Live Long and Prosper

November 14, 2023

1 Live Long and Prosper

In a town, essential resources such as food, materials, and energy are required to sustain the citizens and construct new buildings, contributing to the town's prosperity level. Building establishments enhances prosperity, but demands resources. The objective is to design a search agent tasked with devising a plan to elevate the town's prosperity level to 100. Various actions, each impacting resources and prosperity differently, can be taken by the agent.

The agent operates within a budget of 100,000 and has no additional income sources. Resource storage is limited to 50 units per resource. Resources deplete with every agent action, and to replenish them, a delivery must be requested. Consuming resources incurs costs, affecting the available budget. The cost is the total money spent on resources and builds from doing different actions using the operators available to the agent.

Actions available to the agent include `RequestFood`, `RequestMaterials`, and `RequestEnergy`, each with parameters specifying the amount and delay for resource delivery. Waiting (`WAIT`) is an option for the agent to wait for pending deliveries without further actions. `BUILD1` and `BUILD2` are actions directly influencing prosperity level, but they consume resources and entail additional costs, with each build type having distinct parameters and different units for each resource consumed. Each build type increases the prosperity of the town by a distinct number.

The agent's goal is to find a sequence of actions that raises the town's prosperity level to 100, ideally minimizing the expenditure. The success of the agent is measured by achieving the prosperity goal while optimizing the use of resources and budget.

2 Classes

1. `public class Node`

   The node class has the following attributes, and the following constructor.

```
private State state;
private Node parent;
private String operator;
private int depth;
private int pathCost;
```

```
public Node(State state, Node parent, String operator, int depth, int pathCost) {
super();
this.state = state;
this.parent = parent;
this.operator = operator;
this.depth = depth;
this.pathCost = pathCost;
}
```

where each node has a state, a parent, an operator, which is the action that led to this state, a depth and a path cost. For our project, the path cost is the money spent.

2. `public class GenericSearch`

   This is generic search problem class. It has a general search method that chooses a search strategy according to the its parameters. Our used strategies are:

   (a) Breadth-First search

   (b) Depth-First search

   (c) Uniform-Cost search

   (d) Iterative-Deepening search

   (e) Greedy search with 2 different heuristics

   (f) A* search with 2 different heuristics

   The class contains the following methods:

   - `public static SearchResult generalSearch(SearchProblem p, String qFun)`

     This method takes the search problem and a queueing function, gets the initial state of the problem and creates a node. This node is the root of the search tree. According to each queueing function there is a method that handles the enqueueing and the dequeueing of the nodes. A `SearchResult` is an object that has a `String result` and a `String visualize`. The result is in this form:

     `plan;monetaryCost;nodesExpanded`

     Where the plan is the sequence of actions taking to reach the goal, the monetary cost is the total money spent in each step, and the nodes expanded are the number of nodes that got expanded to find the solution.

   - `public static SearchResult breadthFirstSearch(SearchProblem p, Node n)`

     This method takes the root and the search problem and enqueues the root. Then proceeds to check for each node if it passes the goal test. If it passes, then it returns the search result, else, it proceeds to expand each node and enqueues its children in a FIFO fashion. The method

     `public static void bfs(Queue<Node> queue, Queue<Node> expandedNodes)` handles the enqueueing of the expanded nodes.

- `public static SearchResult depthFirstSearch(SearchProblem p, Node n)`

  This method takes the root and the search problem and pushes the root into a stack. Then proceeds to check for each node if it passes the goal test. If it passes, then it returns the search result, else, it proceeds to expand each node and enqueues its children in a LIFO fashion. The method `public static void dfs(Stack<Node> stack, Queue<Node> expandedNodes)` handles the enqueueing of the expanded nodes.

- `public static SearchResult uniformCostSearch(SearchProblem p, Node n)`

  This method takes the root and the search problem and pushes the root in a priority queue. Then proceeds to check for each node if it passes the goal test. If it passes, then it returns the search result, else, it proceeds to expand each node and enqueues its children according to the path cost, which is the money spent for our search problem.

- `public static IterativeDeepeningResult`

  `depthLimitedSearch(SearchProblem p, Node n, int depth, int totalNodesExpanded)`

  This is a Depth-First method with a cutoff which is the depth. We use it for the iterative deepening strategy to search for the goal node with every depth until the goal is found or no solution is found. The `IterativeDeepeningResult` returns the result, the number of nodes expanded with this depth only, and a String visualize to visualize the steps taken if a goal is found.

- `public static SearchResult IterativeDeepeningSearch(SearchProblem p, Node n)`

  This method uses the previous method to search for the goal starting from depth 0 until the depth of the deepest leaf node in the tree. It halts when it finds the goal or when the total nodes expanded of the previous iteration with depth-1 is the same as the total nodes expanded in the current iteration.

- `public static SearchResult greedySearch1(SearchProblem p, Node n)`
- `public static SearchResult greedySearch2(SearchProblem p, Node n)`
- `public static SearchResult AStarSearch1(SearchProblem p, Node n)`
- `public static SearchResult AStarSearch2(SearchProblem p, Node n)`

  For the previous 4 methods. They enqueue the root in a priority queue and proceed to expand according to their own heuristics which are discussed further in this report.

3. `public class LLAPSearch`

   This class extends the `GenericSearch` class.

   - `public static SearchProblem parse(String inputString)`

     This method takes the string of the initial state of the search problem and parses it into a search problem.

   - `public static String solve(String initialState, String strategy, boolean visualize)`

     This method takes the initial state and the search strategy and a boolean visualize. It calls the `parse` method on the string of the initial state and creates a new search problem with what the method returned. then it calles the `generalSearch` method with the search problem and the strategy as the parameters. It returns a search result. `Solve` returns the String result of the search problem, and if visualize is true, then it will print the visualization string of the search result which displays each node chosen from the root until the goal node.

4. `public class SearchProblem`

   This class is specifically formulated for our search problem. It has the following attributes.

```
private int prosperity;
private int food;
private int materials;
private int energy;
private int budget = 100000;

private int unitPriceFood;
private int unitPriceMaterials;
private int unitPriceEnergy;

private int amountRequestFood;
private int delayRequestFood;

private int amountRequestMaterials;
private int delayRequestMaterials;

private int amountRequestEnergy;
private int delayRequestEnergy;

private int priceBUILD1;
private int foodUseBUILD1;
private int materialsUseBUILD1;
private int energyUseBUILD1;
private int prosperityBUILD1;

private int priceBUILD2;
private int foodUseBUILD2;
private int materialsUseBUILD2;
private int energyUseBUILD2;
private int prosperityBUILD2;

private int delayCountFood = 0;
private int delayCountMaterials = 0;
private int delayCountEnergy = 0;

private int moneySpent = 0;

private State initialState;

private HashSet<String> states = new HashSet<>();
```

   As we mentioned above, we start with an initial budget of 100000 and no requests incoming, represented by the budget attribute and the three delays for the the three types of requests. We added a HashSet for the states that are expanded/created to avoid creating or exploring repeated

states. A repeated state is characterized by having the same prosperity, food , materials, energy, money spent, food delay, materials delay, and energy delay of another state. The following are the main functions of this class.

- `public Queue<Node> expand(Node node)`

  This method returns the children of an expanded node and ensures that the children didn't occur previously in the tree. it also ensures that you can't request any resource while you are awaiting the arrival of a resource.

- `public Node requestFood(Node node)`

  This method handles the effect of one of the operators of our search problem which is `RequestFood`. It sets the delay counter of this type to the delay specified in the problem. The calling of other operators decreases the delay counter. It also returns the resultant node of this operation.

- `public Node requestMaterials(Node node)`

  This one is similar to the previous method but for the `RequestMaterials` operator

- `public Node requestEnergy(Node node)`

  This one is similar to the two previous methods but for the `RequestEnergy` operator.

- `public Node Wait(Node node)`

  This method is for the `WAIT` operator. This operator can only be chosen when there is an active delivery and it decreases the counter of any delay by one. Once they delay reaches zero, the effect of the incoming delivery appears on the returned node

- `public Node build1(Node node)`

  The `BUILD1` operator consumes the resources assigned for this type build, the price of these resources and the price of the build and increases the prosperity according to this type of build. Also if there is any active delay, the operator decreases it by one.Once they delay reaches zero, the effect of the incoming delivery appears on the returned node.

- `public Node build2(Node node)`

  The `BUILD2` operator consumes the resources assigned for this type build, the price of these resources and the price of the build and increases the prosperity according to this type of build. Also if there is any active delay, the operator decreases it by one. Once they delay reaches zero, the effect of the incoming delivery appears on the returned node.

3 Heuristic Functions

For our A* and Greedy search algorithms, we implemented two different heuristics `H1` and `H2`. Both of these heuristics estimate the needed cost to reach the goal of a prosperity of 100.

3.1 H1: Calculating the Build Cost

This heuristic function estimates the cost of building needed to reach a prosperity of 100. The steps involved are as follows:

1. **Determine Build Quantities:** For each type of build (`BUILD1` and `BUILD2`), calculate the number of builds needed based on the required prosperity value.

2. **Identify Minimum Build Quantity:** Compare the number of builds required for `BUILD1` and `BUILD2`. Identify the minimum number of builds among the two options.

3. **Calculate Build Costs:** Determine the cost of each build type, factoring in the cost of the building itself and the cost of the required resources.
4. **Identify Minimum Build Cost:** Compare the calculated costs for `BUILD1` and `BUILD2`. Identify the minimum cost among the two options.
5. **Determine Estimated Cost:** The estimated cost is calculated by multiplying the minimum number of builds from step 2 by the minimum build cost identified in step 4. This represents a relaxed possible cost for reaching the goal.

3.1.1

As taken in class, for a heuristic to be admissible, it has to adhere to two conditions:

1. **Never overestimate the cost:** The estimated cost is less than or equal to the actual cost.
2. **The centering property:** The estimated cost at any goal node is equal to 0.

`H1` adheres to the first condition in the sense that it ensures that the estimated cost is an upper bound on the actual cost. By considering both the minimum number of builds and the minimum cost for building, the heuristic ensures that neither the number of builds nor the cost is overestimated, leading to a reasonable estimation of the total cost.

It adheres to the second condition by estimating the cost to be 0 at any goal node, because if the node's prosperity exceeds 100, the required number of builds would be 0.

3.2 H2: Calculating the Build and Requesting Resources Cost

This heuristic function estimates the cost of building needed to reach a prosperity of 100. In addition to the building cost, it incorporates the cost of requesting the needed resources. The steps involved are as follows:

1. **Determine Build Quantities:** For each type of build (`BUILD1` and `BUILD2`), calculate the number of builds needed based on the required prosperity value.
2. **Identify Minimum Build Quantity:** Compare the number of builds required for `BUILD1` and `BUILD2`. Identify the minimum number of builds among the two options.
3. **Determine Number Of Resources Requests Needed:** Identify the quantity needed for each type of resource, and determine how many requests would be needed for each resource.
4. **Identify Minimum Number of Requests:** Identify the minimum number of requests for each resource based on `BUILD1` and `BUILD2`.
5. **Determine Requesting Resources Cost:** This is calculated by multiplying the minimum number of requests by the cost of the request itself.
6. **Calculate Build Costs:** Determine the cost of each build type, factoring in the cost of the building itself and the cost of the required resources.
7. **Identify Minimum Build Cost:** Compare the calculated costs for `BUILD1` and `BUILD2`. Identify the minimum cost among the two options.
8. **Determine Estimated Cost:** The estimated cost is calculated by multiplying the minimum number of builds from step 2 by the minimum build cost identified in 7, then adding the minimum cost of requesting resources identified in 5. This represents a relaxed possible cost for reaching the goal.

3.2.1

Like `H1`, `H2` adheres to the first condition in the sense that it ensures that the estimated cost is an upper bound on the actual cost. By considering both the minimum number of builds, the minimum cost for building, and the minimum cost of requesting resources, the heuristic ensures that neither the number of builds nor the total cost is overestimated, leading to a reasonable estimation of the total cost.

It adheres to the second condition by estimating the cost to be 0 at any goal node, because if the node's prosperity exceeds 100, the required number of builds would be 0, consequently, no resources would need to be requested, so the number of requests needed would also be 0.

3.3 Greedy and A* Search Admissibility

The greedy algorithms employ the above two admissible heuristics to guide their search. The A* algorithms incorporate these heuristics along with the path cost. This combination ensures that the centering property is maintained while increasing the desirability of expanding nodes that lead closer to the goal.

4 Performance

	BFS	DFS	UCS	IDS	GR1	GR2	AS1	AS2
NE	678,325	3,849,870	4,841,327	341,419	1,420	350	3,133,764	3,127,967
CPU	76%	97%	90%	62%	48%	48%	88%	81%
RAM	400 MB	800 MB	1400MB	400MB	<100 MB	<100 MB	900MB	800MB
Completeness	Complete	Complete	Complete	Complete	Complete	Complete	Complete	Complete
Optimality			Optimal				Optimal	Optimal

4.1 Completeness

The proposed algorithms are guaranteed to terminate and return a solution, if one exists, due to the finite nature of the generated trees and the inherent constraints on expansion and handling of repeated states. This establishes the completeness of the algorithms.

4.2 Optimality

1. Uninformed search algorithms, such as BFS, DFS, and IDS, do not consider the cost of the search path during exploration. This can lead to the expansion of irrelevant or unnecessary nodes, potentially resulting in suboptimal paths with higher costs. In contrast, UCS (Uniform Cost Search) prioritizes expanding nodes with the lowest accumulated cost, ensuring that it always finds the optimal path with the minimum cost.

2. Informed search algorithms, like Greedy Search, employ heuristics to estimate the cost of reaching the goal from a given node. This allows them to make more informed decisions about which nodes to expand, potentially leading to faster search times. However, since heuristics are not always accurate, Greedy Search may not always find the optimal path.

   A* Search combines the benefits of both uninformed and informed search by incorporating a heuristic into UCS. This allows it to make informed decisions while still guaranteeing optimality. As a result, A* Search typically expands fewer nodes than any other optimal search algorithm.

4.3 Nodes Expanded

Refer to the table above.

- **Uninformed Search:**

  Uninformed search algorithms, such as BFS, DFS, and IDS, expand nodes without considering their cost or proximity to the goal. This can lead to the exploration of a large number of unnecessary nodes, resulting in inefficient search strategies.

  Based on an average of five randomly selected tests with the same initial states, BFS expands an average of 678,325 nodes, while DFS expands an average of 3,849,870 nodes. This difference arises from BFS's tendency to explore shallower depths first, while DFS tends to explore deeper depths first, potentially leading to long, inefficient paths.

  IDS, with an average of 341,419 expanded nodes, performs better than BFS due to its iterative deepening approach. However, it still expands more nodes than BFS because of its DFS-based exploration mechanism.

  UCS, with an average of 4,841,327 expanded nodes, exhibits the highest node expansion rate among the uninformed search algorithms. This is because UCS prioritizes expanding nodes with lower costs, which may not directly lead to the desired prosperity increase. Consequently, UCS explores a large number of safe nodes before reaching the optimal solution.

- **Informed Search:**

  Among the informed search algorithms, greedy algorithms exhibit the lowest node expansion rates. GR1 expands an average of 1,420 nodes, while GR2 expands an average of 350 nodes. This efficiency stems from their focus on estimated costs rather than actual costs, guiding the search towards the goal without getting bogged down in unnecessary exploration.

  AS1 and AS2, with average node expansions of 3,133,764 and 3,127,967, respectively, strike a balance between efficiency and accuracy. They combine the informed decision-making of greedy algorithms with the cost-based approach of UCS, resulting in fewer expanded nodes compared to UCS while maintaining optimality.

4.4 CPU Utilization and RAM Usage

1. **Uninformed Search:**

   As evident from the table, CPU utilization and RAM usage exhibit a direct correlation with the number of expanded nodes. UCS and DFS, with their high node expansion rates, consume the most CPU power, exceeding 90%. However, UCS's RAM usage surpasses that of DFS due to its space complexity of $O(b^{c^*/\epsilon})$, where b is the branching factor, c^* is the cost of the optimal solution and ϵ is the minimum operator cost, compared to DFS's space complexity of $O(bd)$, where b represents the branching factor and d represents the maximum depth. BFS and IDS follow with lower CPU utilization and RAM usage, reflecting their reduced node expansion compared to DFS and UCS.

2. **Informed Search:**

   The greedy algorithms demonstrate a significant reduction in CPU utilization and RAM usage. This stems from the ability of informed search, in general, to expand exponentially fewer nodes at the expense of path cost. The A* algorithms, while not dramatically lower in CPU utilization and RAM usage compared to UCS, still exhibit an improvement due to the influence of the

applied heuristics. This improvement highlights the effectiveness of informed search in reducing computational overhead while maintaining acceptable path costs.

5 Conclusion

In this report, we have implemented various search algorithms for the Live Long and Prosper search problem. We have evaluated their performance in terms of node expansion, CPU utilization, and RAM usage and came to the following conclusion:

While informed search algorithms may require more upfront processing to compute heuristics, their overall performance gains outweigh these initial costs. In scenarios where computational resources are limited, informed search algorithms offer a viable solution for finding optimal or suboptimal paths in complex search spaces.

References

Prof. Dr. Haythem O. Ismail Artificial Intelligence Lecture Slides