Project I Report

# Section 0: Introduction

This report consists of 8 sections. Section 1 will give a brief description of the problem. Section 2 will cover the design followed to implement the search tree node class, the general search problem, and the implementation of the EndGame problem. Inside the main class some methods were created to aid the aim of the project. Those will be discussed in section 3. Next, section 4 will discuss the implementation of the different search algorithms. Section 5 will discuss the implementation and design of the heuristic functions used for the A\* and Greedy Algorithms. Section 6 will show 2 functional examples. Finally, section 7 will compare and contrast the different search algorithm in terms of performance, completeness, optimality, and the number of expanded nodes. Finally section 8 will discuss one error which we got during the testing phase.

# Section 1: Description of the Given Problem

The goal of this search agent is to kill Thanos, after collecting the 6 infinity stones, even if that means that not all warriors will be killed. Iron Man can move in the four directions: up, down, left, right. In addition, it can collect a stone, only if it is in a cell which has a stone. This reflects a damage of **3** on Iron Man, for each stone collected. Besides, it can kill warriors in adjacent cells. This reflects a damage of **2\* the number of adjacent warriors** on Iron Man. Being adjacent to a warrior causes a damage of **1** on Iron Man, **for each adjacent warrior**. As for the case of Thanos, being adjacent to it causes a damage of **5** on Iron Man. To kill Thanos, through the *snap* operator, Iron Man’s damage should be **less than 100 and all the stones should be collected**. This whole problem is the EndGame Search Problem. The goal is to implement a general search problem, which handles any search problem given to it, whether EndGame search problem or any other search problem, using different search strategies. The next section discusses the implementation of the SearchTreeNode, the SearchProblem, and the EndGame classes.

# Section 2: Classes’ Implementation

### SearchTreeNode.java

The *SearchTreeNode* class defines the Search Tree Nodes of any search problem. This class is generic and should be usable by any search problem. The following attributes define the class:

1. ***State state****,* which defines the state associated with the current node*.*
2. ***SearchTreeNode parentNode****,* which defines the parent node associated with the current node*.*
3. **String *operator***, which is a string that indicates the operator applied to produce this node.
4. ***int depth****,* which specifies the depth of that node.
5. ***int pathCost***, specifies the integer cost to reach that current node.
6. ***int heuristic,*** specifics the heuristic cost for the node. However, if the node is not involved in A\* or Greedy search this attribute is set to -1, indicating that this attribute is of no use in the current search strategy.
7. ***int heuristictype***, this attribute is used to specify whether the first **Greedy/A\* (GR1/AS1)** search strategy is going to be applied on the SearchTreeNode, or the second ones. If this attribute is set to 1, then the first heuristic is going to be applied on it. However, if it is set to 2, then the second heuristic is going to be applied.

With those 7 attributes, the *SearchTreeNode* is generic and can be used to create any SearchTreeNode for any search problem.

### SearchProblem.java

The *SearchProblem* is an **abstract** class used to represent any search problem. The attributes of the SearchProblem class are as follows:

1. ***String[] operators;*** this attribute is an array of Strings of the operators which are going to be used for the search problem.
2. ***State initialState***; this attribute is used to carry the initial state of the search problem
3. ***Set<String> repeatedStates***; this set is a hashset which is used to hold the visited states of the problem, to avoid going into a cycle if they are revisited. However, the search problem constructor does not take it as a parameter, it only initializes it to an empty hashset in its body.

SearchProblem has an abstract *transition*, *PathCost*, *PathCostVisualize* and *GoalTest* methods. The *transition* function Is an abstract one. It basically performs a transition on the *SearchTreeNodes* passed to it by applying an operator on it. Also, it returns the resulting SearchTreeNode resulting from the transition. The *PathCost* function takes a SearchTreeNode and returns a string which is the sequence of operators applied to reach the goal node. The *GoalTest* function is an abstract Boolean method which has an intent of observing the SearchTreeNode and check whether it’s a goal SearchTreeNode or not. Finally, an abstract void method called *PathCostVisualize* takes as a parameter a SearchTreeNode and its role is display every grid from the root to the SearchTreeNode provided as a parameter to the method. (Grid of the whole path).

In addition to those abstract method, there is a major method inside this class which is **not** abstract. That is, *GeneralSearch*. The *GeneralSearch* method takes as parameters a search problem, the search strategy to be applied, and a Boolean parameter *visualize,* which specifies whether or not the grid at each step to the goal should be displayed.

The method’s body creates an initial SearchTreeNode from the search problem’s initial state. If the strategy passed to the method is AS1 or GR1 then the SearchTreeNode heuristic type is going to be 1. However, if the strategy passed to the method is AS2 or GR2 then the SearchTreeNode heuristic type is going to be 2. Otherwise, it is 0. It then checks the search strategy passed and uses this strategy to find the search problem’s goal accordingly. This is done through a switch statement. When the strategy is found, its logic is applied to the search problem. If no goal is found a *null* string is returned*.* Otherwise, the sequence of operators applied to reach goal, the path cost to the goal, and the number of nodes expanded are returned as a string separated by semicolons.

Also, in the *GeneralSearch* method different data structures were used for different searches. For the Breadth-First Search, for example, a queue was used. However, for the A\*, Greedy, and Uniform-Cost Search a priority queue was used, and the comparator was overridden (This will be discussed in section 4). Meanwhile, for the Iterative Deeping Search and Depth-First Search a stack was used inside the General Search function to perform the strategy. More details of the General Search will be discussed in section 4 along with the Search Strategies

### EndGame.java

Before getting into the implementation of EndGame. There is some foundation to be known. The *State* class, which represent a generic state, is empty. However, there is another class called *EndGameState*, which has the state in a form of a string as an attribute to it. This string is typical to the grid. However, concatenated to it, are two more things: the path cost of the state’s node, and a true or false string to represent whether or not Thanos is alive.

1. **String grid;** this string is semicolon separated, where stones, iron man, and Thanos positions, are comma separated. The same applied for the grid dimension. The damage and the stringified Boolean which represents if Thanos is alive or not, are only semi colon separated.

The EndGame class is a subclass of the SearchProblem. Therefore, all the abstract methods which were created in the SearchProblem had to exist in EndGame, and be implemented. Firstly, the EndGame constructor calls the super constructor. It has no addition parameters within it. There are several helper methods which were created to solve the EndGame search problem. The first one is *adjThanos,* which takes as parameters the integer x and y position of Iron Man and Thanos, respectively. It returns true iff Thanos is adjacent to Iron Man. Otherwise, it returns false. Besides, there is another method, *adjWarriors*, which is used to return the count of the adjacent warriors to Iron Man. *checkWarrior*, however, is a Boolean method used to check if there is a warrior in the cell Iron Man wants to move to. A method called *killWarriors* is used to remove the adjacent warriors killed from the array list they exist within. (Putting the values within the EndGameState string in data structures will be discussed later).

As for implementing the methods inherited from the parent class *SearchProblem*, the *PathCost* method takes a SearchTreeNode as a parameter and returns the sequence of operators used to reach a current SearchTreeNode. Within the GoalTest method, the last element of the EndGameState string is checked, the stringified Boolean which tells whether Thanos is alive or not. In case it is dead, the GoalTest returns true. Otherwise, it returns false. The *PathCostVisualize* takes as a parameter a SearchTreeNode and its role is display every grid from the root to the SearchTreeNode provided as a parameter to the method. (Grid of the whole path).

Finally, the transition method, of SearchTreeNode return type, takes a string operator to be applied on a SearchTreeNode, which is also passed to the transition function. The transition method accesses the EndGameState within the SearchTreeNode. The *grid* String attribute is then accessed at the beginning of the method. The String grid which represents the EndGameState is separated by semi colons. At the beginning of the transition function the rows and columns of the grid are placed in two (2) int variables. Likewise, the x,y position of iron man are placed in two (2) int variables. The same goes for Thanos x and y positions. Next, the stones positions are separated by commas in the EndGameState grid. Therefore, every two consecutive number are placed as a Point object in an ArrayList of type Point (Point class is imported in EndGame). Another ArrayList was created for warriors to serve the same purpose. The grid’s damage was placed in an int variable. Finally, the stringified Boolean was converted to Boolean and placed in a Boolean variable, isAlive. If the damage is the SearchTreeNode’s state is more than 100, null is return. Otherwise, the operator is checked.

If the operator is *collect*, a check is made to make sure that Iron Man is standing in a cell with a stone. If that is not the case, null is returned, as this action cannot be performed except if Iron Man exists in a cell with a stone. If that is the case, a damage of 3 is added to the variable which carries the parameter SearchTreeNode’s damage. Also, checks are made to make sure no adjacent warriors exist, using the helper function *adjWarriors*, discussed above. If they do exist, a damage of their number is added to the variable which carries the damage. Using *adjThanos* helpermethod, it can be identified whether Thanos is adjacent to iron man or not. If that is the case, a damage of 5 is added. Next, there is a method called generateEGS, it takes all the variables and data structures we have created from the EndGameState grid String at the beginning of the transition function, and converts them back to a string. We use that method to generate the new state and store it in a variable, say *str*. What we need to do now, is to check if a state is repeated or not. Therefore, we take this new string, and crop the last 2 values separated by semicolon from it, the stringified damage and Boolean which tells if Thanos is alive or not, in another variable. We then call the EndGame *repeatedStates* HashSet and compare if a string value like the cropped one exists within it. If that is the case, a null is returned because the state is already visited. Otherwise, a new SearchTreeNode is created with the a new EndGameState which is the generated string from *generateEGS* before cropping. The new SearchTreeNode will also contain the parent SearchTreeNode, the SearchTreeNode which was passed to the function. This will be followed by the operator applied to get this new EndGameState, ***collect*** for example. The depth of the SearchTreeNode should also be specified. That is the depth of the parent SearchTreeNode+1. This followed by the path cost of the new SearchTreeNode, which is the same as the parent’s, **in addition** to the damage we calculated from the action performed due to applying the operator. If the SearchTreeNode is of heuristic not of type 1 or 2, the heuristic cost is set to 0. (The case of the heuristic type 1 or 2 will be discussed later in section 5). Finally, the heuristic type of the SearchTreeNode is added to the newly created SearchTreeNode, and its returned.

If the operator is *snap*, a check is made to make sure that all stones are collected and Iron Man is in Thanos cell with a damage less than 100. If that is the case, isAlive variable is set to false, and we generate an EndGameState string, and check for its repeatedly, as described above. If the EndGameState is not repeated, we create a SearchTreeNode for it as described above and return it. Otherwise, null is returned if Iron Man is not in Thanos cell because the action cannot be performed outside Thanos cell, or if damage >=100.

If the operator is *left*, a check is made to make sure that Iron Man is not outside the grid. Next, another check is made to check that if by moving left Iron Man is in Thanos cell, then the damage should be less than 100. Otherwise, a null value should be returned, as Iron Man cannot exist in a cell with Thanos and his damage is more than 100. In addition, he cannot be in Thanos cell if he hasn’t collect all the stones. So, a null value is also retuned in this case. If this three (3) checks pass, we check if that the cell Iron Man is trying to move to has a warrior using the helper method *checkWarrior*, if that is the case, Iron Man stays in place, and cannot move. Therefore, the state is repeated and null is returned. However, in case that check passes too, Iron man changes position normally. However, if by moving, Iron Man becomes adjacent to a Warrior or Thanos the damage variable holding the current path cost is incremented according to the enemy adjacent to it. The count of warriors adjacent to Iron Man is retrieved from the helper method *adjWarriors.* The damaged caused by adjacent warriors is their count. Likewise, *adjThanos* helper method is used to check is Thanos is adjacent to Iron Man. If that is the case, the damage is incremented by 5. The EndGameState string is then generated using generateEGS, and the repeated states are checked as described above. Also a new SearchTreeNode is created and returned as described above. The same logic applies for the *up, down, and right* operators.

However, if the operator is *kill,* the first thing we do is get the number of adjacent warriors, if they are above 0, then this action is applicable and we will **NOT** return null. Otherwise we will return null as the action is not applicable. If the action is applicable, we use the *killWarriors* helper method to remove the killed warriors from the array list they exist within. Next, increment the variable which holds the damage by 2\* the number of adjacent warriors. However, you also need to check if Thanos was adjacent to Iron Man while killing. If that is the case. Increment the damage by 5. Then we check for repeated states, and create a SearchTreeNode as discussed previously.

# Section 3: Main Functions

### Main.java

Within the Main class a method called *visualize* was created. This method is used to visualize the grid string passed to it, like the one in the project description. For example, "5,5;2,2;4,2;4,0,1,2,3,0,2,1,4,1,2,4;3,2,0,0,3,4,4,3,4,4". (Which is different than the grid String in the *EndGameState*. For example, "5,5;2,2;4,2;4,0,1,2,3,0,2,1,4,1,2,4;3,2,0,0,3,4,4,3,4,4;0;true" ).

It visualizes the grid through string manipulation and splitting on commas and semicolons, and placing the content in a 2d array, then looping through this 2D array and printing its content. This method is called inside *PathCostVisualize* method in EndGame to visualize the grid.

Inside the solve method, also string manipulation was made to place the string content in data structures and variables. Using the generateEGS method, a new EndGameState string was created. Therefore, we were able to create a new EndGame search problem with an initial state, and an array of Strings containing the operators which will be used. The GeneralSearch was then called on EndGame.

# Section 4: Search Strategies Implementation and Visualization

Inside the General Search we have the different search strategies, switch exists in a switch case. However, before that there are some variable which we set. First a Boolean isExpanded is set to false. This Boolean is used to check if a certain Node is expanded, if so it increments the number of expanded nodes by one. The number of expanded nodes are initially 0 and are saved in a variable called nodesNumber.

### Breadth-First Search

In a queue called *nodes*, the initial SearchTreeNode is enqueued. This SearchTreeNode was formed as described in *Section 2 – SearchProblem.java.* Then we iterate in a while loop which does not terminate unless a goal is found or the queue is empty. Inside the while loop, we check if the queue is empty or not. If it is not, we remove the first SearchTreeNode off the queue, and check if that is a goal SearchTreeNode, through the *GoalTest* method.

If it is the case that the SearchTreeNode removed from the queue is the goal, we call the PathCost method which returns a string of the sequence of operators used to reach the goal SearchTreeNode. However, there is an extra check. If a Boolean value of true was passed to the GeneralSearch, we call the PathCostVisualize function. This function is used to visualize or display the grids which were produced in the path from the root to the goal SearchTreeNode (Which is the parameter passed to the method PathCostVisualize). Finally, a string semicolon separated of the series of operators applied, the goal node path cost, and the number of nodes expanded is returned.

However, if it is not the case that the SearchTreeNode removed is the goal, we apply all the operators on the SearchTreeNode by calling the transition function. As we know, the transition function either returns a null or a SearchTreeNode. If a SearchTreeNode is returned isExpanded is set to true, and the resulting SearchTreeNode is enqueued at the end of the queue. After all operators are applied on the SearchTreeNode, we check if isExpanded is true. If that is the case, the number of SearchTreeNodes expanded, nodesNumber, is incremented by 1, and we go back to the while loop.

### Depth-First Search

In a stack called *stackNodes*, the initial SearchTreeNode is pushed. This SearchTreeNode was formed as described in *Section 2 – SearchProblem.java.* Then we iterate in a while loop which does not terminate unless a goal is found or the stack is empty. Inside the while loop, we check if the stack is empty or not. If it is not, we *pop* the first SearchTreeNode off the queue, and check if that is a goal SearchTreeNode, through the *GoalTest* method.

If it is indeed the case that the SearchTreeNode popped from the stack is the goal, we call the PathCost method which returns a string of the sequence of operators used to reach the goal SearchTreeNode. However, there is an extra check. If a Boolean value of true was passed to the GeneralSearch, we call the PathCostVisualize function. This function is used to visualize or display the grids which were produced in the path from the root to the goal SearchTreeNode (Which is the parameter passed to the method PathCostVisualize). Finally, a string semicolon separated of the series of operators applied, the goal node path cost, and the number of nodes expanded is returned.

However, if it is not the case that the SearchTreeNode popped is the goal, we apply all the operators on the SearchTreeNode by calling the transition function. As we know, the transition function either returns a null or a SearchTreeNode. If a SearchTreeNode is returned isExpanded is set to true, and the resulting SearchTreeNode is pushed on top of the stack. After all operators are applied on the SearchTreeNode, we check if isExpanded is true. If that is the case, the number of SearchTreeNodes expanded, nodesNumber, is incremented by 1, and we go back to the while loop.

### Uniform-Cost Search

We first start by overriding the priority queue comparator. To make sure the SearchTreeNodes with lower path cost are added to the beginning of the queue. (Sorted from least to greatest path cost). Therefore, in a priority queue called *pq*, the initial SearchTreeNode is enqueued. This SearchTreeNode was formed as described in *Section 2 – SearchProblem.java.* Then we iterate in a while loop which does not terminate unless a goal is found or the priority queue is empty. Inside the while loop, we check if the priority queue is empty or not. If it is not, we *remove* the first SearchTreeNode off the priority queue, and check if that is a goal SearchTreeNode, through the *GoalTest* method.

If it is indeed the case that the SearchTreeNode removed from the priority queue is the goal, we call the PathCost method which returns a string of the sequence of operators used to reach the goal SearchTreeNode. However, there is an extra check. If a Boolean value of true was passed to the GeneralSearch, we call the PathCostVisualize function. This function is used to visualize or display the grids which were produced in the path from the root to the goal SearchTreeNode (Which is the parameter passed to the method PathCostVisualize). Finally, a string semicolon separated of the series of operators applied, the goal node path cost, and the number of nodes expanded is returned.

However, if it is not the case that the SearchTreeNode removed is the goal, we apply all the operators on the SearchTreeNode by calling the transition function. As we know, the transition function either returns a null or a SearchTreeNode. If a SearchTreeNode is returned, isExpanded is set to true, and the resulting SearchTreeNode is inserted in the priority queue according to its cost. After all operators are applied on the SearchTreeNode, we check if isExpanded is true. If that is the case, the number of SearchTreeNodes expanded, nodesNumber, is incremented by 1, and we go back to the while loop.

### Iterative-Deeping Search

In iterative-deeping the depth is incremented gradually until a goal is found. Therefore, there is a for loop which loops from 0 till maxDepth. The maxDepth variable starts from 0. Inside the for loop, we have a stack called *stackNodes*, the initial SearchTreeNode is pushed inside it. This SearchTreeNode was formed as described in *Section 2 – SearchProblem.java.* Then we iterate in a while loop which does not terminate unless a goal is found or the stack is empty.

If the stack is empty, we increment the maxDepth by 1 and go back to the for loop, clear the *repeatedStates* hashset, and then go through the while loop once again.

Inside the while loop, we check if the stack is empty or not. If it is not, we *pop* the first SearchTreeNode off the queue, and check if that is a goal SearchTreeNode, through the *GoalTest* method.

If it is indeed the case that the SearchTreeNode popped from the stack is the goal, we call the PathCost method which returns a string of the sequence of operators used to reach the goal SearchTreeNode. However, there is an extra check. If a Boolean value of true was passed to the GeneralSearch, we call the PathCostVisualize function. This function is used to visualize or display the grids which were produced in the path from the root to the goal SearchTreeNode (Which is the parameter passed to the method PathCostVisualize). Finally, a string semicolon separated of the series of operators applied, the goal node path cost, and the number of nodes expanded is returned.

However, if it is not the case that the SearchTreeNode popped is the goal, we apply all the operators on the SearchTreeNode by calling the transition function. As we know, the transition function either returns a null or a SearchTreeNode. If a SearchTreeNode is returned, **we first have to check its depth**. If it is greater than the maxDepth, we do not push it on top of the stack. Otherwise, if its depth is less than or equal to the maxDepth the SearchTreeNode is pushed on top of the stack and isExpanded is set to true, and the resulting SearchTreeNode is pushed on top of the stack. After all operators are applied on the SearchTreeNode, we check if isExpanded is true. If that is the case, the number of SearchTreeNodes expanded, nodesNumber, is incremented by 1, and we go back to the while loop.

### Greedy Search

We first start by overriding the priority queue comparator. To make sure the SearchTreeNodes with **lower heuristic cost** are added to the beginning of the queue. (Sorted from least to greatest heuristic cost). Therefore, in a priority queue called *gpq*, the initial SearchTreeNode is enqueued. This SearchTreeNode was formed as described in *Section 2 – SearchProblem.java.* Then we iterate in a while loop which does not terminate unless a goal is found or the priority queue is empty. Inside the while loop, we check if the priority queue is empty or not. If it is not, we *remove* the first SearchTreeNode off the priority queue, and check if that is a goal SearchTreeNode, through the *GoalTest* method.

If it is indeed the case that the SearchTreeNode removed from the priority queue is the goal, we call the PathCost method which returns a string of the sequence of operators used to reach the goal SearchTreeNode. However, there is an extra check. If a Boolean value of true was passed to the GeneralSearch, we call the PathCostVisualize function. This function is used to visualize or display the grids which were produced in the path from the root to the goal SearchTreeNode (Which is the parameter passed to the method PathCostVisualize). Finally, a string semicolon separated of the series of operators applied, the goal node path cost, and the number of nodes expanded is returned.

However, if it is not the case that the SearchTreeNode removed is the goal, we apply all the operators on the SearchTreeNode by calling the transition function. As we know, the transition function either returns a null or a SearchTreeNode. If a SearchTreeNode is returned, isExpanded is set to true, and the resulting SearchTreeNode is inserted in the priority queue according to its heuristic cost. After all operators are applied on the SearchTreeNode, we check if isExpanded is true. If that is the case, the number of SearchTreeNodes expanded, nodesNumber, is incremented by 1, and we go back to the while loop.

### A\* Search

We first start by overriding the priority queue comparator. To make sure the SearchTreeNodes with **lower (heuristic cost+ path cost)** are added to the beginning of the queue. (Sorted from least to greatest *heuristic cost + path cost*). Therefore, in a priority queue called Astarpq, the initial SearchTreeNode is enqueued. This SearchTreeNode was formed as described in *Section 2 – SearchProblem.java.* Then we iterate in a while loop which does not terminate unless a goal is found or the priority queue is empty. Inside the while loop, we check if the priority queue is empty or not. If it is not, we *remove* the first SearchTreeNode off the priority queue, and check if that is a goal SearchTreeNode, through the *GoalTest* method.

If it is indeed the case that the SearchTreeNode removed from the priority queue is the goal, we call the PathCost method which returns a string of the sequence of operators used to reach the goal SearchTreeNode. However, there is an extra check. If a Boolean value of true was passed to the GeneralSearch, we call the PathCostVisualize function. This function is used to visualize or display the grids which were produced in the path from the root to the goal SearchTreeNode (Which is the parameter passed to the method PathCostVisualize). Finally, a string semicolon separated of the series of operators applied, the goal node path cost, and the number of nodes expanded is returned.

However, if it is not the case that the SearchTreeNode removed is the goal, we apply all the operators on the SearchTreeNode by calling the transition function. As we know, the transition function either returns a null or a SearchTreeNode. If a SearchTreeNode is returned, isExpanded is set to true, and the resulting SearchTreeNode is inserted in the priority queue according to its (path cost + heuristic cost). After all operators are applied on the SearchTreeNode, we check if isExpanded is true. If that is the case, the number of SearchTreeNodes expanded, nodesNumber, is incremented by 1, and we go back to the while loop.

For each of the above search strategies, once the goal is found, the *repeatedStates* hash set is cleared.

# Section 5: Discussion of the Heuristic Functions

The two heuristic functions that we used in our project for the Greedy and the A\* search are:

### Heuristic One:

The heuristic value starts with the value of the number of stones in the game multiplied by two and an addition of 4, therefore if at the current moment we have 6 stones left the heuristic cost will start at 16, but if we have 4 stones left at the current moment then the heuristic cost will start at 12. Then we add a check which gets the number of adjacent warriors to Iron Man, for each adjacent warrior next to Iron Man we add 1 point to the heuristic cost, therefore if the heuristic cost started at 12 and 2 warriors were adjacent to Iron Man then the heuristic cost will be 14. Then, we check if Iron Man is adjacent to Thanos, if the check is true, then we add 1 point to the heuristic cost, therefore if the heuristic cost is 14 it will become 15 if Iron Man is adjacent to Thanos. Lastly, we check if Iron Man is collecting a stone which is in the same position on the grid as Iron Man, if so, we deduct 2 points from the heuristic cost, if the heuristic cost is 15 it will equal to 13.

In the case of all the stones being collected the heuristic will start at 4 which will always be an underestimate since the step before snapping requires us to be adjacent to Thanos which costs 5 points, therefore 4<5 leading to an underestimate.

### Heuristic Two:

The heuristic value starts with the value of the number of stones in the game multiplied by two and an addition of 4, therefore if at the current moment we have 6 stones left the heuristic cost will start at 16, but if we have 4 stones left at the current moment then the heuristic cost will start at 12. Then, we check if Iron Man is collecting a stone which is in the same position on the grid as Iron Man, if so, we deduct 2 points from the heuristic cost, if the heuristic cost is 12 it will equal to 10.

In the case of all the stones being collected the heuristic will start at 4 which will always be an underestimate since the step before snapping requires us to be adjacent to Thanos which costs 5 points, therefore 4<5 leading to an underestimate.

Our A\* search uses two different heuristic functions that are admissible as they underestimate the path cost to the goal node because the heuristic cost in both heuristic approaches are always strongly dependent on the number of stones collected. The main aim of the game is to finish collecting all stones and then to get Iron Man to Thanos’s cell to snap his fingers and end the game. Therefore by making the number of stones a strong factor in both of the heuristic approaches will always assure us that as the stones are decreasing which also means that Iron Man is closer to winning our heuristic cost which is calculating the cost to the goal is decreasing as well which means that it is expecting that the current node is getting closer to the goal node while still underestimating since being adjacent to Thanos before the last step should cost 5 therefore the 4 will be an underestimate. Therefore, both of our heuristic functions are admissible.

# Section 6: - Demonstration of Two-Functional Examples

The following examples are runs of the solve method for every type of search that we implemented. We added two different types of grids for each of the search examples:

### Breadth-First Search

*solve*(grid5,"BF",**false**);

Output:

right,right,collect,left,left,up,collect,left,down,collect,left,down,collect,down,collect,right,collect,right,snap;33;11680

*solve*(grid6,"BF",**true**);

Output:

- T - - - -

w s s w w -

- s w - - -

s - - w - -

- - - s - s

- - - I - -

- T - - - -

w s s w w -

- s w - - -

s - - w - -

- - - s - s

- - - - I -

- T - - - -

w s s w w -

- s w - - -

s - - w - -

- - - s - s

- - - - - I

- T - - - -

w s s w w -

- s w - - -

s - - w - -

- - - s - Is

- - - - - -

- T - - - -

w s s w w -

- s w - - -

s - - w - -

- - - s - I

- - - - - -

- T - - - -

w s s w w -

- s w - - -

s - - w - -

- - - s I -

- - - - - -

- T - - - -

w s s w w -

- s w - - -

s - - w - -

- - - Is - -

- - - - - -

- T - - - -

w s s w w -

- s w - - -

s - - w - -

- - - I - -

- - - - - -

- T - - - -

w s s w w -

- s w - - -

s - - w - -

- - I - - -

- - - - - -

- T - - - -

w s s w w -

- s w - - -

s - - w - -

- I - - - -

- - - - - -

- T - - - -

w s s w w -

- s w - - -

s - - w - -

I - - - - -

- - - - - -

- T - - - -

w s s w w -

- s w - - -

Is - - w - -

- - - - - -

- - - - - -

- T - - - -

w s s w w -

- s w - - -

I - - w - -

- - - - - -

- - - - - -

- T - - - -

w s s w w -

- s w - - -

- I - w - -

- - - - - -

- - - - - -

- T - - - -

w s s w w -

- Is w - - -

- - - w - -

- - - - - -

- - - - - -

- T - - - -

w s s w w -

- I w - - -

- - - w - -

- - - - - -

- - - - - -

- T - - - -

w Is s w w -

- - w - - -

- - - w - -

- - - - - -

- - - - - -

- T - - - -

w I s w w -

- - w - - -

- - - w - -

- - - - - -

- - - - - -

- T - - - -

w - Is w w -

- - w - - -

- - - w - -

- - - - - -

- - - - - -

- T - - - -

w - I w w -

- - w - - -

- - - w - -

- - - - - -

- - - - - -

- T - - - -

w I - w w -

- - w - - -

- - - w - -

- - - - - -

- - - - - -

- IT - - - -

w - - w w -

- - w - - -

- - - w - -

- - - - - -

- - - - - -

- IT - - - -

w - - w w -

- - w - - -

- - - w - -

- - - - - -

- - - - - -

right,right,up,collect,left,left,collect,left,left,left,up,collect,right,up,collect,up,collect,right,collect,left,up,snap;44;35057

### Depth-First Search

*solve*(grid5,"DF",**false**);

Output:

kill,down,right,kill,down,kill,up,up,up,up,left,left,kill,down,down,down,down,left,collect,up,up,up,up,right,right,down,down,down,right,right,up,collect,down,left,left,up,up,up,left,left,down,down,down,down,right,collect,up,up,up,right,collect,down,down,left,left,collect,up,right,collect,down,down,right,snap;82;134

*solve*(grid6,"DF",**false**);

Output:

up,kill,down,right,right,up,up,up,up,kill,down,down,down,down,left,left,up,up,up,kill,down,down,down,right,right,up,up,up,up,up,left,left,left,down,left,kill,down,down,down,down,right,right,up,up,up,up,up,right,right,down,down,down,down,collect,down,left,left,up,up,up,up,left,left,down,down,left,collect,down,down,right,right,up,up,up,up,left,collect,down,down,down,down,right,right,up,collect,down,right,right,up,up,up,up,up,left,left,down,down,left,left,collect,down,down,down,right,right,up,up,up,up,left,collect,down,down,down,down,left,left,up,up,up,up,up,right,snap;68;204

### Uniform-Cost Search

*solve*(grid5,"UC",**false**);

Output:

right,right,collect,up,left,left,collect,left,down,collect,left,down,collect,down,collect,right,collect,right,snap;32;14627

*solve*(grid6,"UC",**false**);

Output:

left,up,left,up,left,collect,down,down,right,right,up,right,collect,left,left,up,up,kill,right,up,collect,down,left,down,down,right,down,right,right,right,up,collect,down,left,left,left,up,left,up,up,collect,up,collect,up,snap;37;38607

### Iterative-Deeping Search

*solve*(grid5,"ID",**false**);

Output:

kill,down,left,left,collect,up,up,right,right,collect,down,right,right,kill,collect,down,left,left,up,left,collect,down,down,left,collect,right,collect,right,snap;52;74753

*solve*(grid6,"ID",**false**);

Output:

up,right,right,collect,down,left,left,up,collect,down,right,right,up,up,up,up,up,left,left,left,down,collect,left,kill,down,collect,down,left,collect,up,up,right,collect,up,snap;58;193933

### Greedy Search 1

*solve*(grid5,"GR1",**false**);

Output:

left,collect,left,down,collect,down,collect,up,up,right,up,right,collect,right,down,right,collect,left,up,left,left,up,kill,left,down,down,down,down,right,collect,left,up,up,right,right,kill,right,right,kill,left,left,left,down,down,right,snap;45;136

*solve*(grid6,"GR1",**false**);

Output:

right,up,right,collect,left,down,left,left,left,left,up,up,collect,right,up,collect,kill,down,down,right,right,collect,kill,left,up,up,right,kill,left,up,collect,down,left,left,kill,right,up,collect,up,snap;42;204

### Greedy Search 2

*solve*(grid5,"GR2",**false**);

Output:

left,collect,right,kill,up,collect,left,down,left,down,collect,down,collect,right,collect,up,right,right,up,right,collect,kill,left,down,left,down,snap;49;73

*solve*(grid6,"GR2",**false**);

Output:

up,collect,kill,down,right,right,up,collect,left,left,up,up,kill,up,left,collect,left,collect,down,collect,left,down,collect,right,right,up,up,up,left,snap;47;113

### A\* 1

*solve*(grid5,"AS1",**false**);

Output:

right,right,collect,left,up,left,collect,left,down,collect,left,down,collect,down,collect,right,collect,right,snap;32;12727

*solve*(grid6,"AS1",**false**);

Output:

up,collect,left,left,up,up,kill,right,up,collect,down,left,down,down,right,down,right,right,right,up,collect,down,left,left,left,left,left,up,up,right,up,collect,down,left,collect,up,kill,up,right,collect,up,snap;38;34862

### A\* 2

*solve*(grid5,"AS2",**false**);

Output:

right,right,collect,left,up,left,collect,left,down,collect,left,down,collect,down,collect,right,collect,right,snap;32;11780

*solve*(grid6,"AS2",**false**);

Output:

left,up,left,up,up,kill,right,up,collect,down,left,down,left,collect,right,down,right,down,right,right,right,up,collect,left,left,collect,left,left,up,up,collect,up,collect,up,snap;37;28890

# Section 7: - Performance Comparison

### Breadth-First Search

#### Complete

Generally, breadth-first search is complete because eventually at some level l, the goal is going to be found if it exists. Here in this EndGame problem, since we are eliminating repeated states, then for sure the breadth-first search is complete.

#### Optimal

The breadth-first search can be considered optimal when all the nodes have the same path cost. Therefore, choosing the shallowest one would be considered optimal. Since the path cost was not taken into consideration here the breadth-first search is somehow optimal, as it returns the first goal it finds.

#### Expansion and Cost in 5x5

In the 5x5 grid the result of the damage/path cost was 33. Besides, the nodes expanded were 11680

#### Expansion and Cost in 6x6

In the 6x6 grid the result of the damage/path cost was 44. Besides, the nodes expanded were 35057

### Depth-First Search

#### Complete

Usually the Depth-First Search is not complete. However, in the case of this search problem, the repeated states were removed, and only the unique visited states remained. This guaranteed us to reach a solution if one exists.

#### Optimal

Even though Depth-First Search has the best space complexity but is not optimal. That is because you keep getting deeper and deeper in the graph and exploring states, and if you do not find a goal, you have to backtrack. This means even if the solution was the first node in branch 2. You would ignore it just to go deeper in branch 1 searching for the solution. Therefore, it is not optimal.

#### Expansion and Cost in 5x5

In the 5x5 grid the result of the damage/path cost was 82. Besides, the nodes expanded were 134. As you can notice, the cost for finding a goal in DFS is much higher than BFS (Beyond the double in this case/grid).

#### Expansion and Cost in 6x6

In the 6x6 grid the result of the damage/path cost was 68. Besides, the nodes expanded were 204.

### Uniform-Cost Search

#### Complete

Uniform cost expands the nodes with the less cost first. Again, since we handle repeated states then definitely we are to find an answer if one exists. Therefore, UCS is complete.

#### Optimal

Since the cost of the parent node is never greater than the cost of the child (It is either equal or less). Then Uniform-cost is optimal. Also, since the nodes are expanded according to their path cost, then we are going to get an optimal solution.

#### Expansion and Cost in 5x5

In the 5x5 grid the result of the damage/path cost was 32. Besides, the nodes expanded were 14627. The path cost to the goal is less than that of BFS and DFS in this grid.

#### Expansion and Cost in 6x6

In the 6x6 grid the result of the damage/path cost was 37. Besides, the nodes expanded were 38607. Again the damage or the path cost is less than that of BFS and DFS with the same grid.

### Iterative-Deeping Search

#### Complete

IDS combines DFS and BFS to produce a complete optimal search. Therefore, at a certain depth *d* we are to find a solution if one exists. Also, since the repeated states are eliminated, we are guaranteed to find a solution if one exists.

#### Optimal

IDS is optimal given a non-decreasing function. In the EndGame problem case, the successors always have a higher path cost compared to their parents. Therefore, the path cost function is non-decreasing, and hence optimal. Since path cost either increases or stays the same as we move down the search space.

#### Expansion and Cost in 5x5

In the 5x5 grid the result of the damage/path cost was 52. Besides, the nodes expanded were 74753.

#### Expansion and Cost in 6x6

In the 6x6 grid the result of the damage/path cost was 58. Besides, the nodes expanded were 193933.

### Greedy Search

#### Complete

The greedy search is complete since the nodes with the lower costs (closer to the goal) are expanded first. Besides, we traverse level l at a time. Therefore, we are guaranteed to find a solution at some level *l*.

#### Optimal

When adding successors by expanding the node, the successors with the heuristic cost will be expanded first. Therefore, at each step, the optimal local solution is found. Therefore, overall greedy search is optimal.

#### Expansion and Cost in 5x5

In the 5x5 grid the result of the damage/path cost was GR1: 45 and GR2:49. Besides, the nodes expanded were GR1:136 and GR2: 73.

#### Expansion and Cost in 6x6

In the 6x6 grid the result of the damage/path cost was GR1: 42 and GR2:47. Besides, the nodes expanded were GR1:204 and GR2: 113. The number of expanded nodes for GR1 is the same of that of DFS for the same grid. However, the path cost for the greedy is much less showing that it is a more optimal solution compared to DFS.

### A\* Search

#### Complete

A\* is admissible. It never overestimates the path cost. Since we are sorting depend on the path cost+ heuristic (combine greedy and UCS). Then we are always getting the best solution at the front. Therefore, if there exists a solution, we will find it at the front.

#### Optimal

Again, A\* is admissible. It never overestimates the path cost. Since we are sorting depend on the path cost+ heuristic. Then we are always getting the best solution at the front. Therefore, if there exists a solution, we will find the best one at the front.

#### Expansion and Cost in 5x5

In the 5x5 grid the result of the damage/path cost was AS1: 32 and AS2:32. Both heuristics found the most optimal path cost to the goal. Besides, the nodes expanded were AS1:12727 and AS2: 11780.

#### Expansion and Cost in 6x6

In the 6x6 grid the result of the damage/path cost was AS1: 38 and AS2:37. This shows that the second heuristic found a more optimal path cost. Besides, the expanded nodes for AS1:34862 and AS2: 28890.

### Analysis

Since A\* combines both greedy and UCS to find the most optimal solution, in terms of *optimality* A\* is the most optimal solution with the **least path cost**. Even though, uniform cost search produced the same path cost. However, the expanded nodes for the A\* algorithm were less compared to UCS given equivalent grids UCS 5X5 expanded nodes: 14627, and for the 6x6: 38607.

In terms of Nodes expansion, the first heuristic used in the greedy expanded the fewest nodes between the search strategies. For the 5x5 grid: 136 nodes, as for the 6x6 grid: 204 nodes. The same expansion nodes were produced from the DFS for the 5x5 and 6x6 respectively.

It is worth noting that the IDS has the biggest number of expanded nodes, as it keeps on repeating every step it took from the beginning incrementally successive times. This makes it a bad decision to use.

# Section 8: Errors

87/88 tests passed successfully. However, **one** tested resulted in an error. That was the 13x13 IDS grid. It might have failed due to the low RAM of the computer. Also, a different sequence of operators might have made it pass the test. That is because changing the order of operators changes the chosen node to be expanded first in cases like DFS, and IDS.