The German University in Cairo

Media Engineering and Technology

Endgame AI Agent

AI Project #1 Report

Ahmed Mohamed Fawzy Ahmed Darwish	37-9080
Omar Mohamed Youssef Elkilany	37-0732
Nesrin Abdulaziz Alsayed Attia	37-16218

Table of Contents

Description of the Problem

The Endgame problem is a search problem where a rational agent (Iron Man) has to traverse an $n \times m$ grid to collect 6 *stones*, while avoiding being attacked/attacking x warriors if possible, where $x \geq 5$. After collecting all the stones, the agent should go to the goal grid cell where Thanos exists. Thanos attacks the agent whenever the agent is in a cell adjacent to his, where adjacency is defined by having a manhattan distance of 1 unit. Furthermore, if Thanos and the agent are in the same cell, the agent receives damage. Thanos's warriors attack the agent along the way if the agent is in a location adjacent to theirs. If the agent arrives at the goal grid cell with less than 100 units of damage, the search problem is solved, and the agent is able to perform the *snap* that kills Thanos.

The available operators for the agent include: *up, down, left, right*, which change the location of the agent with the expected semantics. Movement to non-Thanos locations is possible if and only if the new location has no living warrior and is within the boundaries of the grid. Movement to the Thanos location is allowed only after collecting all the stones. The operator *collect* causes the agent to collect the stone present in his own cell if such stone exists. The operator *kill* is an action where the agent kills all the warriors in adjacent cells. Finally, the *snap* operator is used to kill Thanos as long as all stones are collected. The *snap* operator has no effect on the search problem; therefore, it was not implemented as part of it.

The cost function of our problem is summarised in the following table:

Operator	Cost
up, down, left, right	$1 \times adjW(targLoc) + 5 \times adjThanos(targLoc)$
kill	$2 \times adjW(currLoc) + 5 \times adjThanos(currLoc)$
collect	$3 + 1 \times adjW(currLoc) + 5 \times adjThanos(currLoc)$
snap	0

where $adjW(location)$ is a function that returns the number of warriors adjacent to $location$; $adjThanos(location)$ is a function that returns 1 if Thanos is adjacent or in the $location$ and zero otherwise; $targLoc$ is Iron Man's location after the action; and $currLoc$ is Iron Man's location before the action.

Our state consists of a point (x, y) representing the agent's location, a bitmask representing the state of each of the stones in the grid (whether collected or not), and another bitmask representing the state of the warriors (whether dead or alive). We did consider including the agent's health as part of our state (with an initial value of 100); however, due to runtime limitations resulting from the added complexity, we decided to remove it and to use the path cost as an indicator of health since the two quantities are semantically equivalent. This failure to include the agent's health in the state has resulted in complications during the handling of repeated states. These complications will be discussed further in the latter sections of the report.

As mentioned before, a search tree node is considered to be the goal node if and only if the path cost is less than 100, all stones have been collected, and the agent is present in the goal grid cell.

The Search-tree Node Class Implementation

The search-tree node is represented in our code by the *Node* class. The *Node* class has five private attributes. The first is a reference to the parent node, which helps in the construction of the final solution. The second is the state of the node. *State* is a class that contains a *HashMap* of labels and values. Each label corresponds to a certain value. For example, in the Endgame problem, one state label is *ironMan* and the corresponding value is a point that represents a location in the grid. The third private attribute of the *Node* class is the path cost. The path cost is the cost from the root node, which contains the initial state, to the current node.

The fourth private attribute of the *Node* class is the operator that was applied to the parent node to produce the current node. The operator is represented by a string and has meaning only within the context of a particular problem. The fifth attribute of the *Node* class is the depth. The depth is 0 for the root node. It is calculated as $parentNode.depth + 1$ for all the other nodes that are created in the search-tree. Therefore, our code protects the depth variable from any external manipulations to minimize the possibility of error. The final attribute is the *heuristicCost*, which represents the estimated cost of reaching the goal node from the current node. The value of this attribute depends on the heuristic function being used for the search. It is used only for the greedy and A* search algorithms.

The *Node* class contains getters to retrieve the values of these private attributes and a *toString()* method that returns the state, path cost, and operator as a readable string.

The Search Problem Class Implementation

The search problem is represented in our code as the *GenericSearchProblem* class. It contains four attributes. The first is the expanded nodes counter. This variable is used to count the number of nodes that the search algorithm expanded until it arrived at the result. The second attribute is an array of strings that represents the operators of the problem. These operators can be applied to produce nodes with new states. The third attribute is the initial state, which is a *State* class instance that describes the starting state of the problem. The root node of the search tree is created using this initial state. The last attribute of the class is a *HashSet* that represents the visited states. This *HashSet* was used to meet the runtime requirements. Nodes with repeated states are not expanded to save time.

The *GenericSearchProblem* class contains four abstract methods. The first is the *getPathCost()* method, which takes as input a parent node, the operator that was applied to the parent node, and the state that resulted from the application of this operator. The method returns a numerical value that represents the cost of reaching this node. The three method

parameters were chosen for generality. The path cost is then defined specifically for each unique problem.

The second abstract method is *applyOperator()*, which takes as an input a node and an operator. The method returns the state that results from applying the operator to the node. The third abstract method is the *isGoalState()* method, which defines the test that should be passed to consider a state to be a goal state. For generality, the method takes a node as an input. It is then supposed to check the state of the node along with any other node parameters needed to satisfy a problem's goal. The method returns a boolean value, which is true if the node's state is a goal.

The last abstract method is *evaluateHeuristic()*, which was included for convenience and code simplification. The method is inherited by any specific problem and can be implemented to reflect the heuristic functions for the search algorithms that require such heuristics. The reason for its inclusion in the *GenericSearchProblem* class is that this way it has access to any global variables in a problem. In our code, for example, we represented Thanos's position as a global variable since it never changes. Alternatively, the method can be placed in another class without a loss of generality.

GenericSearchProblem also contains three implemented methods. These methods are *expand()*, *resetVisitedStates()*, and *getFullSolution()*.

expand() is used by any search algorithm to expand the nodes in the search problem. It takes a dequeued node as an input and returns an *ArrayList* of expanded nodes. If the state of the input node was visited before, the node is not expanded. The method has access to the operators of the problem and applies them sequentially to the input node. The results of inapplicable operators are ignored. The cost of the operation is calculated inside this method. Finally, if the node was expanded successfully, the expanded nodes counter is incremented.

resetVistiedStates() resets the *HashSet* that keeps track of the visited states.

getFullSolution() is a static method that takes a goal node as an input and returns a stack that contains the full solution from the initial state to this goal. It was included in the *GenericSearchProblem* class because it was assumed that any general search problem is bound to need a solution constructor. It can be moved to any other class without a loss in generality.

The Endgame Problem Implementation

Our *Endgame* class extends the *GenericSearchProblem* class, inheriting all the aforementioned attributes and methods. The implementation of the abstract methods will be explained in detail below. Any Endgame-specific attributes and methods will be discussed as well.

The *Endgame* class contains several additional attributes, which make the state representation of our problem more efficient by saving space. These attributes never change; therefore, they were not included in the state. *thanosPos* represents the location of

the goal grid cell, which contains Thanos. *warriorsIdx* are data structures containing the locations of all the warriors, where the *warriorsIdx* TreeMap provides an efficient way of checking whether a certain cell contains a warrior (whether dead or alive). The same goes for *stonesIdx*, which do the same function but for the stones. *rows* and *columns* represent the values n and m of the grid and are used for validity checks to avoid movements outside the grid boundaries.

Next up are the static variables and the *stateContents* enum. The static integer values *thanosAttack, warriorAttack, warriorKill,* and *stoneCollect* are used to ensure consistency when the cost of an action is calculated. They store the value of their respective costs according to the obvious semantics. Similarly, the *stateContents* enum is used to make sure that the state contents are consistent throughout the search process. It contains the only allowed objects an Endgame state may hold. As for the enum, it is simply used to ensure consistency when accessing the state object. It does that by containing the string value used to access the hashmap in an attribute called *label*.

The state contents were mentioned previously in the description of the problem. We implemented them in the same way: The agent location is represented as a *Point* object, which consists of instance variables *x* and *y*. *x* is the row and *y* is the column Iron Man is in. A byte is used to represent the stone's bitmask since there are only 6 stones. We map the stones to the least significant 6 bits of the byte. On the other hand, the domain of the number of warriors in the Endgame problem necessitated the use of a *BitSet* object, which can represent any number of bits as required.

applyOperator() receives an operator string as an input and applies the appropriate operations to produce the result state as follows:

- If the operator is *kill*, we first check for the existence of any living warriors in the adjacent cells. If none exist, we invalidate the operator. Otherwise, we set the bit for all the adjacent warriors to 0 in the result state.
- If the operator is *collect*, we check for the existence of a non-collected stone in the agent's location. If no such stone exists, we invalidate the operator. If the validity check is passed, the bit representing the stone is set to 0.
- If the operator is any of the movement actions (*left, right, up, down*), we check that the resulting location is inside the grid and then make sure that the new location does not contain a warrior. Furthermore, if the location is Thanos's, we ensure that Iron Man has all the six stones. If these rules are breached, the operator is invalidated.
- We decided not to include the operator *snap* in the list of the problem's operators for simplicity. We consider arriving at the goal grid cell with all stones collected and less than 100 points of damage to be sufficient to imply that the snap action will take place.

After applying the operator, *getPathCost()* calculates the damage the agent receives from its last action according to the table that we provided in the problem description section.

For informed search strategies—as mentioned before—the *evaluateHeuristic()* method is used to calculate the values of the heuristic functions. To optimise the running

time of our implementation, we utilize the *heuristicCost* attribute of the *Node* class to calculate the value only once for any node. If the *heuristicCost* attribute is not null, we return the value calculated before.

Finally, the *isGoalState()* function checks whether a node contains the goal state..

In addition to the abstract methods, we have implemented several helpers to make validity checks smoother and more organized:

- *isInsideGrid()* is used to check whether a pair of coordinates corresponds to a point inside the problem's grid borders.
- *isWarrior()* is used to check whether a cell contains a living warrior at a certain location in a particular state.
- *isStone()* is similarly used to check whether there exists an uncollected stone at a certain location in a particular state. However, it is not used for optimization reasons.

Description of the Functions in *Main*

The *Main* class contains several methods along with their helpers. The *solve()* method, which was mentioned in the project description, first passes a string object representing the grid to the *parser()* helper method. The parser performs the necessary string analysis, creates an instance of *Endgame*, and returns this instance to the *solve()* method.

Afterward, a switch statement takes as input the *strategy* string and creates the chosen search algorithm instance. If the chosen algorithm is not IDS, the algorithm and the problem instance are both passed to *performGeneralSearch()*, which implements the general search pseudo-code present in the textbook and the lecture. Otherwise, *performIDS()* is invoked. *performIDS()* takes the problem instance as input and uses the DLS algorithm in each iteration. If a solution node is found, *performIDS()* returns it; otherwise, *performIDS()* loops forever.

If a goal node is found and the *visualize* parameter is true, *visualizeSolution()* is invoked. It first creates a stack of nodes describing the full solution from the initial state to the goal state. Then it loops over the elements of this stack and builds a string of visualizations. Each visualization shows the action performed, the cost/damage of the action, and the grid state after the action is performed.

Finally, *constructSolution()* is invoked. It returns the solution string, containing the list of all the operators needed to reach the goal node, as well as the cost and the number of expanded nodes.

The Implementation of the Search Algorithms

To keep our search algorithms consistent with the implementation of *performGeneralSearch()*, we implemented an interface called *searchAlgorithm*, which enforces the implementation of the following methods:

- *makeQueue()*, which instantiates the data structure of the search strategy and adds the root node to it.
- *enqueue()*, which adds nodes to the data structure according to the data structure's insertion rules.
- *dequeue()*, which simply dequeues the node at the head of the data structure.
- *isQueueEmpty()*, which returns true if and only if the data structure is empty.
- *printQueue(),* which prints the data structure's content for debugging purposes.

The detailed implementation of each search algorithm required by the project is provided below.

Breadth-first Search (BFS)

We used a *LinkedList* with the *Queue* interface as the data structure for our BFS implementation. The *Queue* interface handles data in a first-in-first-out (FIFO) fashion. *makeQueue* instantiates an empty *Queue* and adds the root node. The rest of the methods include the functionally equivalent methods from the *Queue* class: *addAll(), remove(), isEmpty(),* and *toString()*.

Depth-first Search (DFS)

For DFS, the Java *Stack* class was used because of its first-in-last-out (FILO) data handling. *makeQueue* simple instantiates the stack and adds the root node to it. The *Stack* class methods *addAll(), pop(), isEmpty(),* and *toString()* are used to implement the interface functions.

Uniform Cost Search (UCS)

Because UCS requires sorting based on the path cost, the *PriorityQueue* class is used in its implementation. In *makeQueue(),* a lambda function is passed to the constructor of the queue. This lambda function defines the comparison criteria for the sorting. Again, *addAll(), poll(), isEmpty(),* and *toString()* are used in their respective methods.

Depth-limited Search (DLS) and Iterative Deepening Search (IDS)

To perform IDS, a DLS class was first implemented. It takes in the constructor a *limit* parameter, representing the greatest depth the nodes are allowed to have. In the *enqueue* method, the depth of each node is compared to the *limit* and is accordingly added to the frontier or not. The rest of the functions are implemented in the same manner as in DFS.

The *performIDS()* method in our *Main* class takes as an input the problem instance and then loops, performing DLS in each iteration and incrementing the limit (which is initially 0) by one until a goal is found.

Informed Search Algorithms (A* and Greedy Search)

Both *Greedy* and *AStar* classes are implemented in almost the same manner as UCS. However, they take as an input a problem instance and an integer, representing the heuristic they are supposed to follow. Those values are then used in the lambda function of the *PriorityQueue* in the following manner:

- For *Greedy*, the queue sorts the nodes according to the value of the problem's *evaluateHeuristic()* method only.
- For *AStar*, the queue sorts the nodes according to the value of the problem's *evaluateHeuristic()* as well as the *pathCost* of the node.

Heuristic Functions

We defined two admissible heuristic functions and used them for both greedy and A* searches. Both functions are designed to help Iron Man reach the goal state with lower costs. The second heuristic function is presented first for ease of discussion.

The Second Heuristic Function

The second heuristic function is the remaining stones-collection cost that Iron Man must incur before he reaches the goal. This function was implemented in the *getRemainingStonesDamage()* method in the *Endgame* class. Its goal is to prioritize states with less uncollected stones.

$h_2(n)$ is defined for the current node n and the current state s as follows:

$h_2(n) = 3 \times N$, where N is the number of remaining uncollected stones in s.

Each stone causes 3 points of damage when it is picked up, and Iron Man must collect all stones in order to enter the goal state. Therefore, to arrive at any goal state, Iron Man will inevitably receive $3 \times 6 = 18$ points of damage that will be added to the cost. This heuristic function represents this fact and assigns a cost of 18 to the states with 6 uncollected stones. If there are only 5 stones left, the function will assign a cost of 15, and so on, never overestimating the cost to the goal. Therefore, $h_2(n) \leq h^*(n)$, where $h^*(n)$ is the actual path cost from node n to the nearest goal node. Accordingly, $h_2(n)$ is admissible. Furthermore, because (i) a cost of 0 is assigned to the states where all the stones are collected and (ii) all goal states are states where all the stones are collected, the function satisfies the centering property.

The First Heuristic Function

The first heuristic function is the warrior damage that would result from one future step added to the damage that would result from collecting the remaining stones.

This function was implemented in the *getOneMoveWarriorDamage()* and *getRemainingStonesDamage()* methods in the *Endgame* class. Its goal is to put Iron Man in states that will help him avoid the warriors and thus lessen their contribution to the solution cost.

For a state s, where Iron Man is in a location (i, j), $h_1(n)$ is defined as follows:

If s is a goal state:

$$h_1(n) = 0 + h_2(n)$$

If the current state is not a goal state:

$$h_1(n) = min(adjW(i+1, j),\ adjW(i-1, j),\ adjW(i, j+1),\ adjW(i, j-1)) + h_2(n)$$

where $adjW$ is a function that returns the number of warriors adjacent to the cell (i, j).

This heuristic function is obtained by relaxing the problem into one where the damage from the warriors in the next move and the number of remaining stones matter. If Iron Man is in the goal cell, the function will return 0. Therefore, the centering property is satisfied. Collecting the stones is inevitable; therefore, the inclusion of $h_2(n)$ is justifiably admissible. If Iron Man is not in the goal cell, he must move using *Right*, *Left*, *Up*, or *Down* to reach the goal. Even if he performs a *Kill* or a *Collect*, movement is inevitable in the sequence to the goal. Thus, it is established that at least one move must follow. This single move will lead Iron Man to one of the four locations $(i+1, j)$, $(i-1, j)$, $(i, j+1)$, and $(i, j-1)$. Any warriors adjacent to Iron Man in that future location will attack him. Therefore, in the best possible case, he will incur the minimum damage of these four possible damages.

Since all the costs are non-negative, the cost from the state s to the goal state must include the value $h_1(n)$ as part of it. Therefore, $h_1(n) \leq h^*(n)$, where $h^*(n)$ is the actual path cost from node n to the nearest goal node. Accordingly, $h_1(n)$ is admissible.

We attempted to use *getOneMoveWarriorDamage()* alone as the admissible heuristic; however, it performed poorly in grids with a few warriors. Furthermore, due to the limitations in the keeping of the repeated states, the greedy search returned no solution instead of a solution. This is mainly due to the fact that we do not keep track of damage in the state. States are visited with high costs and then ignored when they are visited again at lower costs. If we include the damage in the state, *getOneMoveWarriorDamage()* will perform as expected and produce solutions when they exist.

Experiments and Performance

Example Runs of the Implementation

Input #1: grid = "5,5;0,0;4,4;0,1,0,2,0,3,0,4,1,4,2,4;2,0,3,0,3,1,4,0,4,1"
strategy = “BF”
visualize = true

Output #1:

Step #0: damage = 0, step performed is: null

I	S	S	S	S
-	-	-	-	S
W	-	-	-	S
W	W	-	-	-
W	W	-	-	T

Step #1: damage = 0, step performed is: right

-	I, S	S	S	S
-	-	-	-	S
W	-	-	-	S
W	W	-	-	-
W	W	-	-	T

Step #2: damage = 3, step performed is: collect

-	I	S	S	S
-	-	-	-	S
W	-	-	-	S
W	W	-	-	-
W	W	-	-	T

Step #3: damage = 3, step performed is: right

-	-	I, S	S	S
-	-	-	-	S
W	-	-	-	S
W	W	-	-	-
W	W	-	-	T

Step #4: damage = 6, step performed is: collect

-	-	I	S	S
-	-	-	-	S
W	-	-	-	S
W	W	-	-	-
W	W	-	-	T

Step #5: damage = 6, step performed is: right

-	-	-	I, S	S
-	-	-	-	S
W	-	-	-	S
W	W	-	-	-

| W | W | - | - | T |

Step #6: damage = 9, step performed is: collect

-	-	-	I	S
-	-	-	-	S
W	-	-	-	S
W	W	-	-	-
W	W	-	-	T

Step #7: damage = 9, step performed is: right

-	-	-	-	I, S
-	-	-	-	S
W	-	-	-	S
W	W	-	-	-
W	W	-	-	T

Step #8: damage = 12, step performed is: collect

-	-	-	-	I
-	-	-	-	S
W	-	-	-	S
W	W	-	-	-
W	W	-	-	T

Step #9: damage = 12, step performed is: down

-	-	-	-	-
-	-	-	-	I, S
W	-	-	-	S
W	W	-	-	-
W	W	-	-	T

Step #10: damage = 15, step performed is: collect

-	-	-	-	-
-	-	-	-	I
W	-	-	-	S
W	W	-	-	-
W	W	-	-	T

Step #11: damage = 15, step performed is: down

-	-	-	-	-
-	-	-	-	-
W	-	-	-	I, S
W	W	-	-	-
W	W	-	-	T

Step #12: damage = 18, step performed is: collect

-	-	-	-	-
-	-	-	-	-
W	-	-	-	I
W	W	-	-	-
W	W	-	-	T

Step #13: damage = 23, step performed is: down

-	-	-	-	-
-	-	-	-	-
W	-	-	-	-
W	W	-	-	I
W	W	-	-	T

Step #14: damage = 28, step performed is: down

-	-	-	-	-
-	-	-	-	-
W	-	-	-	-
W	W	-	-	-
W	W	-	-	T, I

right,collect,right,collect,right,collect,right,collect,down,collect,down,collect,down,down,snap;28;4403

Input #2: grid ="14,14;0,0;9,9;8,6,9,4,7,1,4,4,4,7,2,3;8,13,0,4,0,8,5,7,10,0"
strategy = “UC”
visualize = false

Output #2:

down,right,right,right,down,collect,left,down,down,right,right,collect,down,left,left,left,down,down,collect,up,up,right,right,up,up,right,right,down,right,up,right,down,collect,right,right,down,down,down,left,left,left,down,collect,left,left,down,collect,down,down,right,right,right,down,right,right,down,right,up,right,right,up,left,up,right,up,left,left,left,snap;30;391636

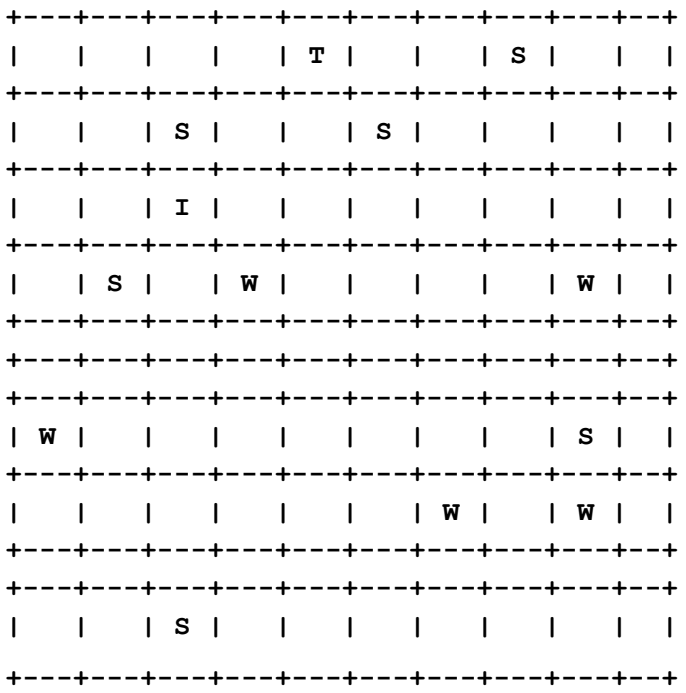
Performance Comparisons

To compare the performance of our different algorithms, we try three grids of different sizes, and compare the cost of the output goal and the number of expanded nodes for each algorithm as well as the different heuristics of informed search. The three grids are:

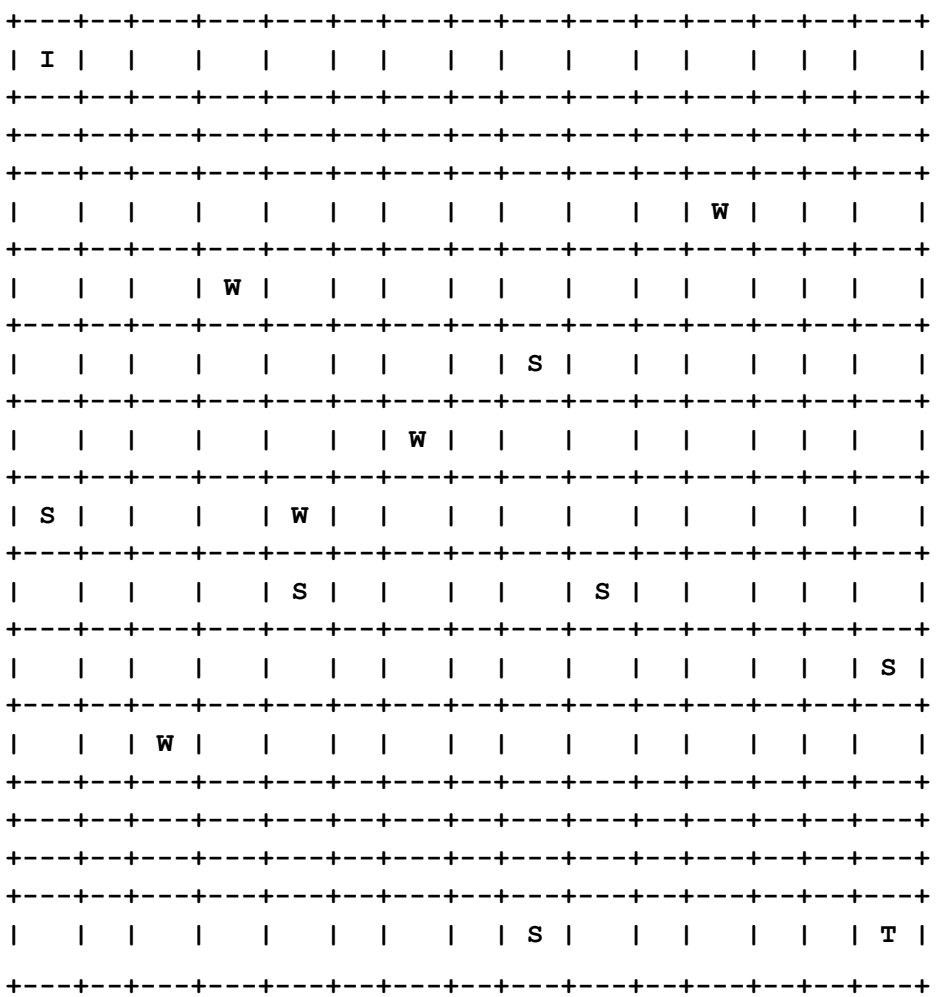
5x5 Grid

W				
		S		
	S	I		S
S		W		W
S	S	T	W	W

10x10 Grid



15x15 Grid



Grid Size	Strategy	Cost of found goal	Number of expanded nodes
5x5	BFS	40	12224
	DFS	92	152
	UCS	37	24375
	IDS	41	53769
	Greedy (Heuristic 1)	58	76
	Greedy (Heuristic 2)	58	76
	A* (Heuristic 1)	37	20637
	A* (Heuristic 2)	37	20637
10x10	BFS	35	190885
	DFS	53	676
	UCS	30	196452
	IDS	49	2962248
	Greedy (Heuristic 1)	56	440
	Greedy (Heuristic 2)	56	440
	A* (Heuristic 1)	30	171413
	A* (Heuristic 2)	30	171413
15x15	BFS	30	355132
	DFS	43	1258
	UCS	30	450738
	IDS	40	3120451
	Greedy (Heuristic 1)	42	328
	Greedy (Heuristic 2)	42	328
	A* (Heuristic 1)	30	428651
	A* (Heuristic 2)	30	428651

Comments on the Results of the Comparison

As expected, UCS and A* provided optimal solutions for all the tested grid sizes. In addition, A* always expanded less nodes than UCS regardless of the heuristic function used for the search. It can be noted that the number of expanded nodes for both heuristics was the same in A*. This was due to the relatively low number of warriors in the test grids. In problem instances with higher numbers of warriors, the heuristics expanded different numbers of nodes. Greedy Search and DFS expanded nodes the least. However, that came at the cost of optimality as the path costs of their solutions were the worst among other solutions. IDS expanded nodes the most for all tests, producing moderately optimal solutions. Finally, BFS showed a good balance between optimality and path costs.

For all the grids tested, the algorithms were able to find solutions. However, this does not mean that they are all complete. Since we have not considered the damage as part of the state (even though it affects the next step, which can be invalid if the damage exceeded 100), there will be cases where DFS, BFS, and IDS will fail to find the existing solution. Let s_p be a state in the single correct sequence to the goal. In these search strategies, which do not necessarily explore the nodes with the smallest cost first, s_p can be visited with a high damage/cost, for example, 99 or 100. In this case, when the operators are applied, the node might fail to produce the nodes that lead to the goal because of the high damage. During the remainder of the search, we may visit s_p with a smaller damage/cost that will allow us to do some operations and reach the goal; however, because s_p was already visited (regardless of the cost), we disregard it and do not expand it again. Therefore, in such cases, our versions of BFS, IDS, and DFS will not find solutions and are consequently not complete.

Because we check on the repeated states after the dequeue operation (instead of the expand operation), our implementations of UCS and A* are both optimal and complete. A solution will always be found at the lowest cost possible if it exists. This is a natural result of the sorting operation, which is used to order the queue.

The optimality and completeness of greedy search depends on the problem and the heuristics. Theoretically, greedy search is neither complete nor optimal. For our implementation, greedy search is also neither complete nor optimal due to the handling of the repeated states without the consideration of the cost and due to the poor performance of the heuristics in approximating the final cost.

In the following page, there is a comparison of the theoretically-proven completeness and optimality of the search strategies against their performance in our implementation.

	Theoretically-proven		Actual (Our Implementation)	
Strategy	Completeness	Optimality	Completeness	Optimality
BFS	✔	✘	✘	✘
DFS	✘	✘	✘	✘
UCS	✔	✔	✔	✔
IDS	✔	✘	✘	✘
Greedy	✘	✘	✘	✘
A* (assuming admissible heuristic)	✔	✔	✔	✔

References and Citations

1. Java™ Platform, Standard Edition 7 API Specification. (2017). Retrieved November 3, 2019, from https://docs.oracle.com/javase/7/docs/api/.
2. Russell, S. J., & Norvig, P. (2010). *Artificial intelligence: a modern approach* (3rd ed.). Upper Saddle River: Prentice-Hall.