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.

Search Strategies

11. **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.
12. **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.
13. **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.
14. **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.
15. *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 Metrics

The performance of the algorithms is evaluated based on the following metrics:

- **Completeness:** Whether the algorithm is guaranteed to find a solution if one exists.
- **Optimality:** Whether the algorithm is guaranteed to find the best solution.
- **Number of Expanded Nodes:** The number of nodes that the algorithm expands during the search process.
- **CPU Utilization (%):** The percentage of CPU resources used by the algorithm.
- **Used Memory (MB):** The amount of RAM used by the algorithm (in megabytes).

Performance Comparison

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

Initial State 0

Algorithm	Num of Expanded Nodes	CPU Utilization (%)	Used Memory (MB)
DFS	4	8.14	5
BFS	25	8.68	5
IDS	35	7.66	5
UC	94376	15.32	104
GR1	5	4.33	5
GR2	502	8.68	6
AS1	94413	16.8	104
AS2	94376	17.39	103

Initial State 4

Algorithm	Num of Expanded Nodes	CPU Utilization (%)	Used Memory (MB)
DFS	459	6.85	6
BFS	673003	6.19	338
IDS	8246906	13.06	529
UC	1305994	8.96	620
GR1	1355	10.85	7
GR2	16359	8.14	8
AS1	1305994	10.88	627
AS2	1305994	5.21	645

Certainly, here's a summarized table of observations:

Initial State 0 Observations:

Algorithm	Number of Expanded Nodes	CPU Utilization	Used Memory	Remarks
DFS	Moderate	Low	Low	Lightweight but may not guarantee optimality
BFS	High	Moderate	High	Thorough but resource-intensive
IDS	Moderate	Moderate	Moderate	Combines advantages of DFS and BFS
UC	High	High	High	Explores based on path cost, substantial memory

Algorithm	Number of Expanded Nodes	CPU Utilization	Used Memory	Remarks
GR1 (Greedy)	Low	Low	Low	Efficient but not guaranteed optimal
GR2 (Greedy)	Moderate	Moderate	Moderate	More exploration than GR1
AS1 (A* Search)	High	High	High	Considers both path cost and heuristic, demanding
AS2 (A* Search)	High	Moderate	High	Another heuristic, similar characteristics

Initial State 4 Observations:

Algorithm	Number of Expanded Nodes	CPU Utilization	Used Memory	Remarks
DFS	Moderate	Low	Moderate	More nodes explored, slightly higher memory usage
BFS	Very High	Moderate	Very High	Extremely high number of nodes, highly memory-intensive
IDS	Very High	Moderate	High	Extensive node exploration, better memory efficiency compared to BFS
UC	High	Moderate	High	Explores based on cost, high memory usage
GR1 (Greedy)	Low	Moderate	Low	Consistently efficient with a low number of nodes explored
GR2 (Greedy)	Moderate	Low	Moderate	More exploration than GR1 but remains efficient
AS1 (A* Search)	High	Moderate	High	Similar characteristics to the first set, demanding
AS2 (A* Search)	High	Low	High	Lower CPU utilization but similar memory usage

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.

Algorithm Descriptions

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

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):** UCS is complete and optimal. It explores options in every direction according to path cost.
5. **Greedy Best-First Search (GR1 and GR2):** The completeness and optimality of these algorithms depend on the heuristic used. If the heuristic is admissible and consistent, the algorithm is both complete and optimal.
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).