Artificial Intelligence
Project 1 Report

Team Members:

1. Maryam Sherif Hamdy Amin (49-5892) (T17)
2. Sara Sherif Kamal Hassan (49-6069) (T17)
3. Sonia Medhat Mohamed Elshamy (49-3973) (T17)
4. Mayar Ibrahim Ibrahim Abdelaziz (49-5951) (T17)

- **Project Description:**
  A town requires critical resources (food, materials, and energy) to support its residents and develop new structures. The town's prosperity level reflects the well-being of its residents. Buildings are essential for increasing the town's wealth, and these constructions necessitate specialized resources. The goal of this project is to create a search agent tasked with designing a plan to increase the town's prosperity to 100. The consequences of various activities on prosperity and resource levels differ. We are given a $100,000 budget with no other sources of income. The town has a resource storage limit of 50 units per resource, and each action the agent does depletes these resources. A delivery must be ordered to replenish the level of a resource. Using resources incurs costs, which indicate the cost of these resources. Our objective is to develop a search agent capable of investigating and, if possible, devising a plan to raise the town's prosperity to level 100.

- **A discussion of your implementation of the search-tree node ADT.**
  The 'State' Java class represents a state in the context of a search problem. Each instance of this class represents the current state of a system as defined by various attributes such as prosperity, food, materials, energy, money spent, and so on. The class is intended for use in search algorithms, where actions can be used to move from one state to another. The state contains information such as the search tree depth, the action that resulted in this state, and a reference to its parent state. The constructor sets the state's values for prosperity, food, materials, energy, money spent, total money owned, food, materials, and energy delays, the depth of the search tree, the action associated with this state, and a reference to its parent state. To facilitate proper comparison and hashing of state objects, the class overrides the 'equals' and 'hashCode' methods. The class includes getter methods for accessing various state attributes as well as setter methods for modifying the action and parent attributes. The class also defines a method for generating successor states based on a set of predefined actions, ensuring that valid state transitions are taken into account.The 'toString' method returns a human-readable string representation of the state that displays its various attributes. Overall, the 'State' class, with its representation of system states and their relationships, serves as a fundamental building block for modeling and solving search problems.

- **A discussion of your implementation of the search problem ADT.**
  GenericSearch code defines an abstract class named GenericSearch serving as a template for various search algorithms. The class includes static methods for common algorithms such as Breadth-First Search (BFS), Depth-First Search (DFS) with a depth limit, Iterative Deepening Depth-First Search (IDDFS), Greedy Best-First Search, and A* Search. Each method takes a State parameter, implying their applicability to tree structures, and currently returns null as a placeholder for the actual implementation. The class is designed to be extended by concrete subclasses, allowing for the specific implementation details of each search algorithm and the structure of the State class to be defined based on the requirements of the problem domain.

- **A discussion of your implementation of the LLAP problem.**
  Our LLAP problem is structured using object-oriented programming principles with classes like Initials and Actions. This helps organize the code and encapsulate related functionalities.

- **Initials Class:**

Initialization: The Initials class efficiently initializes the state of the town based on the provided input string. The use of a constructor that parses the input string ensures a clean and organized way to set initial parameters.

Attributes: The class effectively represents the state of the town with attributes like prosperity, food, materials, energy, and various parameters for requests and building constructions.

- **State Class**

The State class represents the state of the town in the LLAP problem. It encapsulates essential information, including prosperity level, resource quantities (food, materials, energy), money spent, delays, depth in the search tree, the action performed, and a reference to the parent state.

- **Actions Class**

The Actions class provides methods to perform various actions in the town, such as requesting food, materials, or energy, waiting, and building. It calculates the effects of these actions on the state and returns the resulting state. The performAction method serves as a dispatcher for different actions, making the code modular and easy to extend

- **Search Algorithms**

Our implementation incorporates several search algorithms to find an optimal plan for achieving a prosperity level of 100. These algorithms include Breadth-First Search (BFS), Depth-First Search (DFS), Uniform Cost Search (UCS), Iterative Deepening Search (IDS), Greedy Search with two heuristics (Greedy1 and Greedy2), and A* Search with two heuristics (A1 and A2).

1. **BFS:** The Breadth-First Search explores states level by level, utilizing a priority queue based on the depth of each state. This ensures that shallower states are explored first, making it complete and optimal for finding the deepest path to the goal state.
2. **DFS:** The Depth-First Search explores deeper states first using a stack. While not guaranteed to be optimal, DFS can be more memory-efficient than BFS.
3. **UCS:** The Uniform Cost Search prioritizes states based on the total money spent. It guarantees optimality by always selecting the path with the minimum cumulative cost.
4. **IDS:** The Iterative Deepening Search is a depth-limited strategy that systematically increases the depth limit until the goal state is found. It combines the advantages of DFS and BFS, providing an efficient and complete solution.
5. **Greedy Search:** The Greedy Search algorithm uses two heuristics (Greedy1 and Greedy2) to prioritize states. Greedy1 estimates the remaining distance to the goal based on prosperity, while Greedy2 considers the prosperity-to-cost ratio.
6. **A* Search:** The A* Search algorithm utilizes two admissible heuristics (A1 and A2) to guide the search efficiently. A1 estimates the remaining prosperity needed, and A2 estimates resource shortages. Both heuristics ensure admissibility and optimality.

Our implementation demonstrates a comprehensive approach to solving the LLAP problem. The State class, along with the Actions class, forms a solid foundation for representing and manipulating states in the LLAP problem. The use of different search algorithms and heuristics allows for a thorough exploration of the state space, providing flexibility in finding optimal solutions based on different criteria.

- **Main Functions:**

    - This '**RequestFood**' function depicts an action that the search agent can take in the context of a town simulation. The goal of this action is to replicate the process of obtaining food resources for the town, which will have an effect on many elements of the town's status. The function takes the town's current state ('s') as input and returns a new town state once the required food action is done.
      Here's a rundown of what the function does:

      1. Resource Reduction: It reduces the town's existing amounts of food, resources, and energy by one unit each. This decrease reflects the use of these resources as a result of the meal request.
      2. Increased Cost: It computes the entire cost sustained by the town as a result of resource depletion. The unit costs of food, materials, and energy influence the cost. The whole amount is then added to the town's current budget.
      3. Budget Revision: It adjusts the town's budget by deducting the entire cost from the total amount owned. The entire amount of money possessed is effectively the town's budget.
      4. Check for Validity: It looks for circumstances such as resource availability delays, limited resources, or budget limits. If any of these requirements is satisfied, the action is deemed invalid, and the function returns 'null,' indicating that it cannot be executed in the present state.
      5. State Formation: If the action is legitimate, it generates a new state that represents the town after the food request. This new state incorporates revised values for prosperity, resource levels, budget, delays, and depth. The action is set to 'REQUESTFOOD,' and the parent state is saved to monitor the plan.

      - The `**RequestMaterials**` function models an action within a town simulation, where the search agent initiates a request for materials to support the town's needs. The function takes the current state (`s`) of the town as input and returns a new state representing the town after the requested materials action is performed.

      Here is a breakdown of what the function is doing:
      1. Resource Decrease: It reduces the current levels of food, materials, and energy in the town by one unit each, simulating the consumption of these resources as a result of the materials request.
      2. Cost Increase: It calculates the total cost incurred by the town due to the resource decrease. The cost is determined by the unit prices of food, materials, and energy. The total cost is then added to the current money spent by the town.
      3. Budget Update: It updates the budget of the town by subtracting the total cost from the total money owned, representing the available budget.
      4. Validity Check: It checks for various conditions, such as delays in resource availability, insufficient resources, or budget constraints. If any of these conditions are met, the action is considered invalid, and the function returns `null`, indicating that this action cannot be performed in the current state.

5. State Creation: If the action is valid, it creates a new state representing the town after the materials request. This new state includes updated values for prosperity, resource levels, budget, delays, and depth. The action performed is set to `REQUESTMATERIALS`, and the parent state is also stored for tracking the plan.

- The **'RequestEnergy'** function represents a town simulation action in which the search agent initiates a request for energy resources to support the needs of the town. It accepts the current state ('s') of the town as input and returns a new state representing the town after the requested energy action is performed.
  Here's a rundown of the function:
  1. Resource Reduction: It reduces the town's current levels of food, materials, and energy by one unit each, simulating their consumption as a result of the energy request.
  2. Increased Cost: It computes the total cost incurred by the town as a result of resource depletion. The unit prices of food, materials, and energy determine the cost.
  3. Budget Revision: It updates the town's budget by subtracting the total cost from the total amount owned, resulting in the available budget.
  4. Check for Validity: It looks for conditions such as resource availability delays, insufficient resources, or budget constraints. If any of these conditions is met, the action is deemed invalid, and the function returns 'null,' indicating that it cannot be performed in the current state.
  5. State Formation: If the action is valid, it generates a new state that represents the town following the energy request. This new state incorporates revised values for prosperity, resource levels, the budget, delays, and depth. The action is set to 'REQUESTENERGY,' and the parent state is saved to track the plan.

- The **'WAIT'** function represents a town simulation action in which the search agent decides to wait for resource availability. It accepts the town's current state ('s') as input and returns a new state representing the town after the wait action is completed. Here's a rundown of the function:
  1. Resource Reduction: It subtracts one unit from the town's current levels of food, materials, and energy, simulating the passage of time while waiting for resources.
  2. Increased Cost: It computes the total cost incurred by the town as a result of resource depletion. The unit prices of food, materials, and energy determine the cost. The total cost is then added to the amount of money currently spent by the town.
  3. Budget Revision: It updates the town's budget by subtracting the total cost from the total amount owned, resulting in the available budget.
  4. Delay Handling (Food, Materials, and Energy): It determines whether there is a lag in the availability of any resource. If there is a delay, it decrements the delay counter. If the delay reaches zero, the current resource level is checked to see if it is greater than 50. If yes, the resource level is limited to 50; otherwise, the resource level is increased by the amount requested.
  5. Check for Validity: It looks for conditions like insufficient resources or budget constraints. If any of these conditions are met, the action is deemed invalid, and the function returns 'null,' indicating that it cannot be performed.

6. State Formation: After the wait action, if the action is valid, it creates a new state representing the town. This new state incorporates revised values for prosperity, resource levels, budget, delays, and depth. The action is set to 'WAIT,' and the parent state is also saved to track the plan

- The **'BUILD1'** function represents a town simulation action in which the search agent decides to build a building of type 1. It accepts the town's current state ('s') as input and returns a new state representing the town after the 'BUILD1' action is completed. Here's a rundown of the function:
  1. Resource Depletion: It reduces the town's current levels of food, materials, and energy based on the resource requirements for building type 1 (specified by 'init.getFoodUseBUILD1()', 'init.getMaterialsUseBUILD1()', and 'init.getEnergyUseBUILD1()').
  2. Increased Cost: It computes the total cost incurred by the town as a result of resource depletion and the cost of building type 1. The cost includes the unit prices of food, materials, and energy multiplied by their respective usage in building type 1, as well as the building type 1 fixed price. The total cost is then added to the town's current budget.
  3. Budget Revision: It updates the town's budget by subtracting the total cost from the total amount owned, resulting in the available budget.
  4. Delay Handling (Food, Materials, and Energy): It determines whether there is a lag in the availability of any resource. If there is a delay, it decrements the delay counter. If the delay reaches zero, the current resource level is checked to see if it is greater than 50. If yes, the resource level is limited to 50; otherwise, the resource level is increased by the amount requested.
  5. Check for Validity: It looks for conditions such as a lack of resources, budget constraints, or exceeding the maximum budget. If any of these conditions is met, the action is deemed invalid, and the function returns 'null,' indicating that it cannot be performed in the current state.
  6. State Formation: If the action is valid, it after the 'BUILD1' action creates a new state representing the town. This new state incorporates revised values for prosperity, resource levels, budget, delays, and depth. The action is set to 'BUILD1', and the parent state is also saved to track the plan's progress.

- The '**BUILD2**' function represents a town simulation action in which the search agent decides to build a building of type 2. It accepts the town's current state ('s') as input and returns a new state representing the town after the 'BUILD2' action is completed. Here's a rundown of the function:
  1. Resource Depletion: It reduces the town's current levels of food, materials, and energy based on the resource requirements for building type 2 (specified by 'init.getFoodUseBUILD2()', 'init.getMaterialsUseBUILD2()', and 'init.getEnergyUseBUILD2()').
  2. Increased Cost: It computes the total cost incurred by the town as a result of resource depletion and the cost of building type 2. The cost includes the unit prices of food, materials, and energy multiplied by their respective usage in building type 2, as well as the building type 2 fixed price. The total cost is then added to the town's current budget.

3. Budget Revision: It updates the town's budget by subtracting the total cost from the total amount owned, resulting in the available budget.
4. Delay Handling (Food, Materials, and Energy): It determines whether there is a lag in the availability of any resource. If there is a delay, it decrements the delay counter. If the delay reaches zero, the current resource level is checked to see if it is greater than 50. If yes, the resource level is limited to 50; otherwise, the resource level is increased by the amount requested.
5. Check for Validity: It looks for conditions such as a lack of resources, budget constraints, or exceeding the maximum budget. If any of these conditions is met, the action is deemed invalid, and the function returns 'null,' indicating that it cannot be performed in the current state.
6. State Formation: If the action is valid, it after the 'BUILD2' action creates a new state representing the town. This new state incorporates revised values for prosperity, resource levels, budget, delays, and depth. The action is set to 'BUILD2', and the parent state is also saved to track the plan's progress.

- **A discussion of how we implemented the various search algorithms.**
  - **BFS** :

    Our code uses the Breadth-First Search (BFS) algorithm to traverse states in a town simulation. The 'bfsSearch' method in the 'BFS' class takes an initial state as input and seeks a plan that leads to a prosperity level of 100 in the town. To prioritize states based on their depth in the search tree, the algorithm employs a priority queue ('openSet') and a custom 'BFSComparator'. Lower-depth states (closer to the root) are explored first.
    The algorithm iteratively expands states from the top of the priority queue, first exploring successors at the same depth level and then progressing to deeper levels. The goal is to find a state where the level of prosperity is equal to or greater than 100. The algorithm keeps track of the sites that have been visited.
    If a goal state is discovered, the algorithm reconstructs the path that led to it by tracing back through the parent pointers. The plan is made up of a series of actions, and the resulting 'SearchResult' object contains information like the plan, monetary cost, number of nodes expanded, and a list of state descriptions.

    The **'BFSComparator'** class defines the depth-based comparison logic, ensuring that states with lower depths are dequeued and explored before states with higher depths. This method ensures that BFS explores states level by level, making it complete and optimal for finding the deepest path to the goal state.

  - **DFS:**

    For exploring states in a town simulation, the provided code implements the Depth-First Search (DFS) algorithm. The 'DFS' class uses a stack ('openSet') to keep the state exploration order, with deeper levels explored first. This is in contrast to Breadth-First Search (BFS), which explores states at the same level before moving on to deeper levels. The 'DFS' algorithm is intended to find a plan that will lead to the town reaching a prosperity level of 100.The 'dfsSearch' method iteratively pops states from the stack, exploring their successors. The goal is to find a state where the level of prosperity equals or exceeds 100. If a goal state is encountered, the algorithm reconstructs the plan by tracing back through parent pointers, forming

a sequence of actions. The resulting 'SearchResult' object contains information such as the plan, monetary cost, number of nodes expanded, and a list of state descriptions.The 'DFSComparator' class defines the comparison logic for states based on their depth. States with greater depths are considered first, ensuring that the DFS algorithm explores deeper branches of the search tree before returning to shallower levels. This depth-first exploration strategy may result in faster solution discovery, but it is not guaranteed.

- **UCS:**
  The primary objective of UCS is to find the lowest-cost path from the initial state to a goal state in a search space. The `UCS` class extends the `**UCSComparator**` class, which defines a comparator for prioritizing states based on their total money spent. This total cost is crucial for UCS, as it selects the path with the minimum cumulative cost.In the `ucsSearch` method, the algorithm initializes a priority queue, `openSet`, using the `UCSComparator` to prioritize states with lower total money spent. The search iteratively expands states until a goal state is reached, updating the priority queue accordingly. The algorithm keeps track of visited states to avoid revisiting them. When the goal state is found, the method constructs and returns a `SearchResult` containing the optimal plan, monetary cost, number of expanded nodes, and the sequence of states. The `**UCSComparator**` class provides a comparison method that prioritizes states based on their total money spent. Lower-cost states are favored, ensuring that the algorithm explores paths with progressively lower cumulative costs. The UCS algorithm guarantees optimality in finding the lowest-cost path due to its nature of exploring paths in increasing order of cost.

- **IDS:**
  The Iterative Deepening Search (IDS) algorithm implemented in the `IDS` class is a depth-limited search strategy that systematically increases the depth limit until the goal state is found. The algorithm utilizes a depth-first search approach but with a depth limit set for each iteration. The `idsSearch` method initializes the depth limit to zero and iteratively performs depth-limited searches until a solution is found. In each iteration, the `performDepthLimitedSearch` method conducts a depth-limited search using a stack to manage the state space. The algorithm explores states up to the specified depth limit, checking for the goal state. If the goal state is not reached within the current depth limit, the algorithm increments the depth limit and repeats the process. The search is optimized by maintaining a set of visited states to avoid redundant exploration. The algorithm returns a `SearchResult` containing the solution plan, monetary cost, and nodes expanded upon reaching the goal state. Overall, IDS combines the advantages of depth-first and breadth-first search strategies, providing an efficient and complete solution for the LLAP.

- **Greedy1:**
  The provided code implements the Greedy Search algorithm using the Greedy1 heuristic. The 'Greedy1' class prioritizes states based on their estimated distance to the goal using a priority queue ('openSet') and a custom comparator, **'Greedy1Comparator'**. The heuristic estimates the remaining distance to the goal state by combining two factors: prosperity level and total money spent. The

comparator arranges states in ascending order of estimated distance, favoring states closer to the goal. The 'greedy search' method greedily explores states, prioritizing successors that are expected to move the algorithm closer to the goal state. The algorithm will keep searching until it finds a state where the prosperity level equals or exceeds 100, indicating the discovery of a goal state. The method now constructs a plan by backtracking through parent pointers and returns a 'SearchResult' object containing relevant information such as the plan, monetary cost, nodes expanded, and state descriptions. Greedy Search with the Greedy1 heuristic is best suited for problems that have a simple and effective estimate of the remaining distance to the goal. The heuristic presented here combines prosperity and monetary cost considerations, and its effectiveness may be affected by the problem domain's specific characteristics.

- **Greedy2:**

  The provided code implements the Greedy Search algorithm using the Greedy2 heuristic. The 'Greedy2' class prioritises states based on a prosperity-to-cost ratio using a priority queue ('openSet') and a custom comparator, **'Greedy2Comparator'**. This heuristic seeks to direct the search to states that provide greater prosperity for a lower cost. Based on the calculated ratio, the comparator ranks states in descending order, favouring states with better prosperity-cost trade-offs. The 'greedySearch' method greedily explores states, prioritising successors based on the prosperity-to-cost ratio. The algorithm will keep searching until it finds a state with a prosperity level equal to or greater than 100, indicating the discovery of a goal state. Following the discovery of the goal state, the method creates a plan by backtracking through parent pointers and returns a 'SearchResult' object containing relevant information such as the plan, monetary cost, nodes expanded, and state descriptions.Greedy Search with the Greedy2 heuristic is appropriate for problems where an effective heuristic involves weighing the benefits and costs. This heuristic prioritises states with a higher prosperity-to-cost ratio, indicating a preference for solutions that achieve a higher level of prosperity with less monetary expenditure. This heuristic's effectiveness is determined by the problem domain and the properties of the underlying state space.

- **A* 1:**

  The code implements the A* search algorithm in the `AStar1` class, utilizing a specific heuristic defined in the `**AstarComparator1**` class. A* search is an informed search algorithm that efficiently finds the optimal path from the initial state to a goal state in a search space. The heuristic, defined in `**AstarComparator1**`, guides the search by considering the remaining prosperity needed to reach the goal state. The comparator assigns priorities to states based on their estimated total cost, which is the sum of the money spent so far and the remaining prosperity needed.In the `aStarSearch` method, the algorithm maintains a priority queue, `openSet`, using the `AstarComparator1` to prioritize states with lower estimated costs. The search iteratively expands states until a goal state is reached, updating the priority queue accordingly. The algorithm keeps track of visited states to avoid revisiting them. When the goal state is found, the method constructs and returns a `SearchResult` containing the optimal plan, monetary cost, number of expanded nodes, and the sequence of states.The heuristic implemented in `AstarComparator1` ensures that

the search focuses on states with a lower estimated total cost, considering both the money spent and the remaining prosperity needed to achieve the goal. This enables the A* algorithm to intelligently explore paths, favoring those that appear more promising based on the heuristic information. A* guarantees optimality when the heuristic is admissible, and the provided implementation adheres to this principle by using a heuristic that never overestimates the remaining cost to reach the goal.

- **A* 2:**
  The code implements the A* search algorithm in the `AStar2` class, incorporating a specific heuristic defined in the `**AstarComparator2**` class. A* search is an informed search algorithm widely used for finding the optimal path from an initial state to a goal state in a search space. The heuristic, defined in `**AstarComparator2**`, guides the search based on resource shortages, considering the deficits in food, materials, and energy. The comparator assigns priorities to states based on their estimated total cost, which includes the money spent so far and the estimated resource shortages.In the `aStarSearch` method, the algorithm employs a priority queue, `openSet`, initialized with the `**AstarComparator2**` to prioritize states with lower estimated costs. The search iteratively expands states until a goal state is reached, updating the priority queue accordingly. The algorithm keeps track of visited states to prevent revisiting them. Upon finding the goal state, the method constructs and returns a `SearchResult` containing the optimal plan, monetary cost, number of expanded nodes, and the sequence of states.The heuristic implemented in `AstarComparator2` ensures that the search focuses on states with lower estimated total costs, considering both the money spent and the resource shortages in food, materials, and energy. This enables the A* algorithm to intelligently explore paths, favoring those that appear more promising based on the heuristic information. A* guarantees optimality when the heuristic is admissible, and the provided implementation adheres to this principle by using a heuristic that never overestimates the remaining cost to reach the goal.

● A discussion of the Heuristic functions

1. **Greedy 1**
   The Greedy1 algorithm used in this example is a heuristic search algorithm that seeks a solution to a problem that maximizes prosperity while minimizing the remaining distance to the goal. It explores states using a priority queue, prioritizing those with a lower estimated distance to the goal based on a heuristic function. In this case, the heuristic function is defined in the Greedy1Comparator class and calculates the difference between a state's current prosperity and the maximum prosperity (100) allowed to estimate the remaining distance to the goal. The algorithm keeps expanding states until it reaches a goal state with a prosperity of 100 or higher. The resulting solution is a series of actions that lead to a state of maximum prosperity while keeping the distance to the goal as short as possible. It is important to note, however, that this algorithm does not guarantee an optimal solution because it makes decisions solely on the heuristic without considering the total cost to achieve the goal.

The heuristic function used in the Greedy1 algorithm is admissible. In this case, the heuristic function estimates the remaining distance to the goal by calculating the difference between the current prosperity of a state and the maximum prosperity (100) allowed. Since prosperity is a measure of progress towards the goal, and the heuristic estimates the remaining distance by subtracting the current prosperity from the maximum possible prosperity, it is guaranteed to be non-negative, and, more importantly, it does not overestimate the remaining distance.

2. **Greedy2**

The Greedy2 algorithm uses a heuristic based on the prosperity-to-cost ratio to guide the search. The heuristic aims to strike a balance between reaching a high level of prosperity and minimizing the cost spent to achieve that prosperity. Let's break down the components of this heuristic:
The heuristic used by Greedy2 is a ratio of prosperity to cost, calculated as follows for each state `s`:
Heuristic Value = Prosperity of s / Money Spent in s
1. Prosperity: The numerator of the ratio represents the current prosperity level of the state `s`. The algorithm prioritizes states with higher prosperity levels.
2. Money Spent: The denominator of the ratio represents the cost incurred to reach the state `s` in terms of money spent. The algorithm aims to minimize this cost.
3. Prosperity-to-Cost Ratio: The heuristic value is the ratio of prosperity to cost. States with a higher prosperity-to-cost ratio are considered more promising, as they offer a better balance between achieving prosperity and minimizing expenditure.

The goal of the Greedy2 algorithm is to find a path to a state where prosperity is maximized relative to the cost spent. By considering the prosperity-to-cost ratio, the algorithm prioritizes states that are more efficient in achieving high prosperity with lower expenditure.

The heuristic used by Greedy2 is not guaranteed to be admissible. An admissible heuristic must never overestimate the cost to reach the goal from a given state. In the case of Greedy2, the heuristic is based on the ratio of prosperity to cost.While this heuristic provides a measure of the efficiency of reaching a state in terms of prosperity relative to cost, it does not guarantee that the estimated cost to the goal is always lower than or equal to the actual cost.

3. **A* 1**

The heuristic function employed in the A* search algorithm, as implemented in the `AstarComparator1` class, estimates the remaining prosperity needed to reach the goal state from a given state. The heuristic aims to guide the search by prioritizing states that have a lower remaining prosperity, reflecting a more direct path to achieving the goal of prosperity of 100. The calculation involves determining the difference between the maximum prosperity (100) and the current prosperity of a state, ensuring a non-negative value. This remaining prosperity is then combined with the money spent so far, creating an estimated cost that serves as a heuristic evaluation. The heuristic is designed to be admissible, as it never overestimates the actual cost to reach the goal. States are compared based on these estimated costs, with lower values indicating a more promising

path, allowing the A* search algorithm to efficiently explore and discover optimal solutions in terms of both prosperity and cost.

The heuristic used in the A* search algorithm, as implemented in the `AstarComparator1` class, is admissible. The calculation of `remainingProsperity` ensures that the heuristic value is always non-negative, and adding it to the money spent so far provides a lower bound on the actual cost to reach the goal. Therefore, the heuristic is admissible, and A* search using this heuristic is guaranteed to find the optimal solution in terms of both prosperity and cost.

4. **A* 2**

The heuristic function employed in the A* search algorithm, as implemented in the `AstarComparator2` class, focuses on estimating the remaining resource shortages in a state. This heuristic aims to guide the search by prioritizing states with lower shortages in food, materials, and energy, thereby favoring paths that address resource deficiencies more efficiently. The calculation involves determining the maximum shortfall in each resource (50 units) and subtracting the current amount possessed by a state. This ensures non-negative values for resource shortages. The heuristic then combines these shortages with the money spent so far, creating an estimated cost that serves as a heuristic evaluation. Similar to the previous heuristic, the AstarComparator2 heuristic is designed to be admissible, as it never overestimates the actual cost to reach the goal. States are compared based on these estimated costs, with lower values indicating more promising paths, allowing the A* search algorithm to efficiently explore and discover optimal solutions considering both resource shortages and cost.

The heuristic used in the A* search algorithm, as implemented in the `AstarComparator2` class, is admissible. The key condition for admissibility is that the heuristic must not overestimate the true cost to reach the goal. The heuristic used here satisfies this condition because it accurately reflects the remaining resource shortages and combines them with the money spent, providing a conservative estimate of the remaining cost. As a result, the A* search algorithm using this heuristic is guaranteed to find an optimal solution.

- **A comparison of the performance of the different algorithms implemented in terms of completeness, optimality, RAM usage, CPU utilization, and the number of expanded nodes. You should comment on the differences in the RAM usage, CPU utilization, and the number of expanded nodes between the implemented search strategies.**

	Completeness	Optimality	RAM usage	CPU utilization	Expanded nodes
BFS	Complete	Optimal only if the path to any node is equal to one	1.4 MB	48 ms	388
DFS	Incomplete.	Not Optimal.	1.03 MB	22 ms	58
UCS	Complete only if we do not have negative cost or	Optimal only if the cost is a non-decreasing	2.71 MB	33 ms	1771

	Completeness	Optimality	RAM usage	CPU utilization	Expanded nodes
BFS	Complete	Optimal only if the path to any node is equal to one	1.4 MB	48 ms	388
	the cost of the solution is not very big.	function as we go deeper in the tree.			
IDS	Complete.	Optimal provided that the path cost is a non-decreasing function in the depth of the tree.	1.67 MB1	48 ms	348
GR1	Complete	Not Optimal	1.03 MB	12 ms	21
GR2	Complete	Not Optimal	1.43 MB	52 ms	461
Astar1	Complete	Optimal	2.74 MB	48 ms	1737
Astar2	Complete	Optimal	2.71 MB	48 ms	1771

Analysis of the comparison:

- **RAM Usage**: UCS consumes the most memory, followed by Astar1 and Astar2. GR1 and DFS use the least memory.
- **CPU Utilization:** GR1 is the fastest, followed closely by GR2 and DFS. UCS and A* algorithms take more time due to their optimality.
- **Expanded Nodes:** UCS expands the most nodes, while DFS and GR1 expand the least.