AI Project I Report Team 34

By: Abdullah Abdelhafez, Marwan Ashraf, Laila Hammouda

The "Live Long and Prosper" project is a search problem where the goal is to create a plan to make a town reach a prosperity level of 100 by managing its resources and choosing which actions to perform whether it is establishing new buildings, requesting food, materials or energy or waiting for other events to occur such as receiving awaited requested materials then taking another action. The agent has a budget of 100,000 and limited storage for resources – namely 50 units per resource type. The project requires implementing a generic search procedure which takes a problem and a search strategy choice as inputs, and it implements the search strategy of choice to find a plan that will lead this town to a prosperity level of 100 if one exists. The search strategies covered by this project are breadth first search (BF), depth first search (DF) uniform cost search (UC), iterative depth search (ID), greedy search (GR) and A* search (AS).

Search Tree Node ADT

The Node class represents nodes in a search space for the search tree. It contains instance variables such as:

- `State state`: Represents the state associated with the node. It contains information about prosperity level; food, energy and materials available, and money spent.
- `Node parent`: Represents the parent node in the search space.
- `int cost`: Represents the cost associated with reaching this node.
- `String deliveryType`: Represents the type of delivery awaited that is associated with this node.
- `int delay`: Represents the delay associated with resources requested prior to reaching this node.
- `String action`: Represents the action taken to reach this node.

In addition, it contains getter methods for all instance variables of a node. It, also, contains additional methods that perform actions such as getting the cumulative cost of reaching a node; incrementing energy, food and materials – to be used when requesting any of them; getting the path from the root that leads to this node; checking whether this node passes the goal test or not; and overriding the `String toString` method for printing and visualizing.

The node class also includes a nested class inside it for the state. The state nested class contains the following attributes:

- `int prosperity`: Represents the current prosperity level in the state represented by the node.
- `int moneySpent`: Represents the total money spent till reaching this node.
- `int food`: Represents the number of currently available food units.
- `int energy`: Represents the number of currently available energy units.
- `int materials`: Represents the number of currently available materials units.

In addition it contains getter methods for all instance variables of a state, setter methods for food, energy and materials and overriding the `String toString` method for printing and visualizing.

Search Problem ADT

For implementing the generic search problem, we created an `abstract class genericSearch` which is to be later implemented in the `LLAPSearch` class.

LLAP Search Problem

The LLAP search problem is an implementation of the generic search problem which implements how exactly the agent is going to create the plan for solving the search problem. It mainly calls a method `solve` and gives it the search strategy wanted, and accordingly, `solve` calls the search method requested and gives it as input parameters the initial values given in the problem.

Implemented Functions

The first function we implemented as discussed above is `String solve (String initialState, String strategy, boolean visualize)` which takes as input the initial state of the problem, a search strategy and a boolean representing whether we want to visualize steps at nodes or not. It ,first, creates a root node with the initial state and calls the function of the search strategy – based on the one requested as input – and gives it the root as input and as a beginning for creating the search tree.

Next, we have the function `Queue<Node> expand (Node node)` which mainly creates a queue of the child nodes of all the possible actions that can be made on the input node after checking whether we have enough money/ food/ materials/ energy to perform these actions or not.

In addition, we have the `Queue<Node> expandUC(Node node)` and `Queue<Node> expandDF(Node node)` which are very similar to the `expand` function. However, the difference in `expandUC` is that if the agent has enough resources he will always build, and if not he will request resources or wait. As for the `expandDF`, the logic is the same yet inverted as it will first perform the build action rather than wait to avoid entering an infinite path of waits.

We also have the `int getCost(String action)` function which calculates the cost of performing actions such as requesting food according to the inputs given when initializing the problem.

Moreover, we implemented two versions of a function to create child nodes based on the action that will be done. The first one is `Node createChild(Node parent, String action, int prosperity, int food, int energy, int materials)`. This function is used whenever the action done is build or wait. It adds the action done to the plan; calculates the new prosperity level after performing this action – with a maximum of 100; deducts the amount of resources given when initializing the problem; adds the cost of reaching this node to the cumulative cost; in addition, it checks whether there was an awaited delivery of a request for food, energy or materials, decrements the delay for this material, and if this is the time of arrival of this material it checks what type of material is arriving, increments the value of it with a maximum of 50 and resets the awaited delivery type to null. Lastly, it constructs a new node with the attributes calculated above and increments the number of expanded nodes. The second function that creates a child node is `Node createChild(Node parent, String action, String deliveryType, int initialDelay)` which is used whenever the action leading to the child node is requesting food, energy or materials. Similar to the first one, it adds the action to the action plan, gets the cost of this action and updates the cumulative cost; then, it gets the current prosperity level and decrements the amount of

food, money and energy available. Lastly, it creates a new node with the above achieved variables and increments the number of expanded nodes.

Moreover, we have the `Node requestFood(Node parent), Node requestMaterials(Node parent)`and `Node requestEnergy(Node parent)` functions which decrement all the available resources and calls the `create child` function described above with the corresponding action given as an input parameter.

Furthermore, there is th`e Node WAIT(Node parent)` function which decrements the resources specified in addition to the delay variable, and it creates a child node as specified above.

Then, we have `Node build1(Node parent)` and `Node build2(Node parent)` which calculate the cost of building according to the given values when initializing the problem adds it to the total monetary cost and subtracts it from the value of money in addition to decrementing the delay for requested resources if there was any. Then, it creates the child node as specified above.

Additionally, we created the function `void decrementResources()` which was used in some functions mentioned above to decrement the current value of food, energy and materials, add their price to the cost and deduct it from the amount of money we have. In addition, we created the method `String arrayListToString(ArrayList<String> list, String delimiter)` which converts an arraylist to a string with items separated by thi delimiter given as input.

Lastly, the function `Node initializeVariables(String initialState)` which parses the string that is given as input to the problem and initializes all variables that are used in the LLAP search problem and then creates the root of the tree.

Search Algorithms

To begin with, there are some common procedures implemented in all search algorithms on every node we get out of the queue. These are checking whether we visited this node before, and if we did, we skip this node and move to the following one,

if not we add it to the hash set of visited nodes in which we hash nodes based on the path we took to get there. Then, we check if it passes the goal test, we return the current prosperity level and print it. If not, we check if the amount of money spent is greater than 100000 and return “NOSOLUTION”. Also, at the end of every search algorithm, in case the plan was successful, we convert the plan arraylist to a string and print it, and we get the path to the solution, the total money spent and the number of expanded nodes.

The first search algorithm we implemented is the *breadth first search (BF)*. It mainly has a queue of nodes that starts with the root, and then as long as the queue is not empty we dequeue the head of the queue; and apply the above procedures on it. If it passes all above, we apply the `expand` function on it. Then, we enqueue every node in the result achieved and repeat the process all over again if the queue is not empty.

Next we have the *depth first search (DF)*. In the DF algorithm, we used a stack to represent the queue of the search problem starting with only the root in it. Then, as long as the stack is not empty, we pop the top of the stack and apply the above mentioned procedures on it. If it passes them all, we apply the `expandDF` function on the popped node and then push all the resulting child nodes on top of the stack and repeat the procedure again.

For the *uniform cost search (UC)*, we create a priority queue that orders nodes in the queue based on the cumulative cost of getting to this node which begins with only the root in it. Then, as long as this queue is not empty, we dequeue the head of the queue, and after applying the same common steps for all search algorithms on this node, we apply the `expandUC` function on it and enqueue all the children we achieve for this node in the priority queue and repeat the process all over again.

Next we have the *iterative depth search (ID)*. The ID search is basically an implementation of the DF search. We start with the same way of initializing a stack to act as the queue of the search problem. Then, we create two extra variables: one for the iteration counter (which represents the maximum depth), namely `i`, and one for tracking the current `expansionDepth` in the tree. Then, as long as the expansion depth is less than the maximum depth, we perform the loop of the DF search as is with two

modifications. First, at the end of every inner loop we increment the `expansionDepth`; moreover, after reaching an empty queue (implying no solution) in the inner loop, we increment the maximum depth – `i` – and repeat the same process all over again.

The next algorithm is the *greedy search algorithm (GR1 & GR2)*. Both algorithms implement the same procedures yet with different heuristics. In both algorithms, we first create a priority queue which sorts nodes based on the results of applying the heuristic functions on them and add the root to it. Then, while this queue is not empty, we dequeue the head of the queue and apply the above steps upon. If the node passes all above steps, we apply the `expand` function on it and add all the resulting children to the priority queue; then, we repeat the process again.

The last algorithm is the *A* search algorithm (AS1 & AS2)*. It is implemented in the exact same way as GR1 and GR2, yet instead of applying the `expand` function, we apply the `expandUC` function to the nodes we dequeue from the priority queue.

Heuristic Functions

As for the greedy search procedures, the first heuristic function used is the cost of food required until the prosperity level reaches 100, and the second heuristic function is the cost of building until the prosperity level reaches 100. These two heuristics were considered suitable measures of closeness to the goal (prosperity reaching 100) as they possess the central property and are simple to calculate.

As for the A* search strategy, we used the same heuristics as the greedy search, but we added on them the cumulative cost to reach this node. Both of these heuristics are considered admissible, since they depend on the difference between the goal prosperity and the prosperity at the agent's current state, meaning neither formulae will give an overestimated value.

Test Results *(on Initial State #3)*

	Complete?	Optimal?	CPU utilization (ms)	RAM utilization (MB)	# expanded nodes
BF	yes	yes	70	2.1	3097
DF	no	no	19	0.485	41
UC	yes	no?	30	0.734	132
ID	yes	no?	22	0.242	41
AS1	yes	yes	44	0.187	44
AS2	yes	yes	25	0.334	88
GR1	yes	no?	40	0.389	132
GR2	yes	no?	26	0.389	132

Breadth-First: It is complete and optimal. It explores the shallowest nodes first and ensures finding the optimal solution, but it may have higher CPU and memory requirements. It expands a relatively large number of nodes, as it explores all the nodes at the current depth level before moving to the next level.

Depth-First: It is not complete nor optimal. DF can get stuck in infinite loops in certain cases and may not find the optimal solution. Expands a smaller number of nodes, focusing on a single path until it reaches the end.

Uniform-Cost: Expands a moderate amount of nodes, depending on the cost associated with each path. It aims to minimize the total cost.

Iterative-Deepening: Expands a relatively small number of nodes, similar to Depth-First, but with multiple iterations at increasing depths.

A*: Expands a relatively small number of nodes, prioritizing nodes based on the

summation of the cost and given heuristic.

Greedy: Expands a larger number of nodes, prioritizing nodes based solely on the heuristic without considering the cost to reach them.