Live Long and Prosper

1. Problem Discussion:

The problem involves a search agent that needs to deal with building a town and making it prosper. For the citizens living in the town, they need some resources such as food, materials and energy to live and build their town and increase its prosperity. For this problem, it is required to design a search agent that is responsible for finding an optimal plan to build and make the town more prosperous. The agent's goal is to reach prosperity level 100 and while spending least money and resources possible, as the agent has a budget of 100,000. However, the agent is given some criteria, possible actions and some initial resources to try to work with. The agent's challenge is to find a plan that leads to prosperity level 100 using some search algorithms that will be discussed later. Moreover, the agent should try to explore a possible plan that leads to the goal or return no solution if ever existed.

2. Search-tree Node Implementation:

In accordance with Russell and Norvig's framework, each node within the search tree is conceptualized as a 5-tuple, comprising the following components:

1. State:
    - This encapsulates the state space to which the node pertains
2. Parent node:
    - Signifying the predecessor node responsible for generating the current node through the application of a specific action.
3. Operator:
    - Denoting the action undertaken to generate the current node.

4. Depth/level:
    - Reflecting the position of the node within the hierarchical structure of the search tree.

5. Path cost/ money_spent :
    - Indicating the cumulative cost expended from the root to reach the current node.

Furthermore, supplementary parameters have been incorporated to meticulously track each pathway within the search tree, including:

6. Food:
    - The total quantity of food resources available to the agent at this particular node.

7. Materials:
    - The aggregate materials accessible to the agent at this node.

8. Energy:
    - The entirety of energy resources in possession of the agent at this node.

9. Prosperity:
    - Signifying the level of prosperity achieved up to the current node.

10. Wait:
    - Representing the waiting time associated with this node, if applicable.

11. RequestType:
    - Categorizing the type of resource request, with numerical assignments (1 for Food, 2 for Material, and 3 for Energy). This aids the agent in resource delivery decisions.

12. Heuristic:
    - Encompassing the heuristic value guiding the exploration of the respective path.

These additional parameters have been introduced to empower the agent in meticulously monitoring and managing resources across various paths explored during the course of its search.

3. *Search Problem Implementation:*

The implementation of the search problem aligns with the concepts discussed in the lecture. The general function was used, taking both the problem and a queueing function as inputs, and yielding either a solution or a failure. This function exhibits generality, adaptable to diverse search algorithms. Notably, the dequeuing aspect remains consistent across various algorithms. However, The key point is the user-defined queueing function, a parameter determined by the user. This function dynamically adjusts its operations based on user input, facilitating the generation of a solution or indicating a failure. The problem is framed with the root of the tree as the initial state, from which agents systematically explore potential plans to attain an optimal goal, which is check each time if a goal state was reached.

4. *LLAP Problem Implementation:*

The LLAP problem implementation revolves around the fundamental concept of "Live Long and Prosper" (LLAP). The LLAP class extends the genericSearch class, encompassing the definition of all variables and the invocation of the general search problem. Notably, the LLAP class features a static method named 'solve,' serving as the entry point for the entire project and taking user-input parameters.

The solve method, with the signature solve(String initialState, String strategy, boolean visualize), operates as follows:

- Initial State:
  The 'initialState' parameter represents the starting point of the problem, serving as the root of the tree where the agent initiates exploration to formulate potential plans.
- Strategy:
  The 'strategy' parameter determines the choice of the search algorithm, influencing the exploration of feasible paths leading to the goal. Each search algorithm has its queuing function string which is given to generic search problem by calling the generic search function.
- Visualize (Debugging):
  The 'visualize' parameter, a boolean flag, governs whether the state information is printed throughout the search algorithm. Primarily utilized for debugging purposes, it provides insights into the internal workings of the algorithm.

5. Search Algorithms Implementation:

Several search algorithms were implemented. They all differ in the enqueuing way as the dequeuing from the front is fixed for all search algorithms in the general search function. An ArrayList “nodes” and “ChildNodes” were used in the process of enqueueing for each search algorithm.

1. **Breadth-First-Search (BF)**: The BFS enqueues the nodes at the end of the nodes queue to ensure that the agent explores level by level. The Queuing function “EnqueueAtEnd” is passed to the general search problem along with the initial state. All generated children of a specific node are created and inserted in the ChildNodes queue then the Childnodes dequeues all its components to nodes queue at the end (FIFO).

2. **Depth-First-Search (DF)**: The DFS enqueues the nodes at the front of the nodes queue to ensure that the agent explores a whole path of a node till finding a valid solution or backtracks to explore another path. The queueing function “EnqueueAtFront” is passed to the general search problem along with the initial state. All generated children of a specific node are created and inserted in the ChildNodes queue then the Childnodes dequeue all its components to nodes queue at the front (LIFO).

3. **Uniform-Cost-Search (UC)**: The UCS acts exactly the same as BFS(exploring level by level) however, nodes with lower pathCost are favored inside the nodes queue. The queueing function “EnqueueCost” is passed to the general function along with the initial state. All generated children of a specific node are created and inserted in the ChildNodes queue and then ChildNodes dequeues all its components to nodes queue at the end, however, the nodes queue is then sorted according to the path cost ascendingly .. nodes are removed from the childNodes queue and passed to a BinarySearchandInsert function where the location - where the node should be put in order to make the nodes queue sorted ascendingly according to the money_spent - is searched using binary search for efficiency and then added in O(1) runtime as the nodes queue is of ArrayList data type.

4. **Iterative-Deepening-Search (ID)**: The IDS is a modified version of the DFS that works exactly the same however, in each iteration, sets a limit to the tree to avoid incomplete search and infinite loop. The queueing function “EnqueueID-Front” is passed to the general function along with the initial state. All generated children of a specific node are created and inserted in the ChildNodes and the ChildNodes dequeue all its components,only if the level of this node is smaller than or equal to the current idLevel, to the nodes queue however, a variable “idLevel” is incremented after finishing all dequeuing to allow the algorithm to loop several times while having each time a different limit to the tree, therefore, the looping stops when the agent finds a goal state. Moreover, the algorithm checks for anytime this node’s level and the current idLevel if they are equal, therefore the root is reinserted to the nodes queue to allow the algorithm to expand again however, with an incremented idLevel. Moreover, the hashset of redundant states is reset in each iteration to avoid counting non-existing redundant states. More on the redundant states will be explained in the main functions section.

5. **Greedy 1 (GR1):**
   The greedy algorithm works on a heuristic function (h(n)) which is basically counting how much money will be needed for requestMaterial actions the agent needs to reach a goal state. The queueing function “EnqueueProsperityGR1” is passed to the general function along with the initial state. For each child node inserted in the ChildNodes queue, then nodes are removed from the childNodes queue and passed to a BinarySearchandInsert function where the location - where the node should be put in order to make the nodes queue sorted ascendingly according to the heuristic of the node - is searched using binary search for efficiency and then added in O(1) runtime as the nodes queue is of ArrayList data type.

6. **Greedy 2 (GR2):**
   The greedy algorithm works on a heuristic function (h(n)) which is basically counting how much money will be needed for the Builds actions the agent needs to reach a goal stateThe queueing function “EnqueueProsperityGR2” is passed to the general function along with the

initial state. For each child node inserted in the ChildNodes queue, then nodes are removed from the childNodes queue and passed to a BinarySearchandInsert function where the location - where the node should be put in order to make the nodes queue sorted ascendingly according to the heuristic of the node - is searched using binary search for efficiency and then added in O(1) runtime as the nodes queue is of ArrayList data type.

7. **A* 1 (AS1):**
   The A* algorithm works on both the path cost which is estimated money_spent and a heuristic function of Greedy 1 to estimate how many steps are left to reach the goal state (f(n) = g(n) + h(n)). Also using the same concept of searching and sorting the nodes function ascendingly as the Greedy methods.

8. **A* 2 (AS2):**
   The A* algorithm works on both the path cost which is estimated money_spent and a heuristic function of Greedy 2 to estimate how many steps are left to reach the goal state (f(n) = g(n) + h(n)). Also using the same concept of searching and sorting the nodes function ascendingly as the Greedy methods and A*1.

6. Main Functions:

The main function of the Search problem is the GenericSearch class which holds all the main functions of the algorithm inside The GenericSearch function. The GenericSearch class initializes essential variables, including those in the initial state (problem). The 'exCount' variable meticulously tracks the expansion count, a crucial metric printed in the final solution, if one exists.

Two ArrayLists, 'nodes' and 'ChildNodes,' are integral components, along with a HashSet of redundant states strategically employed to manage instances of redundant states. Upon parsing the user's input initial state, a root node is crafted to encapsulate the initial state values of resources and prosperity, with the parent set to null. The root node is then inserted into the 'nodes' queue, initiating the iteration process.

In each iteration on the 'nodes' queue, an initial check ensures the queue is not empty. If empty, the output string is generated as "NO SOLUTION." Alternatively, the front node is dequeued to execute the necessary operations, depending on the operator. The switch statement assesses whether the operator corresponds to root, requestFood, requestMaterial, requestEnergy, Build1, Build2, or wait. Subsequently, the algorithm explores each potential path.

1. **root:** the only possible action is creating the corresponding 5 children nodes having all possible actions that a root can lead to such as requestFood, requestMaterial, requestEnergy, Build1 and Build2. Afterwards the agent will explore each of the possible paths to check how to reach a goal state. New children nodes are inserted in the childNodes queue and then dequeued according to the queueing function previously passed to the generic search and the corresponding search algorithm is applied as described above in section 5.
2. **requestFood, requestMaterial, requestEenergy:** for requesting a resource, there are several scenarios that can occur depending on the state of this node.
    a. Wait time of a node reaching 0: after checking that requestType of node is still not = 0 , means that a previous request food,material or energy is ready to be delivered and changes the values of my resources however, keeps track that the value of any of my resources does not exceed 50. Also, a checker is made on the type of resource the agent received to increase its corresponding value and requestType of node is reset to 0 indicating that other requests of any resources are pending. Moreover, the action of requestingFood, requestMaterial or requestEnergy is applied too, which is the main focus and requestFood, requestMaterial or requestEnergy function is present in node class. Resources functions decrease the values of my resource by 1 each and turn on the int flag (requestType) into its corresponding value (1-> food, 2-> material , 3 -> energy) and allows the wait of this node to take the value needed to wait for the delivery.

b. No wait time: as for the case of the root, just apply the action of requesting food, requesting material or requesting energy and wait time starts.

- Children: the possible child nodes of a requestFood/ requestMaterial/ requestEenergy are Build1, Build2 and wait.
- Redundancy: after applying all the actions needed, a redundancy check is made to ensure that the agent does not explore a path that was already explored and did not lead to a goal state. This ensures a more optimized algorithm.

After finishing all needed actions that should be done in node of requestFood/requestMaterial/requestEnergy, a check is made on resources that they did not reach negative values and that money_spent did not reach the agent's budget (100k) in order to ensure that my resources never reach negative values. If resources reach negative values, this node should be discarded and its children are not added to nodes queue and this path does not lead to a goal state.

Finally, the prosperity is checked after applying the action if reached 100 or more, the loop should break and the agent should formulate a solution.

3. **Build1,Build2:** for building, several scenarios exist depending on the state of my node.

a. wait=0: the agent should be ready for delivery of resources if any of them were pending. As mentioned above, delivery of a resource requires some checks and several steps to be completed.
b. wait>0: the wait time of the node is decremented by 1 and no pending requests are delivered.

- The agent applies the build actions on the corresponding node and checks if any of the resources reaches 0 or money spent exceeds the agent's budget.
- Children: the builds have 2 scenarios for generating children:
  a. wait>0: means still a resource request is pending therefore, only generates the possible actions of the agent which are Build1, Build2,

wait. (no requesting a resource should never be an option to the agent to explore whenever the wait time of this node is >0).

b. Wait = 0: agent is ready to explore the 5 possible paths which are requestFood, requestMaterial, requestEnergy, Build1 and Build2.

- Redundancy: after applying all the actions needed, a redundancy check is made to ensure that the agent does not explore a path that was already explored and did not lead to a goal state. This ensures a more optimized algorithm.

  After finishing all needed actions that should be done in node of build1/build2, a check is made on resources that did not reach negative values and that money_spent did not reach the agent's budget (100k) in order to ensure that my resources never reach negative values. If resources reach negative values, this node should be discarded and its children are not added to nodes queue and this path does not lead to a goal state.

  Finally, the prosperity is checked after applying the action if reached 100 or more, the loop should break and the agent should formulate a solution

4. **Wait**: there are 2 scenarios:
   a. Wait >0 : wait function is called (which is inside the node class) to decrease the resources each by 1 and the wait of this node.
   b. Wait = 0: as mentioned above, any pending request should be delivered and handled properly (resource amounts do not reach 50 and so on).

- The agent applies the build actions on the corresponding node and checks if any of the resources reaches 0 or money spent exceeds the agent's budget.

- Children: the wait has 2 scenarios for generating children:
  a. wait>0: means still a resource request is pending therefore, only generates the possible actions of the agent which are Build1, Build2, wait. (no requesting a resource should never be an option to the agent to explore whenever the wait time of this node is >0).

b. Wait = 0: agent is ready to explore the 5 possible paths which are requestFood, requestMaterial, requestEnergy, Build1 and Build2.

- Redundancy: after applying all the actions needed, a redundancy check is made to ensure that the agent does not explore a path that was already explored and did not lead to a goal state. This ensures a more optimized algorithm.

- **isRedundant**(): every node in the nodes queue when popped and all necessary actions are applied is checked if it is redundant or not, if redundant, then is it added to a visited HashSet which hashes a stringified object. A state is redundant if both nodes have exactly the same amount of food, materials, energy, money spent, prosperity and requestType.

- **formulateSolution()**: formulates the solution of the path to goal. If a solution exists, a string is generated as path to goal;cost;expandedNodes; else, the no solution string is generated.

7. Heuristic Functions:

Both discussed heuristics and considered admissible as they never overestimate the cost to reach a goal state.

1. Greedy 1:

   In this heuristic. Firstly, the prosperity left to reach the goal from this current state is calculated and divided by the prosperity gained by each of the builds, hence the number of builds needed in each case will be known and used to calculate the amount of material needed to reach the goal from this current state. After that, the amount of material needed is checked against the current state's material to know whether a request material action is needed and if so, how many requests are needed at least to reach the goal state. Finally, the minimal money needed to reach the goal is calculated by multiplying the amount of request material actions needed by the price spent by each build function. The minimal cost of both build types is taken as a heuristic for this current node. If the amount of material needed to reach the goal is less than the amount of

materials the current node has then the heuristic is set to be zero since no request actions will be needed to reach the goal.

1. Greedy 2:

   In this heuristic. Firstly, the prosperity left to reach the goal from this current state is calculated and divided by the prosperity gained by each of the builds, hence the number of builds needed in each case will be known and used to calculate the total money needed for builds to reach the goal state by multiplying the number of builds needed with the price of each build, The minimal cost of both builds is taken as a heuristic for this current node

1. A* 1:

   In this heuristic. Firstly, the prosperity left to reach the goal from this current state is calculated and divided by the prosperity gained by each of the builds, hence the number of builds needed in each case will be known and used to calculate the amount of material needed to reach the goal from this current state. After that, the amount of material needed is checked against the current state's material to know whether a request material action is needed and if so, how many requests are needed at least to reach the goal state. Finally, the minimal money needed to reach the goal is calculated by multiplying the amount of request material actions needed by the price spent by each build function. The minimal cost of both build types is taken and added to the path cost of the current state and taken as a heuristic for it. If the amount of material needed to reach the goal is less than the amount of materials the current node has then the heuristic is set to be the path cost since no request actions will be needed to reach the goal.

1. A* 2:

   In this heuristic. Firstly, the prosperity left to reach the goal from this current state is calculated and divided by the prosperity gained by each of the builds, hence the number of builds needed in each case

will be known and used to calculate the total money needed for builds to reach the goal state by multiplying the number of builds needed with the price of each build, The minimal cost of both builds is added to the path cost of the current state and taken as a heuristic for it.

8. Performance:

	Completeness	Optimality	RAM usage	CPU utilization	Expanded nodes number
BF	Yes	Somehow optimal	1573.725 megabytes.	60608168.33ms	13375.550
DF	NO	NO	168.529	7075723.88 ms	1608.598
UC	Yes	Somehow optimal	2607.566	5068462.465 ms	21772.777
ID	Yes	No	13546.984	132458512.156 ms	1490.800
GR1	Yes	No	1235.345	156485.354ms	780.992
GR2	Yes	No	1025.958	245608.678 ms	2750.272
AS1	Yes	Yes	732.254	1265.165 ms	8600.544
AS2	Yes moot	Yes gdn	473.254	2135.215 ms	1890.218

9. Comments:

16 tests cases have failed due to runtime errors, maybe the cause of this limited time provided in the tests.

Too much work was done on this project and too much thinking!