GUC
German University in Cairo

الجامعة الألمانية بالقاهرة

Winter has come!

Team 04
Omar Ashraf Sallam 34-8761
Mohamed El Zarei 34-14613
Abdelrahman El Shabrawy 34-15459

October 18, 2018

Contents

1 Brief Description of The Problem

In this problem, we try to model a search agent called Jon Snow, a character from Game of Thrones. Jon Snow, the king in the north, tries to kill all the White Walkers. The map in which Jon Snow tries to navigate and kill the White Walkers can be represented as a grid of m rows & n columns where m,n ≥ 4. A grid cell can either be free or it contains one of the following: White Walker, Dragon Stone or an obstacle. In dragon stones cells, Jon Snow can pickup dragon glass pieces by which he can kill the White Walkers. Jon Snow has a maximum capacity of dragon glass pieces to hold at any moment. To kill a white walker, Jon Snow has to be in an adjacent cell where this cell is sharing a side with the White Walker cell. When Jon Snow uses a dragon glass, he can kill all the White Walkers in the adjacent cells, if any found. To navigate around the map, Jon Snow can move in any cell of the four direction as long this cell doesn't contain a White Walker or an obstacle. He can't also move outside the borders of the grid. Jon Snow is set to be initially at the bottom right corner. The objective of problem is to implement different search strategies to help Jon Snow kill all the White Walkers in a randomly generated grid while using the minimum number of dragon glass pieces.

2 Search Tree Node

Search Tree Nodes implementation is a 5-tuples of the following:

- **State**: Java object of a generic class called `State` representing one of the states in the state space. We implemented a subclass of the state class called `WestrosState`, in which we implement our state. The implementation of WestrosState is discussed in 2.1.
- **Parent Node**: A reference for the search tree parent `Node`.
- **Operator**: Java object of a generic class called `Operator` representing the operator applied to generate this node. Each Operator should have a name and a cost of applying this operator.
- **Depth**: Integer value representing the depth of this node in the search tree. This value is set to be d+1 where d is the depth of the parent.
- **Path Cost**: Integer value representing the path cost from the root to this node. This value is set to 0 initially.

2.1 Westros State

WestrosState is a subclass of the *State* class. Westros state is a 4-tuple of the following:

- **Position**: Pair of integers representing the x&y position of Jon Snow in the grid.
- **Dragon Glass**: Integer value representing the number of dragon glass pieces Jon Snow has.

- **Max Dragon Glass**: Integer value indicating the maximum number of dragon glass pieces that Jon Snow can hold at any moment.
- **White Walkers Positions**: Java's `TreeSet` of Pairs. Each of these pairs represent a White Walker position that is not killed yet.

3 Search Problem

Search Problem implementation is a 3-tuples of the following:

- **Operators**: List of Operators objects available to the agent.
- **Initial State**: Java object of class `State` representing the initial state of the search agent.
- **Goal Test**: A method that checks whether a state is a goal state or not.
- **State Space**: A method called `getNextState` that takes a `State` object and an `Operator` object and returns a `State` object after applying the operator to the input state.
- **Path Cost function**: A method that takes an `Operator` object and `Node` object and calculates the path cost value of this particular node.

4 Save Westros Problem

The `SaveWestros` problem class is a subclass of the `SearchProblem` class discussed in 3. When initializing an object instance of this class, a grid is generated representing the map that Jon Snow will navigate through. The following subsections include a detailed description of the properties and methods in the SaveWestros search problem class. First, subsection 4.1 discusses initial state values. Then, `SearchQueue` class is discussed in subsection 4.2.Subsection 4.3 lists the operators available to the search agent Jon Snow along with their cost. Finally, we discuss node expansion implementation in subsection 4.4.

4.1 Initial State

The initial state in the Save Westros problem is a Java object instance from the `WestrosState` class. We initialize the position to be Jon Snow's initial position (bottom right corner) represented by row no. m-1 & column no. n-1. Then, dragon glass is set to have a value of zero and the maximum dragon glass capacity is set to have the random value generated in the generating map function in 5.1. Finally, the white walkers positions is list of pairs representing positions for each of the white walkers generated initially by the generating map function.

4.2 Search Queue

`Search Queue` is an abstract class of our implementation represents the queue in the search algorithm. This abstract class enables us to make different search queue class for each search algorithm. We implemented a subclass of this abstract class for each of the search algorithms mentioned in 6. Every subclass should have a data structure to store the search space and should also have a enqueue and a remove front method to expand the search space. Finally it has a method to check if the data structure is empty or not. The `Search Queue` class and its subclasses have a Java `TreeMap` of previous visited states in order to limit the search space and not to repeat previous subtrees of expansion in the search space. We discuss this part in section 6.1.

4.3 Operators

First, we have an abstract class called `Operator` representing the abstract concept of operator in the search problem and search tree node. Then, we implemented a subclass of `Operator` class called `WestrosOperator` to be the super class for each operator in this specific problem. Then we implemented a super class for movement operators called `MovementOperator`. This class has 4 sub classes one for each movement operator in our problem: up, down, left, right. Finally, we have an operator for using a dragon glass and kill all the adjacent white walkers called `KillOperator` and one for picking up dragon glass pieces from dragon stones called `PickupOperator`.

For movement operators, we check if the cell the search agent tries to move into is valid or not. For a cell to be valid, it first should be inside the borders of the map. We check this by the limits of m,n randomly generated while generating the map. Also, the cell can't contain a white walker or an obstacle to be valid. We check this by looking at the symbol inside this cell according to table 3.

For the `KillOperator` to be valid, we check if Jon Snow has dragon glass pieces and there is at least one adjacent white walker to kill. Finally for the `PickupOperator` to be valid, Jon Snow has to be in a dragon stone cell. Also, the number of dragon glass pieces, he is holding at the moment, should be less than the maximum number of dragon glass pieces that he can hold at any moment.
In case of invalidity of any of the operators above when we apply them on some node, the result of applying the operator returns null. We handle this in the enqueue function for the `SearchQueue` class. Finally, each operator cost is mentioned in 1. We selected the kill operator cost to be 3*(m+n) where m&n are the grid dimensions. This cost was selected to handle the case where the obstacles, which have a count of 15% of the grid size, form a diagonal line in the grid, we attempt to make the agent prefer to re-position itself by walking around this diagonal at a cost of at least 2*(m+n) to perform a more optimal kill. This re-positioning can make it kill more white walkers with just one dragon glass. Therefore the kill operator had to be more expensive than this re-positioning cost, and 3*(m+n) was the choice.

Operator	Cost
Movement Operator	1
Pickup Operator	1
Kill Operator	3*(m+n)

Table 1: Cost of Applying Operators on Search Tree Nodes

4.4 Node Expansion

After each operator checks if it can be applied to this node according to what we discussed in subsection 4.3, It starts applying itself to the state of this node. Table 2 presents how each operator will change node's state accordingly. i&j represents the row and column position of the search agent respectively. d is the number of dragon glass pieces that the search agent is holding at the moment and m is the max number of dragon glass pieces that Jon Snow can hold at any moment. Finally, ww is the list of alive white walkers positions and $\overline{ww}$ is the list of white walkers positions after killing all the adjacent white walkers using one dragon glass. After generating a new node the path cost function is called on this node and the operator to generate the cost. The cost is set to be the cost of the parent added to the cost of the operator.

Previous State	Operator	Next State
(i,j,d,m,ww)	Move Up	(i-1,j,d,m,ww)
(i,j,d,m,ww)	Move Down	(i+1,j,d,m,ww)
(i,j,d,m,ww)	Move Right	(i,j+1,d,m,ww)
(i,j,d,m,ww)	Move Left	(i,j-1,d,m,ww)
(i,j,d,m,ww)	Kill	(i,j,d-1,m,$\overline{ww}$)
(i,j,d,m,ww)	Pickup	(i,j,m,m,ww)

Table 2: State Space

5 Main Functions

5.1 Generating Map

First, dimensions of the grid m,n are randomly generated to have a minimum of 4. A 2D array of characters is generated of dimensions m,n. Then, we set the number of white walkers, obstacles and dragon stones to be 25%, 15% & 10% out of the number of grid cells, respectively. Then, K random positions are generated, where K is the addition of white walkers, dragon stones and obstacles. Then, we start filling the grid for each class by symbols in table 3. Finally, we add Jon Snow symbol in the bottom right corner. After generating the grid, the max number of the dragon glass pieces that Jon Snow can hold at any moment is randomly generated too.

Class	Symbol
Jon Snow	'J'
White Walker	'W'
Obstacle	'#'
Dragon Stone	'D'
Empty Cell	'.'

Table 3: Each Class Symbol Representation in the Grid

5.2 Search Function

We combined the search function requested in the project description with the general search algorithm in one method implemented in `SearchProblem` class. The method takes 3 parameters, a reference to the `SearchQueue` class, `BaseHeuristicFunction` object instance and finally the visualize Boolean variable. Before calling this method in the main to start running the examples, an object instance from `SaveWestros` problem class is initialized and accordingly a map is generated in the same way mentioned in 5.1.

The search method works as the following. First, it instantiates a Java `SearchQueue` Object of the input class. In case, the heuristic function object is not null, we initialize this object with the heuristic function object. We have 3 different heuristic functions that we are using all mentioned in section 7. Finally the visualize Boolean variable is used to be passed to the `printSolution` method in case a solution is found. We will discuss this method later in section 5.2.1. Then, general search algorithm was implemented like the one in the lecture with the exception that when a node is removed from the queue to be explored we check whether it's null or not. if the node is a null we don't process it and remove the next one.

5.2.1 Solution Printing

First, we implemented `Solution` class where in this class we handle printing solutions and visualizing search agent explored solution step by step showing the grid representing the search agent decisions. When the general search method in 5.2 founds a solution it calls a `printSolution` method that's implemented in `SaveWesteros` class. In this methods, a `WesterosSolution` object is initialized having the following parameters: visualize, a Boolean variable representing whether a visual representation for the gird after each step of the search agent decisions and total nodes, an integer value that represents the total number of nodes expanded while searching for a solution.

printSolution method keeps traversing nodes from the goal state by following the reference that each node has to its parent node till it reaches the root of the search tree. Each node in this traversal is added into a Java `ArrayList` of nodes that each `Solution` object has. After this traversal, a `printSolution` method is called which is implemented in the `WesterosSolution` class. In this method, we display information about the sequence of steps that the search agent found as a solution to this problem, like: the total number of expanded nodes, the depth of the path of the solution, the number of dragon glass pieces used to kill all the white walkers and finally the number of expanded nodes for this solution. Also, if visualize option was enables, a grid is printed for each step of the agent discovered solution.

6 Search Algorithms

In this section, we present the implementation details of each of the search algorithms. Each search algorithm is a subclass of the `SearchQueue` class we discussed before in 4.2. for each algorithm, we present what data structure is used to store the explored search space nodes and other methods implemented in each class. But before we present each of the algorithms, we discuss our memoization technique not to repeat search subtrees that were explored before in subsection 6.1. Then, we discuss the uninformed search techniques in 6.3 and the informed search strategies in 6.4.

6.1 Memoization

Memoization is an optimization technique used in search problems that have an overlapping subproblems. This means that a search agent will keep expanding those subproblems as if it never encountered this subproblem before. To avoid that the search agent can keep track of those visited subproblems (states) so before expanding any subproblem in the search tree, a search agent can check whether this was encountered before or not. First a state should be defined, this state should have enough variables that we are sure if any two states have the same values for each of the variables this means that's a repeated state. We choose the variables to be the same in the state defined in subsection 2.1. Then to save those states, `SearchQueue` class has Java Tree Map that keeps track of those states. For that, a `compareTo` is defined in the `WesterosState` class as any object used as a key in Java Tree Map needs to have a `compareTo` method. The check of repeated states is handled in the subclasses representing each strategy below.

6.2 Evaluators & Comparators

In search strategies like uniform cost, greedy and A*, nodes are expanded based on priority set by some evaluation function. To reduce redundancy and code duplication, we first implemented `NodeComparator` class. `NodeComparator` class inherits Java Comparator interface. That enables this comparator to be passed to a priority queue to evaluate priority of nodes to be expanded first. `NodeComparator` class has a `BaseEvaluator` object where this object evaluates that priority of each node when called in the compare method. `BaseEvaluator` is an abstract class for any class that evaluates which node to expand first in uniform cost, greedy and A* strategies. Any subclass of this class has to implement an apply method where in this method is the value of the evaluation function that this evaluator subclass wants to prioritize the nodes upon.

6.3 Uninformed Search

In uninformed search, the search agent generates the search tree without using any domain specific knowledge. It doesn't take into account the location of the goal. In the next subsections, we present our implementation details to some of the uninformed search algorithms like breadth-first search, depth-first search, iterative deepening search and finally uniform cost search.

6.3.1 Breadth-first Search

Breadth-first search explores the search tree level by level. To achieve that, the algorithm needs to rely on a standard queue (first in first out) as a data structure to store the explored search tree nodes. In the `BreadthFirst` class, the data structure initialized is Java Queue. We enqueue nodes normally in the queue, check if it's empty too through `isEmpty` method when we remove a node to start exploring we check whether it's visited or not. If it's visited we don't explore this node we return a null that will be handled in the general search algorithm. Otherwise, the state is saved in the `TreeMap` and we return this node.

6.3.2 Depth-first Search

Depth-first search explores the search tree by starting with the root vertex (initial state) and keeps exploring a single path until it reaches a leaf before backtracking to another path. To approach the idea of backtracking, the data structure we used in the `DepthFirst` class is stacks. Stacks have the concept of LIFO(last-in-first-out) which is the same as backtracking. Again like the breadth-first search, depth-first search enqueue nodes normally in the stack, check if it's empty through `isEmpty` method when we remove a node to start exploring we check whether it's visited or not. If it's visited we don't explore this node we return a null that will be handled in the general search algorithm. Otherwise, the state is saved in the `TreeMap` and we return this node.

6.3.3 Iterative Deepening Search

Iterative deepening search repeatedly calls a Depth Limited Search with different depth bounds starting by 1 up to infinity. Depth Limited Search is like depth-first search but it limits the depth of exploration which means any node with depth greater than the input depth won't be explored.

We implemented `DepthLimitedSearch` class as the `DepthFirst` with the exception that it now takes as an input the depth bound. It has the same data structure as a stack and the enqueue and `isEmpty` method is implemented normally. When removing a node, if its depth is greater than the depth limit we return null and handle this null in the general search method. For the memoization, we don't check only whether this state was visited before or not but we check the depth of the node having this state when we first visited it. If this depth is greater than the current depth we don't ignore this node. Figure 1 explains why. If a state visited with depth lower than the current node we are sure that this node doesn't lead to a goal state on depth less than the depth bound. However, if this state was visited on a depth greater than the current node, it may lead to a goal state but this goal state wasn't reached because of the depth limit.

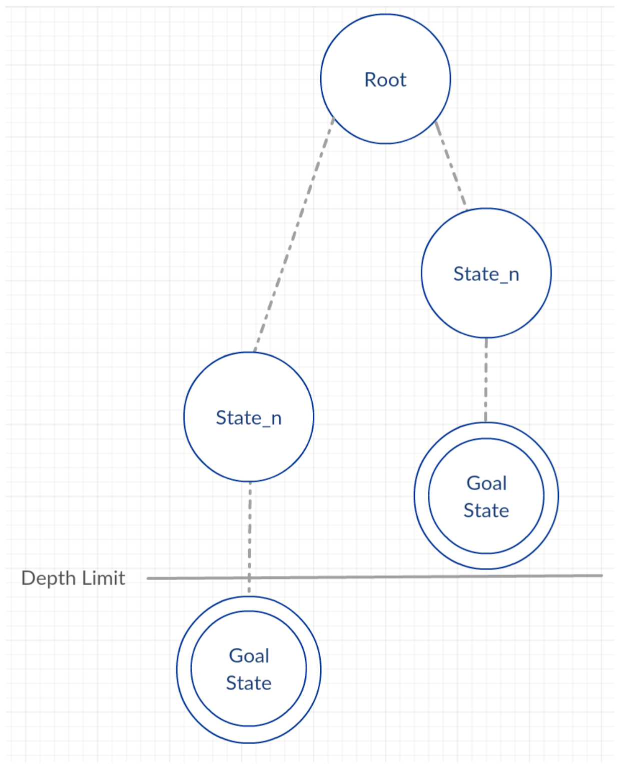

Figure 1: Iterative Deepening Memoization

For Iterative deepening, we implemented `IterativeDeep` class which will be `SearchQueue` object passed to the general search algorithm. When initialized, it initializes an object instance of the `DepthLimitedSearch` with depth equal to 0. When the general search algorithm calls the enqueue and remove front methods of this class, it calls the corresponding methods of the `DepthLimitedSearch` object. For `isEmpty` method, it first checks whether the queue inside the `DepthLimitedSearch` object is empty or not. If it's empty. it reset this instance with an increased depth by 1, new stack and a new `TreeMap` for the memoization.

6.3.4 Uniform Cost Search

So far, all the above algorithms doesn't take the cost of the paths into consideration. In uniform cost search (lowest cost first search), traverses the search tree like the breadth-first search but instead it expands the path with the lowest cost first. This guarantees to find a minimum path cost to a goal state, if any exists. In this particular problem, each operator is assigned a cost referenced in table 1. Each node has a cost value that represents the cost of the path from the root of the search tree to this node. This value is set to the cost of the parent node added to the cost of the operator that was applied to generate this node. For the algorithm to select the lowest cost path first, the data structure used in `UniformCost` class is Java `Priority Queue`. A `NodeComparator` object will be passed to the priority queue. A `UniformCostEvaluator` object will be passed when initializing the comparator. In `UniformCostEvaluator`, the evaluation of a node is based on the cost value in the node object. This will result in the priority queue sorting nodes based on their cost in ascending order. Enqueue and empty check is done normally. When removing a node, we check whether the node is visited before by a lower cost or not. If it was visited by lower a cost, a null is returned to be handled in the general search algorithm. if no lower cost is found in the tree map for the exact same state, the state is added with the new cost in the tree map and the search node is returned to the general search algorithm to explore this node.

6.4 Informed Search

Informed search algorithms use problem specific knowledge beyond the definition of the problem to find solutions more efficiently than the uninformed search strategies. Best first search is a general approach to these kind of strategies. Best first search has the ability to expand the most promising branches of the search tree based on some evaluation function. Evaluation function gives an insight about the goal direction.

6.4.1 Greedy Search

Greedy search tries to minimize the cost to reach the goal. That is done based on a heuristic evaluation function. This evaluation function estimates the cost of the cheapest path from the state at node n to a goal state. Greedy search strategy is to expand nodes with the least heuristic function value first. Since greedy strategy is like the uniform cost in the same aspect of sorting nodes based on some evaluation function and also memoization. We've decided to make `Greedy` a subclass from `UniformCost` class. The difference now that the `NodeComparator` is based on `GreedyEvaluator` instead of a `UniformCostEvaluator`. `GreedyEvaluator` object evaluates nodes based on a heuristic function that is passed to while initialized. The heuristic function will be one from the two mentioned in section 7.

6.4.2 A* Search

A* search strategy uses both lowest-cost-first and heuristic functions combined in its selection of which path to expand. Since A* strategy is like the uniform cost in expanding nodes with the lowest evaluation first and also memoization. We've decided to implement `AStar` class as a subclass of `UniformCost` class. The difference now that the `NodeComparator` is based on `AStarEvaluator` instead of a `UniformCostEvaluator`. `AStarEvaluator` object evaluates nodes based on an evaluation function which is the addition of heuristic function value at this node and the cost value at this node. The heuristic function will be one from the two mentioned in section 7.

7 Heuristic Functions

In this section, we present all the heuristic functions that we used in greedy and A* strategies. Each one of the three is admissible heuristic, an admissible heuristic function is a one that never overestimates the cost from a node to the closest goal state. The following subsections will discuss each of the three heuristic functions that we chose and argument why each function is admissible.

7.1 Cost to kill Heuristic

In this heuristic function, we estimate the cost of killing all the white walkers left. This cost depends on how many dragon glass pieces will be used. First, we estimate the number of dragon glass pieces to be used which is presented in equation 1 where X is the number of white walkers left for this node . Then, it returns the cost of kill value which is the multiplication of dragon glass pieces used by the kill cost. The kill cost is 3*(m+n) where m&n are the dimensions of the grid. This is presented in equation 2.

$$dg = \left\lceil \frac{X}{3} \right\rceil \tag{1}$$

$$h_1(n) = 3 \times (m+n) \times dg \tag{2}$$

Since this function estimates the cost for killing all the white walkers left. This cost is dependent on the number of dragon glass pieces used. This heuristic function never overestimates because it estimates the minimum number of dragon glass pieces to be used, which is in the case where all the white walkers left are in clusters of 3, and each cluster has a reachable cell which is adjacent to all the three white walkers in it. Then we define the division of the number of white walkers left by 3 to be the minimum estimate of the number of dragon glass pieces needed. A ceiling function is applied to this division because if the number of white walkers left is not divisible by 3, (i.e. one cluster remains with a white walker count equal to the non-zero remainder). The number of dragon glass pieces to be used will be this integer division added to a 1, which is the ceiling function. Thus, in all cases, since this heuristic gives us the minimum estimate, it never over estimates, therefore it is admissible.

7.2 Pickup Heuristic

In this heuristic function, we estimate the number of pickup operations that the search agent will apply. This is estimated by first calculating the number of dragon glass pieces to be used which is mentioned already in equation 1, let's call this Y. Then, the number of dragon glass pieces to be picked is the dragon glass pieces that the agent has in the current node state subtracted from Y. Since the agent will pickup the maximum number of dragon glass pieces it can hold, each time it applies the pickup operation, the number of pickups will be the previous result divided by the maximum number of dragon glass pieces the search agent can hold at any moment. If this result is negative, we set it 0 as that means no pickups are needed. More formally, this is represented by equation 3 where dg is the result of equation 1, d is the number of dragon glass pieces the agent is holding at this node and M is the max dragon glass pieces that the search agent can hold at any moment.

$$h_2(n) = \max(0, \left\lceil \frac{dg-d}{M} \right\rceil) \tag{3}$$

W	.	.	.	W	.
W	W	W	.	W	.
.	.	.	.	W	#
.	.	#	#	#	.
.	#	.	D	.	D
.	D	W	.	W	J

Table 4: Grid 1

This function estimates the minimum number of pickup operations done by the agent in order to kill all the white walkers left. The minimum pickup operations will be done if the search agent only picks up when it has 0 remaining dragon stone pieces. Since the number of dragon glass pieces needed is the number of the dragon glass pieces the agent is holding now subtracted from dg presented in equation 1. Then the minimum number of pickups will be the division of dragon stones needed by the maximum number the agent can hold. Thus, in all cases, since this heuristic gives us the minimum estimate, it never over estimates, therefore it is admissible.

7.3 Distance Heuristic

This heuristic estimates the max distance the search agent will walk to kill a white walker starting from the current node. To do that, we loop over all the left white walkers in the current state and check the distance between the white walker and the search agent, of those distances we return the max. This distance is defined as the Manhattan distance between those two minus 1. The Manhattan distance between two grid cells having coordinates (x1,y2) & (x2,y2) respectively is presented in equation 4.

$$distance = |x1 - x2| + |y1 - y2| \tag{4}$$

This function estimates the max distance the search agent will walk to kill a white walker. This distance can never be below the Manhattan distance between the search agent and the furthest white walker minus 1. Since the Manhattan distance assumes the map has no obstructions that the search agent can't step on, the minimum such distance can never be below this estimate. Thus, in all cases, since this heuristic gives us the minimum estimate, it never over estimates, therefore it is admissible.

8 Two running examples

In this section, we present two running examples each is a grid presented in figures 4 & 5. The symbols in those grids are presented in table 3.

9 Performance Comparison

In this section, we compare between different search strategies implemented on both examples presented in section 8. The performance measures are presented in tables 6 & 7. The measures included are completeness, optimality, expanded nodes, dragon

W	W	W	.	.
.	.	W	W	.
W	#	.	.	#
.	.	.	#	.
.	#	.	.	#
D	.	.	D	D
.	.	W	W	J

Table 5: Grid 2

glass pieces used and solution depth. In general, depth first search isn't complete but since we eliminate repeated states it's complete in this specific problem. Also, strategies like breadth first search, depth first search and iterative deepening search aren't complete algorithm but in some examples it can be.

For nodes expansion, depth first search have low expansion rate but it is not optimal overall. Uniform cost search expands a lot of nodes but on the other hand it's guaranteed to find the optimal solution. A* search with kill cost heuristic expands node the least nodes from all the optimal algorithms like uniform cost and A* with other heuristic functions.

10 Instructions how to run the program

To run the search program with different strategies and heuristic function, this can be done in the main method inside `Main.java` class.

1. A `SaveWesteros` object instance is initialized
2. A heuristic function is initialized, in case A* or Greedy strategies are used. The function can be one of the three mentioned in section 7.
3. Search method is called on the `SaveWesteros` instance with three parameters by this order: strategy search queue class, heuristic function instance & visualize Boolean variable.
4. In case of visualize Boolean variable is true, the grid after each action the search agent took in the path from the root to the goal state.

Search Strategy	Completeness	Optimality	Expanded Nodes	DG used	Solution Depth
Breadth First Search	YES	NO	3795	6	22
Depth First Search	YES	NO	50	9	49
Iterative Deepening Search	YES	NO	37381	6	22
Uniform Cost Search	YES	YES	29239	5	23
Greedy Search with kill heuristic	YES	NO	85	7	34
Greedy Search with pickup heuristic	YES	NO	106	9	48
Greedy Search with distance heuristic	YES	NO	255	9	38
A* Search with kill heuristic	YES	YES	3170	5	23
A* Search with pickup heuristic	YES	YES	29276	5	23
A* Search with distance heuristic	YES	YES	28205	5	23

Table 6: Performance Comparison for Search Strategies on Grid 1

Search Strategy	Completeness	Optimality	Expanded Nodes	DG used	Solution Depth
Breadth First Search	YES	YES	2035	6	16
Depth First Search	YES	YES	61	6	45
Iterative Deepening Search	YES	YES	10520	6	16
Uniform Cost Search	YES	YES	28151	6	16
Greedy Search with kill heuristic	YES	NO	101	7	29
Greedy Search with pickup heuristic	YES	NO	124	8	47
Greedy Search with distance heuristic	YES	NO	3898	8	79
A* Search with kill heuristic	YES	YES	13405	6	16
A* Search with pickup heuristic	YES	YES	28176	6	16
A* Search with distance heuristic	YES	YES	28197	6	16

Table 7: Performance Comparison for Search Strategies on Grid 2