Live-Long-Prosper-Agent

Problem Overview

The problem involves designing a search agent for a town that aims to reach a prosperity level of 100. The town requires resources such as food, materials, and energy to achieve this goal. The agent has a budget of 100,000 to spend, and there's no additional source of income. The town has a limit for the amount of resources it can store (50 units per resource), and resources deplete with every action the agent takes.

The agent can perform various actions, including requesting deliveries of food, materials, and energy, waiting for deliveries to arrive, and building structures (BUILD1 and BUILD2) that contribute to the town's prosperity level. Each action has associated parameters, such as the amount and delay for resource deliveries, and different resource consumption, cost, and prosperity increase for building structures.

The challenge is to find a plan that optimally utilizes the budget, manages resource levels effectively, and makes strategic decisions on when to request deliveries, wait, or build structures. The agent needs to consider the time delays for resource deliveries and ensure that the town's resources, money, and prosperity level are maintained above zero to continue taking actions. The overall objective is to formulate a sequence of actions that lead to the town reaching a prosperity level of 100 within the given constraints.

Search Tree Node ADT

The `SearchTreeNode` abstract data type (ADT) is designed to represent nodes in the search tree. Each node has the following attributes:

- **State (`state`):** Represents the current state of the problem, including information about prosperity, food, materials, and energy levels.
- **Parent Node (`parentNode`):** A reference to the node in the search tree that this node was generated from.
- **Operator (`operator`):** Represents the action that was applied to the parent node to generate this node.
- **Depth (`depth`):** The depth of this node in the search tree.
- **Path Cost (`pathCost`):** The total cost of the path from the initial state to this node, calculated based on the costs of the actions that were applied.
- **Heuristic Values (`heuristicValue1` , `heuristicValue2`):** Estimates of the cost from this state to a goal state. Used by some search algorithms to guide the search towards the goal efficiently.

The class includes getter and setter methods for each of these attributes and a `toString` method that returns a string representation of the node.

Search Problem ADT

The `SearchProblem` ADT represents a specific search problem. It includes the following attributes:

- **Operators (`operators`):** A list of all possible actions that can be applied to a state in the problem. Each operator is an instance of the `Operator` class, defining an action and how it transitions from one state to another state.
- **Initial State (`initialState`):** Represents the starting point of the problem. An instance of the `State` class encapsulates the state of the problem at a given point.
- **Goal Test (`goalTest`):** The value defining the goal state of the problem. In this case, it's a fixed value of 100 for prosperity. A state is considered a goal state if some property of the state equals this value and another value of the money spent.

The class includes getter and setter methods for each of these attributes.

LLAPSearch Problem

The `LLAPSearch` class extends the `GenericSearch` class and represents a specific search problem. The constructor takes a `Problem` and a `SearchStrategy` as arguments, which are passed to the constructor of the superclass, `GenericSearch` .

Methods

1. **`solve(String initialState, String searchStrategy, boolean visualize)`** :
    - This static method is the main entry point for the problem-solving process. It takes three parameters: the initial state of the problem (`initialState`), the chosen search strategy (`searchStrategy`), and a boolean flag indicating whether to visualize the search (`visualize`). The method leverages the `GenericSearch` class, encapsulating the problem and search algorithm. After obtaining the goal node through the search, the method returns a formatted string containing the sequence of actions, money spent, and the number of nodes expanded during the search.
2. **`SolveGenericSearch()`** :
    - This method, encapsulated within the `GenericSearch` class, initiates the problem-solving process based on the chosen search strategy. It calculates the minimum cost per prosperity level, the maximum gain from either BUILD action, and the current prosperity level. Using these values, it constructs an initial node and invokes the appropriate search method based on the selected strategy (BF, DF, ID, UC, GR1, GR2, AS1, AS2). The selected search method returns the goal node, or null if no solution is found.
3. **`expand(Node current)`** :
    - The `expand` method generates successor nodes for a given node, representing possible actions that can be taken from the current state. It invokes action methods (`build1` , `build2` , `requestEnergy` , `requestMaterials` , `requestFood` , `wait`) defined in the `Actions` class. The resulting nodes are added to a linked list and returned for further exploration.
4. **`goalTest(Node testNode)`** :
    - The `goalTest` method checks whether a given node satisfies the goal conditions. Specifically, it evaluates if the current prosperity level is greater than or equal to 100, and the money spent is less than or equal to 100,000. If the conditions are met, the method returns true, indicating that the goal state has been reached.

Action Methods

5. **`requestFood(Node input, boolean visualize)`**:
    - This method represents the action of requesting food resources. It checks the feasibility of the action based on delays and resource availability. If the action is valid, it creates a new state and node reflecting the changes after the request.
6. **`requestMaterials(Node input, boolean visualize)`**:
    - Similar to the `requestFood` method, this one represents the action of requesting materials. It checks constraints, updates the state, and creates a new node.
7. **`requestEnergy(Node input, boolean visualize)`**:
    - The action of requesting energy resources is implemented here, adhering to the defined rules. If valid, it generates a new state and node reflecting the changes.
8. **`wait(Node input, boolean visualize)`**:
    - The waiting action is captured in this method. It checks if waiting is allowed based on delays and then updates the state and creates a new node accordingly.
9. **`build1(Node input, boolean visualize)`**:
    - This method simulates the action of building with type 1 resources. It checks constraints, updates the state, and creates a new node if the action is valid.
10. **`build2(Node input, boolean visualize)`**:
    - Similar to `build1`, this method handles the action of building with type 2 resources, ensuring the rules are followed and generating a new state and node if the action is feasible.

`RequestFood` Logic:

- Similar logic applied for RequestMaterial And RequestMaterial.

1. **Delay Check:**
    - Check if delays for obtaining food, energy, or materials are non-zero. If any delay is present, return `null` to indicate that the request for food cannot be fulfilled.
2. **Food Limit Check:**
    - Check if the current food level is already at its maximum (50). If so, return `null` to signify that no further food request is needed.
3. **State Copying:**
    - Create a new state (`newState`) by copying the input state (`input.getState()`).
4. **Money Spent Calculation:**
    - Calculate the amount of money spent during the food request process based on unit prices for energy, food, and materials. Update the total money spent so far.
5. **Money Limit Check:**
    - Check if the total money spent so far exceeds 100,000 units. If this limit is reached, return `null` to halt the food request.
6. **Resource Consumption:**
    - Simulate the consumption of energy, food, and materials by decrementing the corresponding values in the state by 1.
7. **Negative Resource Check:**
    - Check if any of the resource levels (energy, food, materials) have fallen below zero. If this happens, return `null` to indicate that the food request cannot be fulfilled.
8. **Delay Update:**
    - Set the delay for food request (`setCurrentFoodDelay`) based on the delay value specified in the state (`getDelayRequestFood`).
9. **Heuristics Calculation:**
    - Calculate various metrics related to prosperity, spending, and efficiency:
        - `prosperityGainBuild1` and `prosperityGainBuild2` : Prosperity gains for two types of builds.
        - `totalSpentBuild1` and `totalSpentBuild2` : Total spending for two types of builds.
        - `minCostPerProsperityLevel` : Minimum cost per prosperity gain, considering both types of builds.
        - `maxGainBuild` : Maximum prosperity gain among the two types of builds.
        - `currentProsperityLevel` : Current prosperity level in the new state.
10. **Node Creation:**

- Create a new child node (`newNode`) with the updated state, referencing the input node, using the `RequestFood` operator. Include information such as depth, total money spent, efficiency metrics, and other relevant details.

`WAIT` Logic:

1. **State Copying:**
    - A new state (`newState`) is created by copying the input state (`input.getState()`).
2. **Checking Delays:**
    - Check if delays for obtaining food, energy, and materials are all zero. If so, return `null` to indicate that no further waiting is necessary.
3. **Money Spent Calculation:**
    - If delays are present, calculate the amount of money spent during the waiting period based on unit prices for energy, food, and materials. Add this cost to the total money spent so far.
4. **Money Limit Check:**
    - Check if the total money spent so far exceeds 100,000 units. If this limit is reached, return `null` indicating that the operation cannot be performed.
5. **Updating Delays:**
    - Decrement the delays for obtaining food, energy, and materials by 1, simulating the passage of time.
    - If delay reached 0, we update the corresponding resource with delivery amount requested. also if total becomes greater than 50, it is set back to 50.
6. **Resource Consumption:**
    - Simulate consumption of energy, food, and materials by decrementing the corresponding values in the state by 1.
7. **Negative Resource Check:**
    - Check if any of the resource levels (energy, food, materials) have fallen below zero. If this happens, return `null` indicating that the operation cannot be performed.
8. **Calculate Heuristics for the New Node:**
    - Calculate various metrics such as prosperity gain for both types of structures, total spent for both structures, minimum cost per prosperity gain, and maximum gain among the two structures.
    - Create a new child node (`newNode`) with the updated state, and other information, representing the result of the wait operation.

`BUILD1` Logic:

- Similar logic applied for BUILD2

1. **Copy the Current(Parent) State:**
    - A new state (`newState`) is created by copying the input state (`input.getState()`).
2. **Calculate Money Spent:**
    - Calculate the money spent on energy, food, and materials based on the usage(resources consumed) and unit prices.
    - Update the total money spent by adding the costs of the structure and resource usage.
3. **Check Budget Limit:**
    - If the total money spent exceeds a budget limit (100,000 in this case), return `null` indicating that the operation cannot be performed.
4. **Update Resource Delays:**
    - Decrease the delay for energy, food, and materials resources if there any.
    - If delay reached 0, we update the corresponding resource with delivery amount requested. also if total becomes greater than 50, it is set back to 50.
5. **Update Prosperity and Resource Levels:**
    - Update the current prosperity level based on the corresponding gain from building the structure.
    - Deduct the corresponding resources used for building from the current resource levels (energy, food, and materials).
6. **Check Resource Constraints:**
    - If any resource (energy, food, or materials) becomes negative, return `null` indicating that the operation cannot be performed.
7. **Calculate Heuristics for the New Node:**
    - Calculate various metrics such as prosperity gain for both types of structures, total spent for both structures, minimum cost per prosperity gain, and maximum gain among the two structures.
    - Create a new child node (`newNode`) with the updated state, and other information, representing the result of the build operation.

Search Strategies

1. **Breadth-First Search (BF) and Depth-First Search (DF):**
    - The `BF_DFSearch` method utilizes a queue to perform Breadth-First or Depth-First search based on the chosen strategy. It expands nodes in the order determined by the respective strategy and continues until the goal state is reached or all nodes are explored.
2. **Iterative Deepening (ID):**
    - The `IDSearch` method implements Iterative Deepening, gradually increasing the depth limit until a solution is found using the same concept used in DFS.

3. **Uniform Cost Search (UC):**
    - The `uniformCostSearch` method employs a priority queue, prioritizing nodes with lower path costs. This ensures that the algorithm explores paths with lower accumulated costs first, leading to an optimal solution.
4. **Greedy Search (GR1 and GR2):**
    - Both `greedySearch1` and `greedySearch2` methods utilize priority queues with different heuristics. They prioritize nodes based on their heuristic values, aiming for solutions that seem promising according to the chosen heuristic.
5. **A* Search (AS1 and AS2):**
    - Similar to greedy search, the `aStar1` and `aStar2` methods use priority queues. However, the priority is determined by the sum of the path cost and the heuristic value, ensuring a balance between cost-effectiveness and heuristic guidance.

Heuristic Functions

Heuristic Function 1:

h1(n)=(100-current prosperity level of n)/(max prosperity levels that can be produced from an action) it is admissible because we are estimating how many levels needed until we reach a goal which is 100 and dividing it by the maximum increase in levels so that it will give us the minimum number of actions required to reach the goal which of course is less than the actual path cost in which action has a positive cost

-This heuristic is admissible because it estimates how many levels are needed until we reach a goal of 100 and divides it by the maximum increase in levels. This provides the minimum number of actions required to reach the goal, which is less than the actual path cost where an action has a positive cost.

Heuristic Function 2:

h2(n): 1- Calculate the cost per prosperity point for each building action: This can be done by dividing the cost of each building action (including the price of the action and the cost of the resources it consumes) by the prosperity increase it provides.

2-Find the minimum cost per prosperity point: The minimum cost per prosperity point is the lowest cost calculated in the previous step.

3-Estimate the cost to reach the target prosperity level: Multiply the difference between the target prosperity level (100) and the current prosperity level by the minimum cost per prosperity point.

This heuristic is admissible because it never overestimates the cost of reaching the goal. It assumes that all future actions will be as cost-efficient as the action with the minimum cost per prosperity point.

Performance Comparison

The performance of the algorithms for two different initial states is shown below:

Initial State 0 Observations:

Algorithm	Path Cost	Expanded Nodes	CPU Utilization (%)	Total Time (ms)	Used Memory (MB)	Remarks
DF	2943	4	6.51	53	5.76	Moderate expanded nodes, low CPU, and memory usage. Lightweight but may not guarantee optimality.
BF	2943	25	7.44	54	5.76	High expanded nodes, moderate CPU, and memory usage. Thorough but resource-intensive.
ID	2943	35	7.9	60	5.76	Moderate expanded nodes, moderate CPU, and memory usage. Combines advantages of DFS and BFS.
UC	2943	94376	16.62	884	105.68	High expanded nodes, high CPU, and memory usage. Explores based on path cost, substantial memory usage.
GR1 (Greedy)	2943	5	10.02	59	5.76	Low expanded nodes, moderate CPU, and low memory usage. Efficient but not guaranteed optimal.
GR2 (Greedy)	20198	502	8.41	81	6.88	Moderate expanded nodes, moderate CPU, and memory usage. More exploration than GR1.
AS1 (A* Search)	2943	94413	18.17	781	110.58	High expanded nodes, high CPU, and memory usage. Considers both path cost and heuristic, demanding.
AS2 (A* Search)	2943	94376	15.31	643	103.41	High expanded nodes, moderate CPU, and high memory usage. Another heuristic, similar characteristics.

Initial State 4 Observations:

Algorithm	Path Cost	Expanded Nodes	CPU Utilization (%)	Total Time (ms)	Used Memory (MB)	Remarks
DF	58652	459	6.85	84	6.88	Moderate expanded nodes, low CPU, and moderate memory usage. More nodes explored, slightly higher memory usage.
BF	29402	673003	8.4	1552	225	Very high expanded nodes, moderate CPU, and very high memory usage. Extremely high number of nodes, highly memory-intensive.
ID	29402	8246906	9.66	11845	514	Very high expanded nodes, moderate CPU, and high memory usage. Extensive node exploration, better memory efficiency compared to BFS.
UC	29402	1305994	7.67	4329	613	High expanded nodes, moderate CPU, and high memory usage. Explores based on cost, high memory usage.
GR1 (Greedy)	61395	1355	7.44	113	7.76	Moderate expanded nodes, moderate CPU, and low memory usage. Consistently efficient with a low number of nodes explored.
GR2 (Greedy)	99906	16359	8.41	178	8.94	Moderate expanded nodes, low CPU, and moderate memory usage. More exploration than GR1 but remains efficient.
AS1 (A* Search)	29402	1305994	11.83	4416	627	High expanded nodes, moderate CPU, and high memory usage. Similar characteristics to the first set, demanding.
AS2 (A* Search)	29402	1305994	4.65	4035	591	High expanded nodes, low CPU, and high memory usage. Lower CPU utilization but similar memory usage.

These observations provide insights into the performance of each algorithm in both initial states, considering various metrics.

General Observations:

- **Number of Expanded Nodes:**
  - BFS tends to expand an exceptionally large number of nodes.
  - Greedy approaches (GR1 and GR2) are more efficient in terms of the number of nodes expanded.
- **CPU Utilization:**
  - A* searches (AS1 and AS2) and UC have higher CPU utilization
  - DFS and Greedy approaches have lower CPU utilization.
- **Used Memory:**
  - BFS consistently uses a significant amount of memory.
  - IDS maintains better memory efficiency compared to BFS.
  - A* and UC tend to use higher memory due to their exploration strategies based on path cost and heuristic values.

Optimality And Completeness

1. **Depth-First Search (DFS):** DFS is not complete in infinite state spaces or in spaces with loops. It is also not optimal as it does not guarantee that the solution found is the best one.
2. **Breadth-First Search (BFS):** BFS is complete, meaning it will find a solution if one exists. It is also optimal as long as the costs of all edges are equal.
3. **Iterative Deepening Depth-First Search (IDS):** IDS is complete in finite state spaces. It is also optimal as long as the costs of all edges are equal.
4. **Uniform Cost Search (UCS):** Uniform Cost Search (UCS) is complete. This means that if a solution exists, UCS will find it. The completeness is guaranteed as long as the cost of every step is greater than some positive constant, ensuring that UCS will eventually explore all possibilities. UCS is optimal. It guarantees that the solution found is the cheapest among all possible solutions. This is because UCS explores paths in order of their total cost, always choosing the path with the lowest cost first.
5. **Greedy Best-First Search (GR1 and GR2):** Greedy Search is not guaranteed to be complete. It can get stuck in infinite loops or fail to find a solution even if one exists. This is because it always makes the locally optimal choice, which may not lead to a globally optimal solution. Also it is not guaranteed to find the most optimal solution. Due to its myopic nature (choosing the best immediate option), it may miss out on a better overall solution if a better choice appears later in the search.
6. **A* Search (AS1 and AS2):** A* is complete and optimal if the heuristic used is admissible (never overestimates the cost) and consistent (the estimated cost from node n to the goal is no greater than the cost from n to any successor node n', plus the cost from n' to the goal).

- Here's a brief summary of the completeness and optimality of the algorithms:

Algorithm	Completeness	Optimality
Depth-First Search (DFS)	Not complete in infinite state spaces or with loops.	Not optimal.

Algorithm	Completeness	Optimality
Breadth-First Search (BFS)	Complete.	Optimal if all edge costs are equal.
Iterative Deepening DFS (IDS)	Complete in finite state spaces.	Optimal if all edge costs are equal.
Uniform Cost Search (UCS)	Complete, as long as step costs are greater than a constant.	Optimal if step costs are non-decreasing.
Greedy Best-First Search (GR1, GR2)	Not guaranteed to be complete.	Not guaranteed to be optimal.
A* Search (AS1, AS2)	Complete and optimal if the heuristic is admissible and consistent.	Complete and optimal if the heuristic is admissible and consistent.