 test cases were completed in 2.843 seconds, consuming 25.8Mb and 0.2% of the CPU.



(a) RAM usage

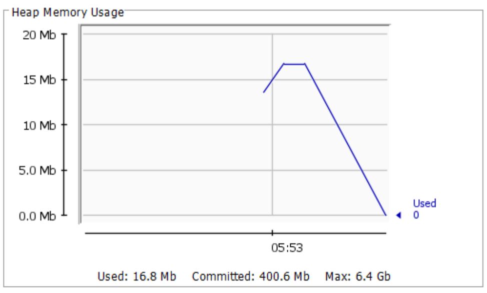


(b) CPU utilization

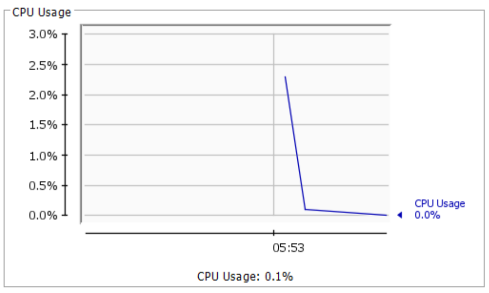
Figure 5: Stats for GR1

6.6 GR2

all the 11 test cases were completed in 2.5 seconds, consuming 16.8Mb and 0.1% of the CPU.



(a) RAM usage

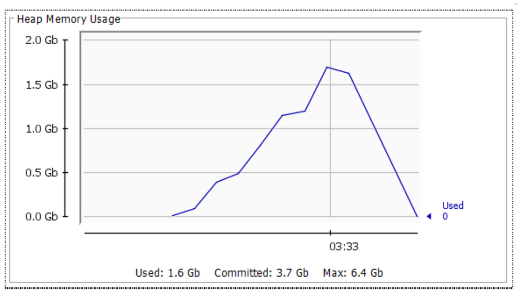


(b) CPU utilization

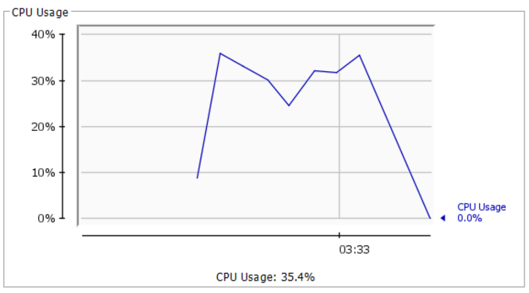
Figure 6: Stats for GR2

6.7 AS1

all the 11 test cases were completed in 39.315 seconds, consuming 1.6Gb and 35.4% of the CPU.



(a) RAM usage

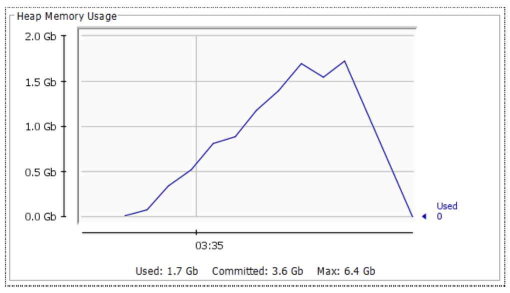


(b) CPU utilization

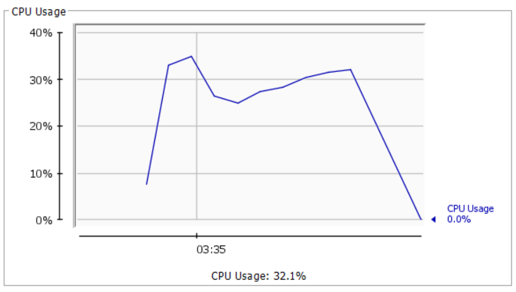
Figure 7: Stats for AS1

6.8 AS2

all the 11 test cases were completed in 47.174 seconds, consuming 1.7Gb and 32.1% of the CPU.



(a) RAM usage



(b) CPU utilization

Figure 8: Stats for AS2

6.9 Summary

the following table shows a comparison between different strategy measures. The expanded nodes are calculated on **test 10**. It can be noticed from the previous table the

Strategy	**RAM**	**CPU**	**expanded Nodes**	**cost**
BFS	755Mb	46.7%	533644	21944
DFS	1.1Gb	**48.4%**	6339051	99764
UCS	**3.3Gb**	36.9%	**6703526**	**13652**
ID	1.8Gb	24.3%	4989694	21944
GR1	25.8Mb	0.2%	28991	57682
GR2	**16.8Mb**	**0.1%**	**291**	22152
AS1	1.6Gb	35.4%	2053000	**13652**
AS2	1.7Gb	32.1%	1829766	**13652**

Table 1: Strategy comparison table

following:

- UCS is performing the worst in terms of RAM usage and the number of expanded nodes, but it finds an optimal solution. The algorithm is also complete because the branching factor in our problem is finite, and there are no paths with negative cost values.
- BFS is using a considerably low RAM with a relatively few number of expanded nodes. However, it does not find an optimal solution, as it returns the closest solution to the root, and the algorithm is considered complete since the branching factor in our problem is finite.
- DFS has the highest CPU usage as it expands a huge number of nodes and finds the worst possible solutions, In terms of optimality, the algorithm does not discover an optimal solution. However, in terms of completeness, the algorithm is complete, provided that our search tree is finite. This assumption holds under the conditions that the build cost and the build prosperity are always greater than zero, so it does not enter an infinite branch as the branch terminates whenever the current budget is exhausted or the resource is depleted.
- IDS optimizes the DFS dramatically as it uses less CPU and expands much fewer nodes while finding a better, less costly solution. The algorithm is complete because the branching factor in our problem is finite.
- GR1 and GR2 are extremely fast with the fewest RAM and expanded nodes. However, they do not find an optimal solution. However, in terms of completeness, the algorithm is complete, provided that our search tree is finite. This assumption holds under the conditions that the build cost and the build prosperity are always greater than zero
- Both AS1 and AS2 find the optimal solution, but since h2 dominates h1, it can be noticed that AS2 is performing better than AS1 on all measures. also, in terms of completeness, the algorithm is complete, provided that our search tree is finite. This

assumption holds under the conditions that the build cost and the build prosperity are always greater than zero