CSEN-901 Project 1

TEAM-55

NAME	ID	TUTORIAL#
MAYAR OSAMA	37-2492	T-08
RANEEM RASHAD	37-0917	T-08
NOUR RAED	37-2082	T-08

The problem that we are solving is a search problem. The goal is to search for a sequence of actions where Iron Man can collect all stones, go to the cell where Thanos is standing and snap before his damage reaches 100. We solve this problem by creating a search tree with the initial state of the problem being the root. For every node the actions that Iron Man can perform represent the branches. We repeat the expansion of the nodes to branches until we find a node where Iron Man reaches his goal.

To help our problem to reach the goal faster we set the order of allowed actions to be from the most important to preform to the least important. So first we have the collect action since it is our top priority is to collect all 6 stones, then we have the 4 movement allowed so that we can reach Thanos or stones as fast as possible, and finally we have the kill action which is our least significant action since the goal doesn't include killing warriors if they are not in our way.

The implementation goes as follows, starting by defining a ***SearchTreeNode*** class, which consists of a *State* attribute which is a general state, an *action* attribute that led to the current node, *costFromRoot* the total cost from root till that node, an attribute representing the *depth* of the current node, a *greedyCost* function and the *_AstarCost* for the current node, and finally a *parent* attribute which is a SearchTreeNode to reference the node before applying the action. And finally each attribute has its own getter and setter method.

The ***State*** class is an abstract empty class that could be used in any problem. Then we have an extended version of the State class specific for our problem which is the ***StateEndgame.***

The ***StateEndgame*** contains the *position* of iron man in this state, a HashSet of the *stonesLeft* in the current state which contains the position of stones left to collect, a HashSet of the *warriorsLeft* in the current state which contains the position of warriors left in the grid, an integer representing the *damage* of iron man so far, and an integer *warriorsArrounStones* which is used by the second admissible heuristic function, it uses the position of the stones to calculate the count of warriors around each stone. The class contains getters and setters methods and a *warriorsAroundStones* helper method.

The *warriorsAroundStones* method calculates the number of warriors around each stone by looping over the *stonesLeft* set to get the position of each stone left to collect and count the warriors around that stone by checking the *warriorsLeft.contains* method and check on the four locations around the stone's location (above it, below it, on its right and its left) and adds the count to the total count of warriors around stones.

As for the ***Action*** class it is an abstract class that contains an array of strings representing the valid actions depending in the problem. And a subclass ***ActionEndgame*** that extends the ***Action*** class.

Moving on to the ***GenericSearch*** class, which can apply a generic search on any given problem.

It has the following attributes: *State initialState* attribute that contains the initial state of the given problem, a set of allowed *actions* of class *Action,* an integer *limit* that is used by the Iterative Deepening Search, and a HashSet for the repeated states *repeatedSet* which is a set of ***KeyState***.

KeyState is a class that is used for checking the repeated states, it has the following attributes: a String *position*, a HashSet of Strings for the *stonesLeft*, a HashSet of Strings for the *warriorsLeft*.

Its constructor takes a ***State*** as a parameter and from the given state we initialize the attributes, the class also has getters and setters for the attributes, overriding the *hashCode* to add the position and each stone's and warrior's position to the hash code of the class, also overriding the *equals* method to compare the ***KeyState*** by checking if their position are equal and same stones left in their sets, since we use the ***KeyState*** class in the HashSet of the repeated states so the states are first compared by their hash code then by the *equals* method.

Back to the ***GenericSearch*** class, which contains a *generalSearch* method that takes as input the grid, a strategy, a visualization Boolean that is used if we want to print every change with every action, and an input limit for the Iterative Deepening Search.

The *generalSearch* method starts by initializing the repeated state set and adds the initial state to it to avoid repeating it. Also adds the initial state to the queue of expanding nodes.

As long as the queue of nodes is not empty we keep looping. Each iteration we get the first node in the queue and increment the count of expanded nodes, we check if this node is a goal node by applying the goal test, in which we check if we are in the same cell as Thanos, collected all the stones and the damage is less than 100.

If the node is a goal node we generate the solution by getting the path from the root to that goal node and adds at the end cost needed from the root till that node and the number of expanded nodes and returns the solution.

If the node is not a goal node we generate its children by looping over the array of valid actions and calling the transition method to get the child of a given action from that node. First we check if this node already exists in the *repeatedSet*, if it exists we ignore that node if not we add it to the queue and add it to the *repeatedSet.* After that we apply one of the strategies to see how we are going to arrange the queue with the new children.

We have 8 strategies: *breadthFirstSearch, depthFirstSearch, iterativeDeepeningSearch, uniformCostSearch, greedy1*, *greedy2, _Astar1 and _Astar1.*

- The *breadthFirstSearch* adds the children to the end of the queue.

- The *depthFirstSearch* adds the children to the start of the queue at position 0 and shifts the rest of the queue to the right.

- The *iterativeDeepeningSearch* applies the iterative deepening search is similar to the depth first search by adding each node from the children to the beginning of the queue but taking into consideration a limit for the depth, if the node's depth is more than the limit we don't add it. Whenever the queue gets empty and solution is not found we increment the limit of the depth and recall the *generalSearch* method and start all over again.

- The *uniformCostSearch* uses a priority queue that sort the nodes according to the ***NodeCompartor*** that compares the nodes by their cost from root

- The ***greedy1*** first sets the greedyCost attribute of each node with the first heuristic function which is the damage from collecting all the stones left by calling the *calculateGreedy1* and passing the children nodes each at a time.
  So h(n) = stonesLeft.size()*3 ; 3 is the cost of collecting one stone
  Then we use a priority queue that is sorted by the ***GreedyNodes*** that compares the nodes by their greedyCost, and finally we arrange the queue and add the children nodes by putting them in the right position according to its greedyCost.

- The ***greedy2*** first sets the greedyCost attribute of each node with the second heuristic function which is the damage from collecting all the stones left and the damage if there is a warrior next to those stones by calling the *calculateGreedy2* and passing the children nodes each at a time.
  So h(n) = stonesLeft.size()*3 + *count of WarriorsArroundStones*; 3 is the cost of collecting one stone
  Then we use a priority queue that is sorted by the ***GreedyNodes***, and add the children to the queue accordingly.

- The ***_Astar1*** first sets the _AstarCost attribute of each node with the first heuristic function adding to it the cost from root till that node, which is the damage from collecting all the stones left by calling the *calculateAstar_1* and passing the children nodes each at a time.
  So h(n) = stonesLeft.size()*3 ; 3 is the cost of collecting one stone.
  This heuristic function is admissible since it only calculates the minimum damage that Iron Man can get, so the minimum is to collect all the stones left and only get their damage assuming that Iron Man will not receive from the warriors by either standing next to one or killing one, also ignoring the damage from standing next to Thanos or even being with him in the same cell to snap.
  Then we use a priority queue that is sorted by the ***AstarNodes*** that compares the nodes by their AstarCost, and finally we arrange the queue and add the children nodes by putting them in the right position according to its AstarCost.

- The ***_Astar2*** first sets the _AstarCost attribute of each node with the second heuristic function adding to it the cost from root till that node, which is the damage from collecting all the stones left by calling the *calculateAsta_2r* and passing the children nodes each at a time.
  So h(n) = stonesLeft.size() *3+ count of warriors around stones; 3 is the cost of collecting one stone.
  This heuristic function is admissible since it only calculates only the damage that IronMan can get from collecting all the stones left and the damage from standing next to the warriors surrounding the stones if exists. Therefore, it underestimates the damage since we ignored the case that he might need to kill some warriors to reach the stones and he might stand next to a warrior that is not around any stone while moving, also it ignores the damage from standing next to Thanos or being in the same cell with him to snap.

  Then we use a priority queue that is sorted by the ***AstarNodes*** that compares the nodes by their AstarCost, and finally we arrange the queue and add the children nodes by putting them in the right position according to its _AstarCost.

Now let's evaluate each search strategy. Since we have a limited valid actions that we can apply, the grid is not infinite and we don't allow repeated states, therefore, all the search strategies are complete. Regarding the optimality of the search strategies that we have, all our search strategies are either sub-optimal or not optimal because of the repeated states restrictions, since there might be a repeated state that helps us to reach the goal with less damage but we eliminate it to avoid the infinite branches.

The following two grids will be used in order to clarify the comparison:

Grid1: "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"
Grid 2: "6,6;5,3;0,1;4,3,2,1,3,0,1,2,4,5,1,1;1,3,3,3,1,0,1,4,2,2"

breadthFirstSearch

Grid1:
up,collect,down,right,right,collect,left,left,left,collect,down,left,collect,down,collect,right,collect,right,snap;34;1117

Grid2:
up,collect,right,right,collect,left,left,left,up,left,left,collect,up,right,collect,up,collect,right,collect,up,left,snap;47;5933

- Complete by default and the above restrictions made sure its complete.
- It's not optimal since the arrangement of the actions can change the output.
- The number of expanded nodes here is not small as we go through the whole branch before going to the next level, and since we need at least 7 levels (6 for collecting all 6 stones and one for being with Thanos at the same cell), and each level have multiple choices from collecting, moving in the 4 directions (if valid) or killing. Therefore, the number of expanded nodes will increase exponentially.

depthFirstSearch

Grid1:
left,left,down,down,right,collect,left,up,up,right,right,right,right,collect,left,left,left,left,down,down,collect,right,up,up,right,up,collect,left,left,down,down,collect,right,up,collect,left,down,down,right,right,snap;47;75

Grid2:
left,left,left,up,up,right,right,kill,left,left,down,down,right,right,right,right,right,up,up,up,up,up,left,left,left,down,left,kill,left,down,down,right,right,right,right,right,down,collect,left,left,left,left,left,up,up,right,right,right,right,kill,left,kill,left,left,left,down,down,right,right,right,collect,left,left,left,up,up,right,right,up,collect,left,left,down,down,collect,right,right,right,right,right,up,up,left,left,left,left,collect,down,collect,left,down,down,right,right,right,right,right,up,up,left,left,up,left,up,left,snap;68;213

- Normally DFS is not complete since it can run in an infinite branch but in our implementation DFS is complete since we don't allow repeated states so it can't be stuck in a loop forever so if there's a solution it will find it.

- It's not optimal since it doesn't take into account the costs, it just outputs the first solution it finds from going into the branches depth first.
- The reason DFS here has a relatively small, for a not informed search number of expanded nodes, is that the order of the actions that we used is (collect then the 4 movements then the last options is kill), so it will go to the first valid branch which has the higher priority so that iron man will collect a stone whenever he can first, and since killing is not a requirement we put it last option.

uniformCostSearch

Grid1:
left,collect,up,right,right,right,down,collect,up,left,left,collect,left,left,down,down,collect,down,collect,right,collect,right,snap;32;1201

Grid2:
right,right,up,collect,left,down,left,left,left,up,up,left,collect,up,right,kill,right,up,collect,down,left,left,kill,down,down,down,right,right,right,up,kill,collect,up,left,up,left,collect,up,collect,up,snap;37;7230

- Complete by default and the above restrictions made sure its complete.
- Optimal since it arranges the nodes to expand according to their damage and since each node's damage is less than or equal to the cost of its successors then it's ensured to be optimal.
- The number of expanded nodes here is relatively high since this search is not informed, we take into consideration only the cost of the node from root, so we expand as much nodes as we can as long as they have minimum cost. When in fact we need to get the damage from collecting the stones but it leaves those nodes as last option since they require damage/increase the cost.

iterativeDeepeningSearch

Grid1:
left,left,down,down,collect,right,up,up,right,right,right,collect,left,left,left,left,down,collect,right,up,right,up,collect,left,down,collect,left,down,down,right,collect,right,snap;41;19918

Grid2:
left,left,left,up,up,right,up,up,right,up,right,right,right,down,down,left,down,down,right,collect,left,left,collect,left,left,left,up,up,right,up,right,collect,left,collect,down,collect,left,down,collect,right,up,up,up,snap;73;81817

- Complete by default and the above restrictions made sure its complete.
- Not Optimal since our repeated states checker doesn't take into consideration the cost from root, so we might have a state that exists in the repeated states set but its node cost is different so we consider it a repeated state while the node itself is not.
- Since IDS expands the same nodes over and over while increasing the depth limit the number of expanded nodes is very large.

greedy1

Grid1:
up,collect,down,left,collect,down,down,collect,left,collect,up,collect,up,right,right,right,right,collect,left,left,left,down,down,right,snap;41;61

Grid2:
up,collect,right,right,collect,left,left,left,up,left,left,collect,up,right,collect,up,collect,right,collect,up,left,snap;47;74

- Complete
- Not optimal since it only considers the future cost of the nodes when arranging them in the queue so it will not take into consideration the cost to get to that node which could lead to non-optimal paths
- The number of expanded nodes is very small since it's an informed search, so it tries to go to the goal test as fast as it could, also since we are using an admissible heuristic function that only underestimates the real cost.

greedy2

Grid1:
up,collect,down,left,collect,down,down,collect,left,collect,up,collect,up,right,right,right,right,collect,left,left,left,down,down,right,snap;41;61

Grid2:
up,collect,right,right,collect,up,left,kill,left,left,left,up,collect,up,collect,right,collect,left,kill,down,down,left,collect,up,up,up,right,snap;59;101

- Complete
- Not optimal since it only considers the future cost of the nodes when arranging them in the queue so it will not take into consideration the cost to get to that node which could lead to non-optimal paths.
- The number of expanded nodes is very small since it's an informed search, so it tries to go to the goal test as fast as it could, also since we are using an admissible heuristic function that only underestimates the real cost.

Astar1

Grid1: left, collect, up, right, collect, right, right, down, collect, up, left, left, left, left, down, down, collect, down, collect, right, collect, right, snap; 32;908

Grid2: left, left, up, up, left, collect, right, down, down, right, right, right, right, up, collect, left, left, collect, left, left, up, up, collect, kill, right, up, collect, left, collect, up, snap; 37;5879

- Complete

- (Sub)-Optimal since it arranges the nodes by calculating the cost to get to the node and an estimation that is always an underestimation for the cost to the goal, so it is guaranteed to expand the node with the least total cost first.

- The number of expanded nodes is very small since it's an informed search, but larger than the greedy since it takes into consideration the cost from root so chooses the least damage towards the goal even if it will expand more nodes but the total cost from root till goal is optimal.

Astar2

Grid1:
left, collect, up, right, collect, right, right, down, collect, up, left, left, left, left, down, down, collect, down, collect, right, collect, right, snap; 32;686

Grid2:
left,left,up,up,up,kill,collect,down,down,right,down,right,right,right,up,collect,down,left,left,left, left,up,up,left,collect,up,kill,right,down,down,right,right,collect,kill,left,up,up,up,collect,left,colle ct,up,snap;37;5267

- Complete
- (Sub)-Optimal since it arranges the nodes by calculating the cost to get to the node and an estimation that is always an underestimation for the cost to the goal, so it is guaranteed to expand the node with the least total cost first.
- The number of expanded nodes is very small since it's an informed search, but larger than the greedy since it takes into consideration the cost from root so chooses the least damage towards the goal even if it will expand more nodes but the total cost from root till goal is optimal

Back to the Endgame problem, we have an ***Endgame*** class that extends the ***GenericSearch*** class to apply the search specific to the endgame problem. The class consists of the following attributes: *ironManPosition* representing Iron Man's current positon*, thanosPosition* representing Thanos' position*, stones* hashset that contains the stones positions*, warriors* hashset that contains the warriors positions, *width* and *height* that represent the dimensions of the grid. It also inherits the *initialState* and *actions* from the GenericSearch class. The Endgame class implements the abstract methods in the GenericSearch problem which are:

- *goalTest:* Our goalTest checks if Iron Man is in the same cell as Thanos, collected all stones and the final damage after going to Thanos cell and getting the inflicted damage on him from being in that cell is < 100.
- *transition:* This method takes as input a ***StateCost*** object and an *action* to be performed. The ***StateCost*** class is a class that consists of the state of the node and the cost from root for the node. We created this class to be able to pass the cost from root to the transition method and accumulate on it any added cost then use that in order to create the child node since we can't use the *damage* attribute of the endgame class in order to get the damage from the state in the *generalSearch* method, we created this class in order to be able to do that without affecting the generality of the *GenericSearch* class. Back to the transition method, first it calls three helper methods:

- **warriorsAround*:*** which takes Iron Man's current position and the array of warriors left and see if there are any warriors around ironMan and returns a boolean array of size 4 that has true in the positions that there are warriors present in any adjacent cells with index 0 mapping to up, 1 to down, 2 to right and 3 to left.
- ***thanosAround*:** which takes Iron Man's current position and Thanos' position as inputs and checks if Thanos is an adjacent cell to ironMan and returns a Boolean array of size 5 that has true in the position that Thanos exists in if it is adjacent to ironMan with index 0 mapping to up, 1 to down, 2 to right, 3 to left and 4 to the same cell.
- ***calculateDamage:*** which uses the returned arrays from *warriorsAround* and thanosAround as well as the current state in order to calculate the total damage inflicted on ironMan in his position before performing any actions. It returns an array of int with 2 positions. The first one contains 5 if Thanos is an adjacent cell to Iron Man and the second one contains the sum of adjacent warriors to ironMan.

- After that there is a switch statement that checks the action that is takes as input which can be any of the following:
  - ***Note***: For any action if it's invalid or if the total damage after performing the action exceeds 99 then null is returned.
  - *Kill:* We use the *warriorsAround* array and we loop on it and any warrior present is removed from a copy of the *warriorsLeft* hashset. Damage is calculated as (2*NumberOfWarriorsKilled) and added to the Thanos damage returned from the *calculateDamage* method and also added to the damage inflicted on ironMan to get to this node. Then a new StateCost is returned with the same position, same stones hashset, *new warriorsLeft hashset, new damage.*
  - *Collect:* We check in the *stonesLeft* hashset if there is a stone in the same position as ironMan. If yes, then we remove the stone from a copy of the *stones* hashset and add 3 to the damage of ironMan till this node + damage from the *calculateDamage* method. A new StateCost is returned with the same position, new stonesLeft hashset, new damage.
  - *Up:* We check if the cell above ironMan isn't a border cell, doesn't contain a warrior, if it contains Thanos we make sure Iron Man's state is valid for snapping. If all conditions are satisfied then the move is valid and we return a new *StateCost* with the newPosition, same stones, same warriors, newDamage that is the sum of the damage inflicted on ironMan to get to this node + damage from *calculateDamage****.***
  - *Down/Left/Right:* same idea as *up.* Each has the new position corresponding to the move it's making.

- *generateSolution:* takes a node and generates the actions that are required to reach the node from the root.
- *visualizeGrid*: this method takes the grid and the solution. First it prints the initial grid with the stones 'S', warriors 'W', ironMan 'I' and Thanos 'T' in their given initial positions and then and prints the new grid that reflects the change from each action performed. Every empty cell is represented by a '.' .It calls a helper method ***printGrid*** that simply take a 2D array and prints it.