

Artificial Intelligence

Project 1 Report

Zainab Samir Soltan

40-2595, T12. Team 29

22nd November, 2020



Contents

1. Implementation of Search Tree Node Abstract DT
2. Implementation of the search problem Abstract DT
3. Implementation of MissionImpossible problem
4. A description of the main functions you implemented
5. A discussion of how you implemented the various search algorithms
6. A discussion of the heuristic functions you employed and, in the case of A^* , an argument for their admissibility.
7. At least two running examples from your implementation.
8. A comparison of the performance of the implemented search strategies on your running examples 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.

Implementation of Search Tree Node Abstract DT

Class name: `SearchTreeNode.java`

A search tree node, is meant to carry the state + more information. My implementation of the search tree node is that it is an object that contains 7 attributes: the first 5 are the 5 tuples as defined by Russel and Norvig, namely: state, parent node, operator, depth and cost from the root. I chose the state to be a string in order to save space and I encoded the different operators as bytes data type, for e.g. an operator value of 1 corresponds to action up...and so on (More on this is discussed in how I modelled the search problem). In addition I added 2 attributes: heuristic1 and heuristic2, which correspond to the value of the heuristics from each heuristic function. The heuristics and cost are calculated using a separate method that exists in the missionImpossible class. I left the computation of the heuristics to be outside of the search tree node class because the search tree node is an abstract class and therefore it should not contain anything specific to the details of the mission. In addition, the search tree node class implements Comparable to be able to sort the SearchTreeNode objects. There are 5 comparators: cost, heuristic1, heuristic2, cost+heuristic1, and cost+heuristic2. Cost comparator is used in Uniform Cost search and the heuristics comparators are used in greedy and A* search.

Implementation of the search problem Abstract DT

Class name: `SearchProblem.java`

Search problem is implemented as an abstract class in order to be a generic class. It is an object that represents a generic search problem and contains 4 attributes:

- operators , represented as an array of bytes
- An Initial state represented as a string
- A hashset of visited states to prevent repeating states.
- A public integer that represents the number of expanded nodes.

In addition the abstract methods implemented are:

- StateSpace, which takes a string that represents the current state and an operator. This acts as a transition function and returns a string representing the next state.
- goalTest, takes a currentState string and returns true if the currentState is a goal state and false otherwise

- `pathCost`, takes a search tree node and computes the path cost from the root and returns it.
- `Heuristic`: takes an integer that represents which heuristic is to be calculated (1 or 2, corresponding to `heuristic1` and `heuristic2`) and a search tree node to calculate the heuristic for.

Finally, the non-abstract methods:

- `GeneralSearchProcedure`: the core of the `SearchProblem` class. It is a static method that takes a problem instance and a string that represents the queuing function. How this method operates is discussed in more detail in the section titled: A discussion of the various search algorithms implementation
- `DepthLimitedSearch`: this is used to perform iterative deepening. It performs normal depth first search but stops at a certain depth which is given as input to the method.

Implementation of MissionImpossible problem

Class name: `MissionImpossible.java`

`MissionImpossible`'s purpose is to create an object that can capture our problem. In order to do that, formulation of the state was first done. As Ethan is advancing through solving the problem, the things that must be known are: his location, and who he is carrying. There is no need to include things like grid dimensions, or the position of the submarine as these things are constant and are not affected by Ethan's actions; therefore these were kept as class attributes. In my implementation, the state is a string of the the format: `ex,ey;s1,s2...sn` where:

- `Ex` and `ey` are Ethan's location.
- `Si...sn` is a sequence that represents the carry status of each of the IMF members. The state of the IMF members can be either: `n` (not carried), `c` (carried) or `s` (at submarine).

For example, consider this grid string: `"5,5;1,2;4,0;0,3,2,1,3,0;20,30,90;3"`. The initial state representation of this would be : `"1,2;nnn"`. This means that all IMF members are not carried. In case we enter a state where we carry an IMF member, say the 2nd member, the state string would be: `"2,1;ncn"`. This means that Ethan is at position (2,1) and only the second member is carried and on the truck. The goal state representation would be: `"ex,ey;sss"` where `ex` and `ey` equal the `x` and `y` coordinates of the submarine. Using this representation and some string manipulation we can find out the number of IMF members that are currently carried and whether Ethan's truck is at full capacity or not. In addition, it is easy to determine the damage that the members are receiving by looking at the `SearchTreeNode` depth and its state. If a member `m` is not carried at depth `i`, this means `i` consecutive actions have been performed by Ethan. Knowing

that every action is performed in one time step and one time step results in a damage of 2 to every IMF member not carried, we can conclude that member m received a damage of $i*2$. The class attributes are:

- M and n : which are 2 bytes that represent the dimensions of the grid
- $submarineX$ and $submarineY$: 2 bytes that represent the place of the submarine
- $IMFLocations$: a string that represents the positions of the IMF members in the format of $x1, y1, x2, y2...x_n, y_n$.
- $IMFHealth$: a string that represents the positions of the IMF members in the format of $h1, h2...h_n$.
- $truckCapacity$ (AKA c)
- $Path$: an arraylist that will be used to store the path (sequence of $SearchTreeNode$ objects) to the goal, by backtracking, only after reaching the goal.

In addition to the abstract methods that the class implements because it extends the $SearchProblem$ superclass, the functions that this class implemented are:

- $genGrid$: generates a random grid
- $Solve$: breaks the grid down, creates an instance of the class $MissionImpossible$, uses the information in the grid to fill in the attributes of the $MissionImpossible$ instance as well as formulate the initial state. The instance of $MissionImpossible$ is passed to $GenericSearchProcedure$ where search is performed there. The goal node returned by search is then used to construct a plan and visualise (if boolean $visualize$ is true)
- $getPlan$: takes a goal node returned by search and backtracks through the nodes till it gets to the root. As it is going along it concatenates the operators to a string that is then returned to the method $solve$. The output is a reversed plan of what ethan should do.
- $decodePlan$: in order to save space, the operators or actions that Ethan can do are represented as numbers rather than strings. $decodePlan$ method takes as an input a string that is the list of numerical operators that will lead to the goal and converts the numbers into what they mean as actions. {1=UP, 2=DOWN, 3=RIGHT, 4=LEFT, 5=CARRY, 6=DROP}.
- $getPathToGoal$: gets a goal node, and backtracks to the root. While backtracking it adds the nodes met to the path arraylist. The end result is that the path arraylist contains $searchTree$ nodes that represent the path from root to goal. Path arraylist is needed for visualisation as well as returning the number of deaths and healths in plan returned by $solve$ method.
- $printGrid$: takes a grid represented as a string, and converts it into a 2 dimensional array and prints it to console. "EH" cell represents where Ethan Hunt is, "SU" is where the

submarine is, and the numbers represent the damage of the IMF members. The location of the numbers is where the members are.

- `visualiseGrid`: visualises the steps that Ethan should take to accomplish his goal. It loops over the steps in the path and uses the information in the node and state to generate a grid. It then calls `printGrid` on the generated grid in order to print it to the console.

A description of the main functions implemented

[StateSpace \(String currentState, byte operator\)](#)

This method is the transition function that determines what the next state will be based on the current state and operator given. It is declared as abstract in the `SearchProblem` class and overridden by `MissionImpossible`. In `MissionImpossible`, the method first manipulates the string to get Ethan's x and y coordinates in addition to the statuses of the IMF members. After that it checks which operator it got. There are 6 possibilities:

- 0-> UP: decrements the x position, only after checking it is initially greater than 0.
- 1-> DOWN: increments x position after checking it is initially smaller than m-1.
- 2-> RIGHT: increments y position after checking it is less than n-1
- 3-> LEFT: decrements y position after checking it is more than 0.
- 4-> CARRY: Checks if the number currently carried is less than the truck capacity, then checks if there is an IMF member in the same cell as Ethan and is not already carried, if a carry action is successful the status of the IMF member is updated from n to c.
- 5-> DROP: Checks if Ethan is at the submarine and drops all members, and updates the dropped members statuses from c to s.


The new state is returned after performing one of the actions. If for any reason the action is not feasible the same state is returned and no update is made, repeated state handling is done by the search procedure itself.

[goalTest \(String currentState\)](#)

Returns true if Ethan's position is the same as submarine and all members are dropped, i.e. the carry status consists of s characters only. Returns false otherwise.

[pathCost \(SearchTreeNode node\)](#)

Path cost's function is to assign costs to sets of actions in a way that penalises actions that do not serve our optimality criteria. For starters, the method accesses the state of the node and is able to know where Ethan is and which members are carried. Next, using the depth of the node the



health is calculated. Then a check is performed to see if the member is dead. If a member dies, a penalty of 5000 is given. This is to deter the agent from actions that contribute to death as much as possible. This death penalty is added once when the IMF member dies, because if we keep adding it the agent will choose to carry the dead members first to reduce the cost. This is why a boolean (incursDamage) is used. In addition, with every time step (action) a penalty is added for every member who is not carried. The penalty amount depends on the total number of IMF members. This if it were to be a fixed number a large number of IMF members will incur a high penalty with every passing time step and this might at some depth be equivalent to a death penalty. Therefore, the penalty is $2 * (11 - \text{number of IMF members})$. This would mean a smaller number would incur a high penalty and a larger number will incur a small penalty with every passing time step. This was done because during testing I have noticed that a fixed penalty does not work every time, as it yields different optimality levels with different grids so this variable penalty fix was made. In addition, to make the agent steer away from carrying dead members so early, a penalty of 3000 was added in case of carrying a dead member. This was done because without adding this penalty the agent would perceive carrying a close dead body equivalent to taking a step towards a living IMF person which is something that we do not want because we prioritise carrying alive IMF members so that the damage is minimal.

[Solve\(String grid, String strategy, boolean visualize\)](#)

Solve begins by parsing the grid and extracting the meaningful info from it. After that it creates an object instance of MissionImpossible, using the information extracted. This object is passed to the general search function GeneralSearchProcedure (how GeneralSearchProcedure works is presented in the next section). After the solution is returned, this function calls helper methods that backtrack and find the path, as well as the plan followed. After obtaining the path, a for loop iterates through it in order to figure out the health of the members using the depth of the nodes and the operators. Once the carry operator is found it is known that the damage stops here for that member. In case the damage is more than 100, it is truncated to a 100. After that, another loop iterates through the healths to determine the number of deaths. The final plan string is a concatenation of the plan, deaths, healths, and number of nodes. The number of nodes is a class variable updated with every expansion from the GeneralSearchProcedure.

A discussion of the various search algorithms implementation

Firstly, I implemented `GeneralSearchProcedure`, which is a static method that takes as an input a search problem and a string `QING_FUN` that represents a queuing function. The method accesses the initial state of the problem and turns it into a `SearchTreeNode`. This acts as a root for the hypothetical search tree. Then an `ArrayList` (nodes) of `SearchTreeNode`s that will act as our queue is initialised and the root is enqueued. In addition an integer (`depthCount`) is initialised; it will help us keep track of the depth limit in the case of iterative deepening search. Next iterative deepening search implementation is discussed, followed by BFS, DFS, Greedy and A*.

After the necessary initializations, the method checks if the type of search is iterative deepening search (`QING_FUN` equals "ID"). If so, a loop starts to execute. The loop condition is if the `depthCount` is less than `Integer.MAX_INT`, if it is true: the problem instance and the `depthCount` is passed to `DepthLimitedSearch` method. `DepthLimitedSearch` performs DFS only up to `depthCount` level. This is done by extracting the initial state from the problem, turning it into a `SearchTreeNode` and enqueueing it at the front of an `ArrayList`. After that the program enters a while true loop. If the `ArrayList` is empty, that means no solution is found and null is returned. If the `arrayList` contains something, we get the element at index 0 and check if it is a goal node. If it is a goal node we return it, else we expand it if and only if its depth is less than `depthCount`. Expansion happens by applying the operators to the state of the node. Applying operators on a state results in a next state (which might have been visited before). To get the next state we use the method `StateSpace` which is overridden in any `SearchProblem` class. After getting the next state, I check if it has been visited, by checking in a hashset of visited states. If the next state has not been visited before, I add it into the `visitedStates`, turn it into a node, and enqueue it at the front of the `arrayList`. The loop ends here and execution starts again from the top until the array is empty or a goal is found. If a goal is found, `DepthLimitedSearch` returns it to `GeneralSearchProcedure`, where `GeneralSearchProcedure` returns it as a goal node. If this is not the case, `DepthLimitedSearch` returns null to `GeneralSearchProcedure` which implies that no goal was found within the depth limits. In that case, we increment `depthCount` and pass it to `DepthLimitedSearch` again. This allows `DepthLimitedSearch` to search in deeper depths to find a solution. It is worth noting that incrementing `depthCount` by only 1 unit makes the program very very slow, therefore I increment `depthCount` by 100 every time in order for iterative deepening search to run faster.

If the `QING_FUN` is not equal to ID, we proceed to enter a while true loop. The loop gets the first element in the `ArrayList` (nodes) and checks if it is a goal state, if so returns it and if not it expands

it by trying the possible operators. To get the next state the StateSpace is used. It returns a nextState and in the case that the next state is not a visited state a new node nextNode is created that contains the next state and a cost is calculated for that node. Then we check for the QING_FUN, if it is equal to...

- BF: add nextNode at the end of ArrayList
- DF: add nextNode at the front of ArrayList
- UC: add nextNode at the front then sort the ArrayList using the cost of each node in it.
- ASi: calculate heuristic number i for nextNode, then add nextNode at the front then sort the ArrayList using cost+heuristic number i.
- GRi: calculate heuristic number i for nextNode, then add nextNode at the front then sort the ArrayList using heuristic number i.

The loop ends and starts over again until a solution is found, or the ArrayList is empty which means that there is no solution.

A discussion of the heuristic functions employed and, in the case of A^* , an argument for their admissibility.

The intuition behind the heuristics implemented is to make the agent prefer to minimise the distance between himself and the most damaged member who could be saved. In order to do this, I get the distance from Ethan to the IMF member with the maximum damage using Manhattan (if it is heuristic 1) or Euclidean (if it is heuristic 2) provided 2 conditions:

1. The IMF member is not dead (damage is less than 100), because then Ethan will be encouraged to save dead members before alive ones.
2. The IMF member will not die by the time Ethan gets there.

The heuristics will give lower values when Ethan is getting closer to members who are not dead. In addition, this observes the centering property as if all members are carried it will return 0. Finally, it is admissible because every step that Ethan takes, a cost of $2 \times (\text{number of IMF members} - 1)$ at least is enforced on him for every uncarried member. This means that the heuristic will never overestimate the path to the goal because it returns the distance between Ethan and the most damaged member (that Ethan will have to save at some point in his journey, and thus Ethan will have to go through that path). If the distance is n , the cost is at least n and the heuristic will return n . Therefore no overestimation occurs.

At least two running examples from your implementation.

Example 1:

Grid: "5,5;1,0;2,4;0,1,1,3,2,2;2,90,3;1"

	2			
Ethan			90	
		3		Submarine

BF:

right,up,carry,down,down,right,right,right,drop,up,left,carry,down,right,drop,left,left,carry,right,right,
drop;1;6,112,37;438

DF:

up,right,carry,down,down,right,right,right,drop,up,left,carry,down,right,drop,left,left,carry,right,right,
drop;1;6,112,37;435

ID:

up,right,carry,down,down,right,right,right,drop,up,left,carry,down,right,drop,left,left,carry,right,right,
drop;1;6,112,37;436

UC:

right,right,right,carry,down,right,drop,left,left,carry,right,right,drop,up,up,left,left,left,carry,down,do
wn,right,right,right,drop;0;38,96,21;203

AS1:

right,right,right,carry,down,right,drop,left,left,carry,right,right,drop,up,up,left,left,left,carry,down,do
wn,right,right,right,drop;0;38,96,21;203

AS2:

right,right,right,carry,right,down,drop,left,left,carry,right,right,drop,left,left,up,up,left,carry,down,down,right,right,right,drop;0;38,96,21;203

GR1:

right,up,carry,down,down,right,right,right,drop,left,left,carry,right,right,drop,up,left,carry,down,right,drop;1;6,124,25;167

GR2:

Up,right,carry,down,right,right,right,down,drop,up,left,left,down,carry,right,right,drop,up,left,carry,down,right,drop;1;6,128,29;241

Example 2:

Input grid : "10,10;3,0;6,5;1,3,2,1,3,1,4,2,5,5;2,96,20,54,99;1"

			2						
	96								
Ethan	20								
		54							
					99				
					Sub				

BF:

right,up,carry,down,down,down,down,right,right,right,right,drop,up,up,up,up,up,left,left,carry,down,down,down,down,down,right,right,drop,up,up,up,left,left,left,left,carry,down,down,down,down,right,right,

right, right, drop, up, up, left, left, left, carry, down, down, right, right, right, drop, up, carry, down, drop; 3; 40, 100, 90, 152, 213; 10767

DF:

Up, right, carry, down, down, down, down, right, right, right, right, drop, up, up, up, up, up, left, left, carry, down, down, down, down, right, right, drop, up, up, up, left, left, left, left, carry, down, down, down, right, right, right, right, drop, up, up, left, left, left, carry, down, down, right, right, right, drop, up, carry, down, drop; 3; 40, 100, 90, 152, 213; 10768

ID:

Up, right, carry, down, down, down, down, right, right, right, right, drop, up, up, up, up, up, left, left, carry, down, down, down, down, right, right, drop, up, up, up, left, left, left, left, carry, down, down, down, right, right, right, right, drop, up, up, left, left, left, carry, down, down, right, right, right, drop, up, carry, down, drop; 3; 40, 100, 90, 152, 213; 10769

UC:

Right, carry, down, down, down, right, right, right, right, drop, up, up, left, left, left, carry, down, down, right, right, right, drop, up, up, up, up, up, left, left, carry, down, down, down, down, down, right, right, drop, up, up, up, up, left, left, left, left, carry, down, down, down, down, right, right, right, right, drop, up, carry, down, drop; 2; 60, 188, 22, 84, 213; 3449

AS1:

right, carry, down, right, down, down, right, right, right, drop, up, up, left, left, left, carry, right, down, down, right, right, drop, up, up, up, up, up, left, left, carry, down, down, down, down, down, right, right, drop, up, up, up, up, left, left, left, left, carry, down, down, down, down, right, right, right, right, drop, up, carry, down, drop; 2; 60, 188, 22, 84, 213; 3407

AS2:

right, carry, right, down, down, right, down, right, right, drop, left, left, up, up, left, carry, right, right, down, right, down, drop, up, up, left, up, up, up, left, carry, down, down, down, down, down, right, right, drop, up, up, up, up, left, left, left, left, carry, down, down, down, down, right, right, right, right, drop, up, carry, down, drop; 2; 60, 188, 22, 84, 213; 3449

GR1:

right, carry, right, down, down, right, down, right, right, drop, left, left, up, up, up, up, up, carry, down, down, right, down, down, down, right, drop, up, up, up, up, left, left, left, left, carry, down, down, down, down, right, right

,right,right,drop,up,up,left,left,left,carry,down,down,right,right,right,drop,up,carry,down,drop;3;36,164,22,152,213;2523

GR2:

right,carry,down,right,down,down,right,right,right,drop,up,up,left,left,up,up,up,carry,right,down,down,down,down,right,drop,up,up,up,up,left,left,left,left,carry,down,down,down,down,right,right,right,drop,up,up,left,left,left,carry,down,down,right,right,right,drop,up,carry,down,drop;3;36,164,22,152,213;2207

A comparison of the performance of the implemented search strategies on your running examples in terms of:

completeness

BFS: complete and finds a solution if there exists one.

DFS: due to repeated states handling, there are no infinite branches therefore DFS finds a solution if there is one.

UC, A*, Greedy, ID: complete and finds a solution in our case.

Optimality

BFS: not an optimal solution as we can see 1 member died when they could have been saved, this is to be expected as BFS does not take cost into consideration.

DFS: Like BFS, it did not produce an optimal solution because it ignores all costs.

ID: For the same reason as DFS and BFS, did not produce an optimal solution.

UC: produced an optimal solution where no one dies and also minimal damage to members. As we can see Ethan chose to save the agent who was about to die (damage 90) then dropped him at the submarine (limited truck capacity) then chose to get the member closest to the submarine instead of getting the far one first in order to minimise damage.

AS: Both heuristics 1 and 2 produced optimal plans just like UC.

Greedy: Did not produce an optimal solution because the heuristics alone were not enough to guide the agent.

RAM usage, CPU utilization & Number of expanded nodes

To determine the time duration of execution, time utility was used. Also to find out the program's used memory in Bytes the runtime utility was used which allows determining the free memory and the total memory, subtracting those results in the program's used memory. Please note that the below values of CPU and RAM from the task manager are an approximation, as the program executes pretty quickly so it was hard to determine the exact utilisation. As we can see, generally, BFS consumes a lot of memory. Iterative deepening generally takes longer time to execute as compared with the rest, and this gets worse the smaller the increment level is. The number of expanded nodes is generally less in uniform cost search, and even less in A* (or equal) which is to be expected because A* has a sense of intuition that comes from the heuristic. The greater the grid is, the more CPU and RAM the program needs and the longer it takes due to the fact that the hypothetical search tree would grow more and there are much more options for search to look through before finding a solution.

Example 1:

	BF	DF	ID	UC	AS1	AS2	GR1	GR2
CPU	20%	18%	17%	16%	16%	17%	19%	20%
RAM (MB)	1315	1328	1224	1222	1258	1348	1352	1278
Memory (Bytes)	1081848	1070656	1070192	1070752	1070952	1070904	1071064	1071032
Time (ms)	128	124	119	117	118	110	118	112
expanded Nodes	438	435	436	203	203	203	167	241

Example 2:



	BF	DF	ID	UC	AS1	AS2	GR1	GR2
CPU	25%	17%	16%	24%	18%	22%	24%	24%
RAM (MB)	1345	1220	1247	1226	1350	1204	1346	1262
Memory (Bytes)	1047920	1047784	1047344	1048328	1047376	1048512	1048640	1048640
Time (ms)	437	450	485	314	350	352	304	309
expanded Nodes	10767	10768	10769	3449	3407	3449	2207	2523