Search Agent Report

Hesham Morgan; Tarek Badreldin; Omar Osama
Team 2
German University in Cairo - Introduction to Artificial Intelligence (CSEN 901)

Abstract

The problem solving agent is one of the goal-based kinds of Artificial Intelligence agents. In this report the uninformed and informed search strategies discussed in the lecture are examined. The search strategies are implemented for a simple problem to observe the differences between them. The problem restricts one of the aspects of the normal search approach. A comparison of the implemented search strategy approach is discussed, and the differences are reflected.

1 Introduction

Description The problem comprises of some 2D grid where some collectables are placed in cells in the grid, A cell is a position in the grid. The agent starts in a given cell initially and tries to reach some goal. Moreover, the agent can move in four directions relative to its position, each move transitions the agent from the cell it was in to an adjacent one. An adjacent cell is a cell that is directly next to the cell in either of the following directions, $directions = \{up, right, down, left\}$. The agent can move in any of the allowed directions as the action of some state, a state is the description of the world at a specific time step. A time step is an indicator which refers to some state of the world based on some action done in some previous state. The agent has value to indicate its damage sustained so far from the environment in the world. An action is the operation done by the agent to change some aspect of the world as defined by the effects of the action if and only if the preconditions of the actions are satisfied in the give state. In the environment there are two types of enemies, the first being the warriors, and the second is the villain(main enemy) of the problem stated. The warriors can inflict *one* damage unit to the agent if the agent was in an adjacent cell, and if the agent tried to kill them, the warriors would fight back which would result with gaining *two* damage units for the agent. Furthermore, when the agent executes the kill action, all the warriors in all of the adjacent cells are eliminated. For the other type of enemies, the villain inflects *five* damage units if the agent was in adjacent cell, and the agent cannot kill the villain in the normal circumstances. Moreover, the collectables placed in the world inflict each three units of damage if the agent collected them. The goal of the agent is to collect all the collectables while minimizing on the damage sustained, and eliminating the villain of the problem. All of the objects in the environment are stationary and cannot move from their initial position. The villain can be eliminated if all the stones are collected and the agent is in the villains cell with damage units less than *one hundred*. Therefore the actions the agent can perform are $actions = \{up, right, down, left, collect, kill, end\}$, where end is the action discussed to eliminate the villain of the problem.

Specification The problem is themed after the Marvel Cinematic Universe latest movies, Infinity Wars & End Game. Therefore more specifications to the described problem will be explained for the context which would be discussed in later sections. After the event of Infinity Wars, Iron Man, the protagonist of the story, goes to freeze Thanos, the villain of the story who eliminated half of the civilization on earth. Iron Man then goes on a journey to retrieves all of the infinity stones scattered in the universe to be able to defeat Thanos. Thanos' followers, warriors, try to stop Iron Man from collecting all of the stones, but Iron Man would prove victorious by either evading them, sustaining the damage, or by eliminating them. There are *six stones* scattered throughout the universe, which Iron Man must collect. After Iron Man attain the *six stones*, Iron Man goes to Thanos, unfreezes him, and eliminate him by snapping his fingers while wearing the *Infinity gauntlet*, which attains him a wish while saying ***"I am Iron Man"*** and sacrificing himself to save humanity.

Motivation A search agent searches for the solution to the problem by applying all possible actions at each state till reaching a goal. A solution is the sequence of actions that lead to a goal state for the given problem. As the problem stated in the description can be solved through *brute force* of the operators at each state to deduce some solution depending on the approach of traversal chosen to be applied. It was thought as a practical approach to understand the various search techniques introduced in the lectures. There are *six approaches* introduced in the course, where each will be discussed in it corresponding section.

2 Background

This section discusses the background material required for the understanding for this report. This includes definition for search, and the search strategies discussed in the lectures.

Search Search is the procedure that takes a problem as an input and returns a solution to that problem [1], which can be categorized into two types. The first being Uniformed search, where the procedure have no additional information about the world, and traverse the search tree in some systematic manner specified by the search strategy chosen. On the other hand, informed search have some kind of heuristic that tries to lead the traversal to some path that have an optimal or sub optimal solution. A heuristic is a function that tries to predict the path cost at each state to the goal. Uninformed search can be though as blind search, where the agent does not know if the path it is on would lead to a goal or not, while informed search can be thought as an educated search about the world, where it tries to choose the best path at each step. Initially, the search procedure enqueues the initial state of the problem, dequeues it to expand it, and enqueues the nodes that are transitioned to from the expansion procedure.

Listing 1: General Search Procedure [1]

```
function GENERAL−SEARCH(problem, QING−FUN)
returns a solution, or failure
nodes ← MAKE−Q(MAKE−NODE(INIT−STATE(problem)))
loop do
if nodes is empty
then return failure
node ← REMOVE−FRONT(nodes)
if GOAL−TEST(problem)(STATE(node))
then return node
nodes ← QING−FUN(nodes, EXPAND(node, OPER(problem)))
end
```

Uninformed search discussed in this report are *Breadth First Search*, *Depth First Search*, *Uniform Cost Search*, and *Iterative Deepening Search*. Moreover, informed search discussed in this report are *Greedy Search*, and *A* Search*.

Breadth First Search *Breadth First Search* traverses the search tree level by level, where it visits all the nodes in the frontier from the same depth level before proceeding to the next depth level. The time complexity of *Breadth First Search* is $O(b^d)$ where b is the number of branches at each node and d is the depth of the shallowest goal state. This is deduced from the cost of *Breadth First Search* is equal to $O(\sum_{i=0}^{d} b^i) = O(b^0 + b^1 + b^2 + ... + b^{d-1} + b^d) = O(b^d)$. For the space complexity for *Breadth First Search*, at each step, all of the frontier must be stored in the memory, and as *Breadth First Search* expands all the nodes in the same level before proceeding to the next, then all the nodes that are in the given depth level must be stored in the memory; therefore the space complexity cost of the *Breadth First Search* is equal to $O(b^d)$. *Breadth First Search* is complete, as the search must reach a goal at the shallowest depth d. However, *Breadth First Search* is not optimal, as the search outputs the goal at the shallowest depth, which does not correlate to the cost of the path. *Breadth First Search* can be optimal though if the cost for all operators is equal to 1.

Depth First Search *Depth First Search* traverses each branch till the leaf before backtracking to unvisited branch in the search tree. The time complexity of *Depth First Search* is equal to $O(b^d)$, as in worst case the entire tree must be traversed. Intuitively, *Depth First Search* may never halt if it was expanding some repeated state in the world. The space complexity of *Depth First Search* is equal to $O(bd)$, as *Depth First Search* traverses each branch before moving to another, therefore only the nodes in the path to the given node being expanded are required to be stored in the memory. The advantage of the space complexity of this strategy is very apparent. *Depth First Search* is not complete, as it does not guaranteed to reach a goal if there was an infinite path in the search tree. *Depth First Search* is not optimal, as the search reaches the goal at the shallowest depth at the most left section of the search tree. *Depth First Search* can be optimal if the cost of the path is distributed so that the cost increase as the branch is at the right side.

Iterative Deepening Search Therefore *Iterative Deepening Search* was deduced, as it combines the completeness of *Breadth First Search* and the space complexity of *Depth First Search*. The search strategy traverses in the same manner as *Depth First Search*, but has a cutoff depth. The strategy halts if the cutoff depth was reached, increments the cutoff depth if a goal state was not expanded, then start searching in the search tree from the root. Hence the time complexity is $O(b^d)$, where the worst case is to traverse the entire search tree, and the space complexity is $O(bd)$. However even though this strategy seems to be a good solution, it suffers from requirement of expensive computational time. The actual time complexity of this strategy is $O(\sum_{i=0}^{d-1}(d-i) * \sum_{j=0}^{i} b^j) = O((d*[b^0]) + (d-1*[b^0+b^1]) + ... + (2*[b^0+b^1+b^2+...+b^{d-2}+b^{d-1}]) + (1*[b^0+b^1+b^2+...+b^{d-1}+b^d])) = O(b^d)$, where d is the depth of the shallowest goal, and b is the number of branches at each node. Clearly it can be noticed that the actual cost of *Iterative Deepening Search* is actually very expensive although the *big O notation* would not show that. *Iterative Deepening Search* is complete as it is guaranteed to reach a goal state as the traversal checks level by level the nodes in a *Depth First Search* manner. However *Iterative Deepening Search* is not optimal for the same reasoning as *Breadth First Search*, and *Depth First Search*.

Uniform Cost Search *Uniform Cost Search* traverses the search tree depending on the value of the path cost of the node. The time complexity of this strategy is equal to $O(b^d)$, as in worst case the entire tree must be traversed. Similar to *Breadth First Search*, *Uniform Cost Search* must store all the frontier in the memory; therefore the space complexity cost of the *Uniform Cost Search* is equal to $O(b^d)$. The sharp reader would notice that *Uniform Cost Search* is optimal and complete in the case that all the costs are non negative numbers and greater than zero, as the strategy expands the nodes with the least cost. For example, a search tree that has two paths from the root, where the first path has non ending path of negative costs and does not lead to a goal states, while second path has the goal with some positive cost. Expanding the cost of the nodes in the first path would always result with some negative number, while the second path from the root node has a positive cost. In that manner the strategy will always expand the nodes on the first path, which is infinite, before trying to expand the node on the second path. Therefore *Uniform Cost Search* may never halt. *Uniform Cost Search* is complete, as the search traverses the tree depending on the path cost. *Uniform Cost Search*is optimal if the cost has no negative values, as the nodes are ordered in the queue depending on the value of the path cost.

Greedy Search *Greedy Search* traverses the search tree through the value of some heuristic function. A heuristic function predicts the cost from a given node to the goal. *Greedy Search* considers the value of the heuristic without considering the actual cost of the node. Therefore *Greedy Search* does not ensure reaching the optimal solution. As greedy search chooses the better cost of heuristic function at each step, but that would not mean that it is the lowest cost for the entire path. *Greedy Search* is not complete, as it is not guaranteed to reach a goal if there was an infinite path which uses some operator that is favorable for the heuristic function. *Greedy Search* is not optimal, as the heuristic value does not guaranteed the actual cost to the goal state, it may overestimate or underestimate.

A* Search *A* Search* traverses the tree depending on the cost of the path of a given node and the cost of the heuristic function. The heuristic cost must be admissible, where the cost is underestimate of the actual cost from a given node to the goal to ensure reaching the optimal goal. Furthermore, the function of the *A* Search* is defined as $\mathcal{F}(n) = \mathcal{G}(n) + \mathcal{H}(n)$, where $\mathcal{G}(n)$ is the path cost from the root to some node n in the search tree, and $\mathcal{H}(n)$ is the predicted cost from some node n in the search tree which must be an less than or equal the actual cost from the given node n to the goal. *A* Search* is complete, as it is guaranteed to find solution as the search traverses through the order of $\mathcal{F}(n)$. *A* Search* is optimal as discussed in the proof that was covered in the lecture.

3 Methodology

In this section the theoretical approach for the search agent for the problem stated is presented. The aim of the agent is to search for the goal state in the search tree. There are b^i possible actions that can be executed at each depth, where b is the number of branches at each state, and i is the depth of the state. Furthermore, there are at most *five* branches that can be expanded at each state. In the worst situation at a given state, the agent traverses the entire grid with accomplishing either a sub goal or a goal to the problem. Therefore the problem is expensive, and requires many optimizations.

Problem A generic search problem is the description of some problem that can be generated for which the agent seeks the solution through expanding all possible action at each state in a given search tree.

Definition. $Problem = \{N, SS, O\}$ is the representation of a search problem where,

- *N* is the initial node of the problem, where the world is described,
- *SS* is the state space, where the state space is a set of the visited states in a search tree,
- *O* is the set of operators that each state uses to transition to the successor state.

For the problem stated in here, some additional arguments are added to the search problem definition. In the problem stated, all of the environment is stationary, do not move from their position, therefore storing such information in each state would be redundant. Hence the positions of the objects in the environment will be stored in the problem itself, which would save memory as such information would be only stored once.

Definition. $EndGame = \{N, SS, O, C\}$ is the representation of the stated problem where it is a subclass to the generic search problem and *C* is the set of coordinates for each objects in the environment excluding the agent, where each coordinate consists of a pair of numbers representing the position of the object in the grid.

State A state is the definition of the world at some given time step. A state describes the orientation of all objects in the world, the state of each object, and the state of the agent in the given time step. In the stated problem, the agent is the only object which can change position in the grid. On the other hand the other objects in the environment are either in the state or do not exist depending on the consequences of the previous sequence of actions in some branch in the search tree.

Definition. $EndGame_{State} = \{I, StS, WS\}$ is the representation of the state in the stated problem where,

- *I* is the position of Iron Man in the grid, where of an object is represented as a pair that represents the coordinates in the x and y axes,
- *StS* is the set of states of each stone in the world, where $st \in StS$, $st \in \{\top, \bot\}$, and initially all the stones' status are set to true,
- *WS* is the set of states of each warrior in the world, where $w \in WS$, $w \in \{\top, \bot\}$, and initially all the warriors' status are set to true.

Operators An operator is the action allowed at some given state to transition to its successor state, a successor state is a state that is the consequence of applying some action at a given state in a give time step. An operator is applicable for some agent to execute if and only if its preconditions are satisfiable in the state of the world. An action's preconditions are satisfiable if all of the conditions stated are satisfied in the world at that given time step. In the stated problem, the applicable set of operators are defined as follows.

Definition. $EndGame_{Operators} = \{up, right, down, left, collect, kill, snap\}$ is the representations of the operators applicable for the agent to execute at a given state if the preconditions were satisfiable where,

- Operator *up* :
  - Preconditions : Iron Man is not at the upper border of the grid and is not trying to go out of bounds.
  - Effect : Iron Man moves to the cell directly above.
- Operator *right* :
  - Preconditions : Iron Man is not at the right border of the grid and is not trying to go out of bounds.
  - Effect : Iron Man moves to the cell directly to the right.
- Operator *down* :
  - Preconditions : Iron Man is not at the bottom border of the grid and is not trying to go out of bounds.
  - Effect : Iron Man moves to the cell directly bellow.
- Operator *left* :
  - Preconditions : Iron Man is not at the left border of the grid and is not trying to go out of bounds.

  - Effect : Iron Man moves to the cell directly to the left.

- Operator *collect* :
  - Preconditions : Iron Man must be in a cell that has a stone in it.
  - Effect : Iron Man collects the stone, sustains three units of damage, and the stone status is set to $\perp$.

- Operator *kill* :
  - Preconditions : Iron Man must be in a cell that is adjacent to one or more warrior(s).
  - Effect : Iron Man eliminates all the warriors in the adjacent cells, Iron Man sustains two units of damage for each warrior killed, and the adjacent warriors are eliminated get their status to be set to $\perp$.

- Operator *snap* :
  - Preconditions : Iron Man must have collected all the six stones, and he should be in the cell Thanos is in.
  - Effect : Iron Man eliminates Thanos.

Node A node is an object being expanded in the search procedure from the search tree.

Definition. $Node = \{S, D, P, O, PC\}$ is the representations of the node where,

- S is the state of the world,
- D is the depth of the given node in the search tree,
- P is the parent node of the given node,
- O is the operator used to reach the given node,
- PC is the cost of the path from the root to the given node.

Path Cost The path cost for a given state in the search tree is the cost from the root to the given state, where the cost is calculated cumulatively. The path cost for a generic problem is chosen to be the most appropriate to the description of the problem, and the factors the elements that are aimed to be reached at some specific state. For the stated problem, the element that is prioritized the damage sustained from the environment, where the optimal solution would have the damage is minimized. Therefore the path cost is the cost of each operator applied in the path, in addition to the following defined situations.

- Adjacent to warrior :
  - Preconditions : Iron Man must be in a cell adjacent to a warrior.
  - Effect : Iron Man sustain one damage unit.

- Adjacent to Thanos :
  - Preconditions : Iron Man must be in a cell adjacent to Thanos or entering Thanos' cell.
  - Effect : Iron Man sustain five damage units.

State Space State Space is the set of states in the world. By definition, state space is the set of states reachable from the initial state by any possible sequence of actions, which means all states possible to occur in the world. However in this problem the state space will be defined as the set of the visited states throughout the whole search tree at some given state. Hence, at a given state, it could be checked if it was visited earlier in the search tree or not, as in the problem repeated states are redundant. A repeated state is a state that was expanded in an earlier iteration in the search procedure. In the stated problem a sub goal initially is to collect one of the uncollected stones. In addition, any cell can be reached in a grid by visiting each cell once in the world state. A world state is defined as the stones collected/uncollected and the warriors eliminated/existing in the world. A state of the world differs when different stones are collected or/and different warriors are eliminated. Therefore to reach any sub goal at a give state of the world, the agent would need to visit each cell once. Moreover the state space stores $\sum_{i=0}^{d} unique(b^i)$ states, where d is the shallowest depth to a goal, the $unique$ function returns the number of the states which are not visited yet by the search agent, and b is the number of branches expanded at each node.

Goal Test A goal test is the procedure that checks for the satisfiability of the conditions that the goal state should have. Each state is checked when generated by the expansion function to have a complexity of b^d instead of b^{d+1} for the additional check required by the procedure after dequeuing each state for expansion. In the stated problem the goal is to have executed the operator *snap*, and have sustained a path cost less than *one hundred*.

4 Implementation

In this section, the implementation of the search agent and the encoding of the problem are presented. This includes the problems in the time complexity and space complexity for the search agent, the various approaches taken throughout the project, the problems in the search agent, etc. The implementation is presented in the intuitive way of constructing the search agent. Note that the code shown in this section is for explanation purposes, and some of the structure might be different.

Space Complexity In the EndGame search problem, one of the main problems is the space complexity as repeated states are not allowed to be expanded. Therefore each state generated is inserted into the state space of visited states. There are $2^{|W|+|S|}$ states of the world, where W is the set of warriors in the world, S is the set of stones in the world, and 2 is the number of possible states of each element in the warriors and stones sets. In addition, there are approximately $m*n$ allowed moves; hence there is a total of $m*n*2^{|W|+|S|}$ possible states. Intuitively, as the grid dimension grows, the more memory would be needed to reach a solution. The eagle-eyed reader would also notice that increasing the number of warriors or stones would increase the amount of memory needed exponentially. The state space stores the visited states; therefore the memory required is directly proportional to the memory required to store each state. Furthermore, the number of references required for each object is more expensive than the number of bits required for each object.

Hence focusing on a state representation with less references is more vital than decreasing the number of bits for the representation. For each reference, a different allocation in the memory is used, which would increase the overhead with the increase of the number of references. Moreover, some objects might prevent the use of specific blocks in the memory for other objects, as each object is allocated to a memory partition that has greater than or equal number of blocks that are required. Consequently memory fragmentation might be caused, and the procedure would run out of memory heap. In addition each reference requires more operations to retrieve from the memory.

Time Complexity After discussing the *Space Complexity* problems, another problem arise. The problem is the access to the state space. This would lead to one of the main factors for the time required for the procedure to halt. Storing the states in an array would be expensive, as each check for the states in the array would cost $O(n)$, where n is the number of visited states at a given time step. Therefore one approach to solve this would be to implement some data structure that has constant access time. Hence Hashing is the appropriate data structure to be used, as the cost for the access of an element is $O(1)$ amortized. Specifically, hashsets would be the most appropriate, as it allows for storing multiple elements with the same hash, which is required as there might be a repeat of hash code between two or more unique states. Hashsets are implemented with buckets, where each bucket is indexed through the hash of the objects. Intuitively, the complexity for the search in a hashset is proportional to the size of the bucket. Therefore, the better the hashing function is the better the complexity of the access to the hashset.

State Model A state defines the information necessary to describe the orientations of the objects inside of the world. The generic class for state is empty as each problem has a unique requirements to define the world of the problem. Furthermore, states should be comparable to be able to compare between the state in the manner the problem requires.

Listing 2: State model

```
public abstract class State implements Comparable<State>{

}
```

In the world of the *EndGame* problem, only the agent has the ability to move, while all of the other objects in the environment are stationary. So, an indicator for the existence of the warrior(s) and stone(s) at a given time step would be sufficient. Initially the state model was defined as discussed in ***State paragraph 3***, where the position of Iron Man was represented as a pair of bytes indicating the position of Iron Man in a give state, and the sets for the warriors and stones were represented as two bit arrays, where each bit in the array indicates the state of the warrior or stone. However as the state model required three references, and as discussed in ***Space Complexity paragraph 4***, the memory ran out of heap. Therefore another state model was deduced where the number of references were minimized to one reference. Before discussing the reasoning, the following must be stated, there must be exactly six stones in the world at the initial state, and Iron Man position only requires two bytes to be stores, as the maximum dimension the grid can be in the problem is 15x15.

Listing 3: Avenger's State model

```
public class AvengersState extends State {
byte [] grid;

    public AvengersState(byte [] gridStatus) {
        this.grid ← gridStatus;
    }
}
public class Main {
.
.
    public static String solve(String grid, String strategy, ↪
   boolean visualize) {
        .
        .
        byte[] gridStatus ← new byte[encoding.length + 2];
            gridStatus[0] ← iron.getX();
            gridStatus[1] ← iron.getY();
            for (int i ← 2; i < gridStatus.length; i++) {
                gridStatus[i] ← Byte.valueOf((byte) (i − 2));
            }
        AvengersState initialState ← new AvengersState(↪
   gridStatus);
        .
        .
    }
    .
    .
}
```

Therefore the information of the state can be stored in one byte array where $grid[0:1] \leftarrow$ Iron Man position, $grid[2:7] \leftarrow$ stones indices, $grid[8:end] \leftarrow$ warrior indices, and the required space per state is equal to $8*(2+|S|+|W|) = 8*(8+|W|)$. In addition, storing the indices of the warriors and stones represented as bytes of their positions that is stored in the coordinates set in $EndGame$ as defined in ***Search Problem paragraph 3*** instead of a bit representing the state of the warriors and stones. Moreover, this representation allows to remove the element from the array in the state when the stone is collected or the warrior is eliminated in a destructive array manner. Hence the the space required for each state decreases as the depth increases, and the time complexity required for the checks on the state of the object decreases. For simplification, the array in the state representation is mapped one to one to the coordinates array initially, $grid[0:1] \leftarrow$ Iron Man position, $grid[2] \leftarrow$ grid dimensions, $grid[3] \leftarrow$ Thanos position, $grid[4:9] \leftarrow$ stones indices, $grid[10:end] \leftarrow$ warrior indices, and the required space per state is equal to $8*(4+|S|+|W|) = 8*(10+|W|)$.

The visited states, $AvengersState$, are hashed in the state space hashset through overriding the $hashCode$ function, where the hash of each state is compromised of the hash of the x and y coordinates of iron man, and the hash of the hash of the $gird[2:end]$ array in $AvengersState$, where the first two indices are used to store the position of Iron Man. In addition, the hash generated for bytes are unique to some arbitrary value, which is not surpassed for the $EndGame$ problem.

Listing 4: Avenger's State hashing

```
public int hashCode() {
    return Objects.hash(this.getIron().getX(),this.getIron().↪
   getY(), Arrays.hashCode(this.getStatus()));
}
```

Node The generic class Node as discussed in ***Node paragraph 3*** is defined as the structure as $\{state, depth, parent, operator, cost\}$, all the instance variables are generic to be able to solve any problem that can be defined. Node implements comparable, where the nodes are compared by the value of the path cost, $cost$.

Listing 5: Node model

```
    public class Node implements Comparable<Node> {

        protected State state;
        private int depth;
        private Node parent;
        private String operator;
        private int cost;

        public Node(State state, String operator, int cost, int ↩
  depth, Node parent) {
            this.state ← state;
            this.operator ← operator;
            this.cost ← cost;
            this.depth ← depth;
            this.parent ← parent;
        }
        .
        .
        @Override
    public int compareTo(Node arg0) {
        return Integer.compare(this.getPathCost(), arg0.↩
  getPathCost());
    }
    .
    .
}
```

Operator The generic class Operator discussed in ***Operator paragraph 3*** is defined the set of actions defined in the problem. Each operator compromise of a name and its cost, and function that applies that operator on the state.

Listing 6: Generic Operator

```
    public abstract class Operator {
        private String name;
        private int cost;
        public Operator(String name, int cost) {
            this.name ← name;
            this.cost ← cost;
        }
        public String getName() {
            return this.name;
        }
        public int getCost() {
            return this.cost;
        }
        public String toString() {
            return this.getName();
        }
        public abstract Node applyOperator(Problem problem,↩
  Node node);
    }
```

Initially, the operator applicability is checked, as some of the operators require the same checks for their preconditions. Hence, the function $availableActions$ in the $EndGame$ class checks for the applicable actions for a given state, which follows the preconditions discussed in ***Operator paragraph 3***. Each operator is implemented in a separate class that extends the $Operator$ class. The *movement* operators are collected in one class that take a value input from one to three to indicate the direction of the movement. The move applicability is checked if Iron Man is transitioning to cell that is either empty or has a stone. Afterwards the $grid$ array for the successor state is created by changing the position of the Iron Man, and the new path cost of the state is added. As for the *kill* operator, if Iron Man is in a cell for a given state that has at least one warrior, then an array of the indices of the adjacent warriors is given as input to the operator. The warriors are removed from the successor state array model representation $grid$, and the cost of the warrior elimination is added to the path cost. Meanwhile, the *collect* operator takes the stone to be collected index if Iron Man was in a cell that contained a stone in a given state. The stone index is removed from the successor state array model representation $grid$, and the cost of the collect is added to the path cost. Lastly, the *snap* operator removes Thanos from the successor state array model representation $grid$ if all the stones were collected and Iron Man is in the cell that Thanos is in.

Search Problem The generic search problem as discussed in ***Search Problem paragraph 3***, the search problem class is a generic definition for any problem that can be defined. $expandedNodes$ is the counter that counts the number of expanded nodes in the generic search procedure.

Listing 7: Search Problem model

```
  public abstract class Problem {

      protected Node initialNode;
      protected HashSet<State> statespace;
      protected Operator[] operators;
      protected int expandedNodes;
      .
      .
      public abstract boolean goalTest(Node node);
      public abstract int pathCost(State state);
      .
      .
  }
```

As noticed, a constructor is not implemented for the generic search problem class. This is to allow the programmer freedom in the definition of the problem, and allows for more flexible implementations for the modeling of the problem. For example, in the implementation of EndGame discussed in this report, the array of operators is not used as will be discussed later.

As for the EndGame problem, as discussed in ***Search Problem paragraph 3***, $EndGame$ is a subclass to $Problem$, and the instance variable $coordinates$ is the description of the positions of Thanos, the six stones, and all the warriors present in the grid.

Listing 8: EndGame Problem model

```
  public class EndGame extends Problem {

      private Cell[] coordinates;

      public EndGame(State initialState, Cell[] coordinates) {
          this.coordinates ← coordinates;
          this.initialNode ← new Node(initialState, null, ↩
  pathCost(initialState), 0, null);
          this.statespace ← new HashSet<State>();
          this.statespace.add(initialState);
      }
      .
      .
  }
```

The path cost function and the goal test are implemented as discussed in ***Path Cost paragraph 3*** and ***Goal Test para-***

graph 3 respectively. For further inspection to the implementation for these functions, please refer to the program code.

Transition The transition function is divided into two partitions. Initially all of the applicable operands are retrieved through *availableActions* function, where a node is given as an input. *availableActions* reduces the number of checks required to be done for each operand, as some of the checks between the operands overlap. For example, the movement operands checks if any warrior is in an adjacent cell, while the kill operand does the same checks. The second part of the process, expands the given node with applying all the applicable operands. Each generated node is checked if it was visited or not and does it exceed the damage unit sustained constraint or not before getting enqueued in the search procedure.

Listing 9: Node Expansion

```
public ArrayList<Node> expand(Node node) {
    ArrayList<Node> successorNodes ← new ArrayList<↩
  Node>();
    ArrayList<Operator> operators ← availableActions(node)↩
  ;
    for (Operator o : operators) {
        Node successorNode ← o.transition(this, node);
        if (successorNode !← null && successorNode.↩
  getPathCost() < 100 && !isVisitedState(successorNode)){
            successorNodes.add(successorNode);
            addState(successorNode.getState());
        }
    }
    return successorNodes;
}
```

The repeated states are checked before enqueuing in the search procedure, hence decreasing the space required in storing the nodes that would not get expanded and decrease the time needed to reach a solution. The continuation of discussion of this point will be in ***Discussion paragraph 5***

Search The search procedure as discussed in the ***Search paragraph 2***, is a generic procedure for searching for a goal in any problem. The queuing function discussed in the lecture is implemented as an abstract class that abstracts the data structure, and functions that are used for each of the search strategies. Each search strategy requires to add a node, remove a node, and check if the queue is empty. In addition, the add functionality has an additional function which takes all nodes added to the queue in an $ArrayList$ to add them in the appropriate order. This aroused from *Depth First Search*, where the nodes generated for each expanded node were added in the reverse order. As for the $isEmpty$ function, the $isEmpty$ function for data structure used for the search strategy is invoked. Note that the search strategies discussed are subclass of $QueuingFunction$.

The search procedure follows the pseudo code discussed in ***Search paragraph 2***. Initially the initial node of the problem is enqueued to the queue. Afterwards, the procedure iterates on the queue till it is empty, expands each node, adds the expanded state to the state space, and enqueues the generated nodes in the queue.

Listing 10: Search Procedure

```
public abstract class QueuingFunction {

    Collection<Node> queue;

    public abstract void add(ArrayList<Node>  nodes);
    public abstract void add(Node  node);
    public abstract Node remove();
    public abstract boolean isEmpty();

}
public abstract class Problem {
    public Node search(QueuingFunction nodes) {
        nodes.add(getNode());
        addState(getNode().getState());
        while (!nodes.isEmpty()) {
            Node node ← nodes.remove();
            expandedNodes++;
            if (goalTest(node)) {
                return node;
            }
            ArrayList<Node> successorStates ← expand(node)↩
  ;
            nodes.add(successorStates);
        }
        return null;
    }
}
```

Breadth First Search is implemented with a linked list, where each element is added in the end of the queue, and the element dequeued is removed from the head of the linked list. The add a list of the generated nodes function adds the nodes in the order it was given.

Listing 11: Breadth First Search

```
    public BFS() {
        super.queue ← new LinkedList<Node>();
    }
    @Override
    public void add(ArrayList<Node>  nodes) {
        for(Node n:nodes) {
            ((LinkedList<Node>) super.queue).add(n);
        }
    }
    @Override
    public Node remove() {
        return ((LinkedList<Node>) super.queue).removeFirst↩
  ();
    }
```

Depth First Search is implemented with a linked list, as linked lists allow insertion and removal at arbitrary positions elements in constant time through positions given by an iterator, and not indexing [2]. Therefore for the strict use used in search problem, it was observed that linked lists have better computational time. Moreover, linked list has the stack push and pop functions pre-implemented.

Listing 12: Depth First Search

```
    public DFS() {
        super.queue ← new LinkedList<Node>();
    }
    @Override
    public void add(Node node) {
        ((LinkedList<Node>) super.queue).push(node);
    }
    @Override
    public Node remove() {
        return ((LinkedList<Node>) super.queue).pop();
    }
```

Uniform Cost Search is implemented with a priority queue which orders the elements through the path cost of the nodes. Priority queues are implemented with minimum heaps in java which allows insertion and removal of elements in $O(log(n))$ [3].

Listing 13: Uniform Cost Search

```
  public UCS() {
      super.queue ← new PriorityQueue<Node>();
  }
  @Override
  public void add(Node node) {
      ((PriorityQueue<Node>) queue).add(node);
  }
  @Override
  public Node remove() {
      return ((PriorityQueue<Node>) queue).poll();
  }
```

Iterative Deepening Search is implemented with a linked list, the implementation of the *Iterative Deepening Search* strategy is like the implementation of the *Depth First Search*; however *Iterative Deepening Search* has a cutoff depth at each iteration as discussed in ***Iterative Deepening paragraph 2***. The difference is in the add and $isEmpty$ functions, where the add function checks for the cutoff value at the iteration, $isEmpty$ function starts the search over from the root and increase the value of the cutoff.

Listing 14: Iterative Deepening Search

```
  Problem problem;
  int depth ← 0;
  public IDS(Problem problem) {
      this.problem ← problem;
      super.queue ← new LinkedList<Node>();
  }
  @Override
  public void add(Node node) {
      if (depth >← node.getDepth()) {
          ((LinkedList<Node>) super.queue).push(node);
      }
  }
  @Override
  public boolean isEmpty() {
      if (((LinkedList<Node>) queue).isEmpty()) {
          depth++;
          if (depth <← Integer.MAX_VALUE) {
              super.queue ← new LinkedList<Node>();
              this.problem.emptyStateSpace();
              add(this.problem.getNode());
          }
      }
      return ((LinkedList<Node>) queue).isEmpty();
  }
```

Greedy Search is implemented in a similar manner as the *Uniform Cost Search*. As discussed in ***Greedy Search paragraph 2***, the priority queue comparator function is overridden to only consider the value of the heuristics. For the *Greedy Search* to be generic to any search problem, the $function$ interface is used, which allows to pass a function as an argument to the constructor. Furthermore, the input and output types for the variables must be specified in the definition of the variable.

Listing 15: Greedy Search

```
  public GS(Function<Node, Integer> heuristicFunc) {
      super.queue ← new PriorityQueue<Node>(new ↩
Comparator<Node>() {
          @Override
          public int compare(Node node1, Node node2) {
              int heuristicCostA ← heuristicFunc.apply(node1)↩
;
              int heuristicCostB ← heuristicFunc.apply(node2)↩
;
              return Integer.compare(heuristicCostA, ↩
heuristicCostB);
          }
      });
  }
```

A* Search is implemented in a similar manner as the *Uniform Cost Search*. As discussed in ***A* Search paragraph 2***, the priority queue comparator function is overridden to consider the summation of the path cost value and the value of the heuristics. For the *A* Search* to be generic to any search problem, the $function$ interface is used.

Listing 16: Greedy Search

```
  public AS(Function<Node, Integer> heuristicFunc) {
      super.queue ← new PriorityQueue<Node>(new ↩
Comparator<Node>() {
          @Override
          public int compare(Node node1, Node node2) {
              int heuristicCostA ← heuristicFunc.apply(node1)↩
 + node1.getPathCost();
              int heuristicCostB ← heuristicFunc.apply(node2)↩
 + node2.getPathCost();

              return Integer.compare(heuristicCostA, ↩
heuristicCostB);
          }
      });
  }
```

Heuristics A heuristic is a value that directs the traversal of an algorithm for the path to choose for a decision to reach a solution for a problem.An admissible heuristic underestimates the cost from a given state at some node to the goal state in the search tree [1]. In the implementation of the $EndGame$ problem two admissible heuristics were implemented. The first heuristic calculates the damage of collecting the remaining stones in the world in some state, the value that is inflicted by Thanos when Iron Man is adjacent after all the stones are collected, and the value of the damage inflected to Iron Man when he enters the cell Thanos is in before snapping. The first heuristic is admissible as the damage calculated must be inflicted to the agent, as the goal is collect all the stones in the world, and performing the snapping action. To perform the snapping action, Iron Man first would need to go to cell that is adjacent to Thanos and enter the cell to perform the action, which would inflect 10 damage units to Iron Man. As for the second heuristic in addition to calculating the value of the remaining stones in the world, and the damage inflicted by Thanos to perform the snapping action, the heuristic calculates the damage of the warriors and Thanos for the stones that adjacent to them. The second heuristic underestimates the cost, as in addition to the cost that is calculated from the first heuristic, the second heuristic calculates the damage units that must be inflicted to Iron Man depending on the position of the stones. Therefore The second heuristic is admissible and dominates the first heuristic.

5 Evaluation

The agent performance is evaluated through the magnitude of the scores achieved in the grid configuration that will presented. The performance is measured according to the path cost, and the number of nodes that are expanded. A comparison of the performance of the different strategy of the search agent are discussed in this section. Afterwards the discussion about the analysis of the results is presented.

Performance In this report, two examples for the $EndGame$ problem are discussed with details. First, the specifications of the machine used for testing is laid out. The computer used for testing had Intel Core i7-4790K Quad-core 4.00GHz base clock Speed, 16 GB 2133MHz DDR3 RAM, and the results were as following.

Input Grid : 5,5; 0,0; 2,3; 1,3, 2,2, 0,4, 4,0, 3,4, 2,0; 1,0, 1,2, 4,2, 3,3, 0,1

	0	1	2	3	4
0	I	W			S
1	W		W	S	
2	S		S	T	
3				W	S
4	S		W		

Figure 1: **5x5 Grid**- 6 Stones & 5 Warriors

BFS :

- Output :

```
kill,down,down,collect,down,down,collect,right,up,↩
right,up,collect,down,kill,right,right,collect,up,up,up,↩
collect,left,down,collect,down,snap;70;11061
```

- Path cost : 70.
- Nodes expanded : 11061.
- Time taken : 68 Milliseconds.

DFS :

- Output :

```
kill,right,right,right,right,collect,left,left,left,left,down↩
,down,collect,right,right,collect,left,left,up,up,right,↩
right,right,right,down,down,down,collect,up,up,left,↩
collect,right,down,down,down,left,kill,left,left,left,↩
collect,right,right,right,right,up,up,left,snap;83;52
```

- Path cost : 83.
- Nodes expanded : 52.
- Time taken : 8 Milliseconds.

UCS :

- Output :

```
kill,down,down,down,down,collect,up,up,collect,up,↩
up,right,right,kill,right,right,collect,left,down,collect,↩
left,left,down,down,down,kill,right,right,right,up,kill,↩
collect,down,left,left,up,up,collect,right,snap↩
;57;11393
```

- Path cost : 57.
- Nodes expanded : 11393.
- Time taken : 87 Milliseconds.

IDS :

- Output :

```
kill,right,right,right,right,collect,left,down,collect,↩
right,down,down,collect,kill,left,left,left,left,up,collect↩
,down,down,collect,right,up,right,up,collect,right,↩
snap;69;69407
```

- Path cost : 69.
- Nodes expanded : 69407.
- Time taken : 194 Milliseconds.

GR1 :

- Output :

```
kill,down,down,collect,down,down,collect,up,up,up,↩
right,kill,down,right,collect,up,right,collect,right,up,↩
collect,down,down,down,collect,up,left,snap;63;65
```

- Path cost : 63.
- Nodes expanded : 65.
- Time taken : 48 Milliseconds.

GR2 :

- Output :

```
kill,right,down,kill,left,down,collect,right,down,down↩
,left,collect,right,up,right,up,collect,down,kill,right,↩
right,collect,up,up,left,collect,right,up,collect,down,↩
down,left,snap;75;75
```

- Path cost : 75.
- Nodes expanded : 75.
- Time taken : 49 Milliseconds.

AS1 :

- Output :

```
kill,down,down,collect,down,down,collect,right,kill,↩
right,right,right,up,collect,down,left,left,left,up,up,↩
left,up,up,right,right,kill,right,down,collect,up,right,↩
collect,left,left,down,down,collect,right,snap;58;9660
```

- Path cost : 58.
- Nodes expanded : 9660.
- Time taken : 148 Milliseconds.

AS2 :

- Output :

```
kill,right,down,kill,down,left,collect,down,down,right↩
,kill,right,right,right,up,collect,down,left,left,left,left,↩
collect,up,right,up,up,up,right,right,right,collect,↩
down,left,collect,left,down,collect,right,snap;58;9728
```

- Path cost : 58.
- Nodes expanded : 9728.

- Time taken : 211 Milliseconds.

Input Grid : 7,7; 3,3; 1,1; 0,1, 1,0, 1,2, 2,1, 4,4, 6,0; 0,0, 0,2, 1,3, 2,0, 2,2, 3,1, 3,4, 4,3, 5,0, 0,6, 5,6, 6,6

	0	1	2	3	4	5	6
0		S	W			W	W
1	S	T	S	W			
2	W	S	W				
3		W		I	W		
4				W	S		
5	W						W
6	S						W

Figure 2: **7x7 Grid**- 6 Stones & 12 Warriors

BFS :

- Output :

```
kill,right,down,collect,left,left,left,left,kill,down,down↩
,collect,up,up,up,kill,right,up,collect,left,up,collect,up↩
,right,collect,kill,right,down,collect,left,snap↩
;92;1938714
```

- Path cost : 92.
- Nodes expanded : 1938714.
- Time taken : 6040 Milliseconds.

DFS :

- Output :

```
left,down,left,down,down,left,collect,right,right,right↩
,right,right,up,up,left,collect,right,right,up,up,left,left↩
,left,kill,left,left,collect,right,right,right,right,up,kill,↩
left,left,left,collect,right,up,kill,left,left,collect,left,↩
down,collect,right,snap;99;2206628
```

- Path cost : 99.
- Nodes expanded : 2206628.
- Time taken : 4593 Milliseconds.

UCS :

- Output :

```
kill,down,down,down,left,left,left,collect,right,right,↩
right,right,up,up,up,up,up,kill,left,up,kill,down,right,↩
down,down,down,collect,up,up,left,kill,left,left,collect↩
,right,up,collect,up,left,collect,left,down,collect,right,↩
snap;84;4641211
```

- Path cost : 84.
- Nodes expanded : 4641211.
- Time taken : 18144 Milliseconds.

IDS :

- Output :

```
left,down,left,down,down,left,collect,right,right,right↩
,right,right,up,up,left,collect,right,right,up,up,left,left↩
,left,kill,left,left,collect,right,right,right,right,up,kill,↩
left,left,left,collect,right,up,kill,left,left,collect,left,↩
down,collect,right,snap;99;5037259
```

- Path cost : 99.
- Nodes expanded : 5037259.
- Time taken : 8117 Milliseconds.

GR1 :

- Output :

```
left,kill,down,kill,down,down,right,right,up,up,collect↩
,left,left,left,up,up,collect,right,up,collect,kill,right,↩
down,right,kill,down,left,left,down,left,down,kill,left,↩
down,collect,up,up,up,kill,up,up,collect,up,right,↩
collect,down,snap;98;22765
```

- Path cost : 98.
- Nodes expanded : 22765.
- Time taken : 186 Milliseconds.

GR2 :

- Output :

```
kill,left,kill,up,up,collect,kill,up,left,collect,left,down,↩
collect,up,right,right,right,right,down,down,down,↩
down,collect,left,down,down,left,left,up,kill,left,down,↩
collect,right,up,up,up,up,collect,up,snap;99;10284
```

- Path cost : 99.
- Nodes expanded : 10284.
- Time taken : 193 Milliseconds.

AS1 :

- Output :

```
kill,down,down,down,left,left,left,collect,right,right,↩
up,up,left,kill,up,left,kill,right,down,right,down,right,↩
right,up,collect,up,up,up,kill,left,up,kill,left,left,↩
collect,right,right,down,down,kill,left,left,collect,left,↩
up,collect,down,down,right,right,up,up,collect,left,↩
snap;84;3767307
```

- Path cost : 84.
- Nodes expanded : 3767307.
- Time taken : 21958 Milliseconds.

AS2 :

- Output :

```
kill,down,left,left,kill,up,left,kill,up,up,collect,down,↩
down,right,right,kill,right,right,down,collect,down,left↩
,down,left,left,left,collect,right,right,right,right,up,up,↩
up,up,up,kill,left,up,kill,left,left,collect,right,down,↩
collect,down,left,collect,up,snap;84;1641271
```

- Path cost : 84.

- Nodes expanded : 1641271.
- Time taken : 14501 Milliseconds.

Discussion The evaluation shows that overall *A** is the best search strategy in the number of nodes expanded and guaranteeing the optimal or sub optimal solution to the problem. The situation for the search strategies are a bit different from the reader's deduction after reading the discussed material in ***Background Section 2***. The main difference returns to the fact that in the $EndGame$ problem does not expand repeated states. In addition, to improve the computational time required for the procedure to find a solution, the repeated states are removed when they are generated. Hence, the space required in the queue to store the nodes is decreased as the queue is ensured to not have repeated states, and the time required for inserting for optimal strategies is decreased. However as the state model does not include the health sustained at a give state, a better path might be considered as repeated as the agent had visited the path previously. Therefore the strategies characteristics might differ. As the state model does not have the cost as an argument, because there is no computational power sufficient to be able to include it to the state. Consequently, some nodes that would reach the combination of applied operators required for the optimal result might be considered as repeated states. Furthermore, some of the search strategies might not be able to find a solution to the problem, but by only changing the order of operators would result with a solution. Moreover, *A** may not reach an optimal solution, as the path leading to it might be considered repeated.

Initially, let's discuss the optimality and completeness of each search strategy according to the $EndGame$ problem. There is a difference that can be deduced to the ultra sharp-eagle sighted reader, in particularly the implementation discussed that improves the computational time required, which improves the space and time complexity through checking on the repeated states in the enqueuing phase instead of the dequeuing step. Unfortunately, the main fallback is that it does not ensure completeness for any of the search strategies. For example, if the grid configuration had it so that the only solutions would be reached with a path cost 99, then the path leading to one of the solutions might be considered repeated at some state in the path leading to it. Therefore none of the strategies are optimal as will be discussed in further details. Firstly, *BFS* is not optimal as discussed in ***Breadth First Search paragraph 2***. As for *DFS*, the search strategy is not optimal as discussed in ***Depth First Search paragraph 2***; however the strategy is not complete though there cannot be infinite branches as the state model does not handle the cost in the repeated state comparison. Moreover, the inclusion of the path cost to the state model would have not been beneficial as most states would have not been considered repeated. In addition, if there existed some object in the world that would have increased the cost, a warrior in a cell adjacent to Iron Man, then after each change in the cost, that would be considered changed and the agent would be able to visit repeated cells without any change in the world other than the cost of the path. Furthermore, the order the operators are applied to the expanded node is a factor for *DFS* to not find a solution even though another order operator would have found a solution to the problem. Meanwhile, for *UCS*, the search strategy is neither optimal nor complete contrary to what is discussed in ***Uniform Cost Search paragraph 2***. Intuitively, one can argue that *UCS* might not be complete mainly because the move operators cost zero to be executed; however repeated states are not enqueued to the nodes queue in the search procedure, therefore there are a limited amount of moves that can be executed in each state of the world, and that would not affect the search strategy's completeness. As for optimality, in contrary to *UCS* main advantage, *UCS* is not optimal in the implementation discussed to improve the computational time required by the search procedure and not considering the cost if the path in the state mode of the $EndGame$ problem, as discussed. Moreover, if there there were two states that differ in one of the collected stones, and one of them differ in the cost than the other by one damage. In addition, the stone that is not collected to the state that has lower damage has a warrior adjacent to it. The state with the lower cost would be expanded first in the optimal search strategies. At the point that the agent reaches the stone, the cost would be equal to the other node, let's assume that this node would be expanded before the other and collect the stone while sustaining the damage of the adjacent warrior. Afterwards the other node that is more optimal would be considered repeated when generates the path that collects the stone that it differed. On the other hand, *IDS* is neither complete nor optimal, contradictory to the discussion in ***Iterative Deepening Search paragraph 2***. Furthermore, *Greedy Search* is not optimal as discussed in ***Greedy Search paragraph 2***, but *Greedy Search* is not complete regardless to the heuristic's admissibility. Despite, intuitively thinking that not allowing the expansion of repeating state would allow *Greedy Search* to be complete regardless of admissibility of the heuristic in the restrictions enforced in the $EndGame$ problem. However, similar to *DFS*, as the health, the path cost, is not considered in the state comparison, which would affect the backtracking process, as the state which would lead to the goal state and have a cost less than a 100 might be considered repeated. Finally, for *A** is not complete, despite the discussion in ***A* Search paragraph 2***. In addition, *A** is not optimal because of the repeated states, and the admissible heuristic does not get the actual cost from a given to a goal state. The reason is that the state model does not reflect the actual difference between actual repeated state and ones that are not, as discussed. Therefore, it can be deduced that the optimal search strategies get an approximate optimal solution. Furthermore, the computational time required for the search agent to find a solution is improved, which allows the agent to solve harder configurations of the $EndGame$ problem.

The characteristics of each search strategy defines the relation of the number of nodes required to be expanded to reach a solution. Furthermore, the completeness and optimality of each search strategy has a relation to the number of nodes required to be expanded. Though as discussed for the $EndGame$ problem, none of the search strategies are either optimal or complete. From observations and deductions, the search strategies node expansion order is be discussed in descending order as follows. First, *IDS* expands the most number of nodes, as the iterations repeat in the strategy till the cutoff value reaches the shallowest depth that has a goal. Afterwards, *UCS* is the second most strategy that expands nodes, as the nodes are expanded depending on the path cost of each node. Following, the *AS* is the third most strategy that expands nodes, as it considers the summation of the path cost of the node and the heuristic value of the node state. Therefore, the number of nodes *AS* expands correlate with the dominance of the admissible heuristic. In succession, *BFS* expands nodes till the shallowest depth that has the goal, as *BFS* does not depend on the cost of the path. *DFS* expands less nodes than *BFS* if the goal can be reached is few number of succession of some operator, however as the world has more objects, and require more mixture of actions to reach a goal state that satisfies the constraint. Lastly, *Greedy Search* expands the least amount of nodes, as it guided by the value of the heuristic function in the traversal of the search tree.

6 Conclusion

In conclusion, the problem defined was expensive in both time and space complexity. Therefore, an approach that focuses on the computational power required was followed, which has the main disadvantage that made all of the search strategies neither optimal nor complete. However the program reaches an approximate optimal solution in most cases, and allows hard grid configurations to be solved in a better running time. Furthermore, each search strategy has its own advantages and disadvantages depending on each problem. For the $EndGame$ problem the search strategy *A** is observed as the most appropriate as it minimizes the cost, expands less node than *UCS*, and requires less computational time in hard configurations of the world where there exits more than five warriors. Despite the theoretically optimal state model raised a problem because of not including the cost of the health in the state comparison. Unfortunately, this problem cannot be handled by the search strategies discussed in this report. Therefore, one of the better approaches would to consider the knowledge deduced at each state and consider the consequences through reasoning about the agent's beliefs.

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2009.

[2] 5gon12eder, "Linkedlist vs stack."

[3] Amaninder.Singh, "Min heap in java."