RISK Simulation

By

Amr Mohamed Nasr (47)
Michael Raafat (57)
Mohammed Deifallah (59)

OVERVIEW

In this assignment you are asked to implement a simple environment simulator that generates instances of a search problem, runs agent programs, and evaluates their performance according to a simple performance measure. The search problem we will use is a simplified and abstract version of the board game **RISK**. Also, it’s required to implement different types of **AI** and non-**AI** agents.

ALGORITHMS USED

As mentioned above, it’s required to implement different type of agents. Below is aa detailed discussion for each one of them:

1) HUMAN AGENT

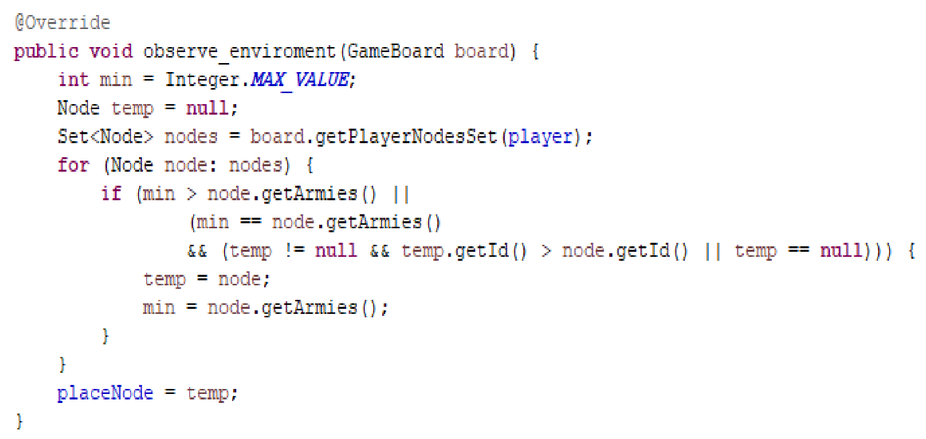
It has nothing to do because the application only receives the input from the user.

2) PASSIVE AGENT

As mentioned in the problem statement, it’s a non-**AI** agent, which places all its bonus armies on the vertex with the fewest number of armies, and never attack.

I. ALGORITHM

As shown in the figure below, the algorithm searches for the node with least number of armies, breaking tie by lower ID.



```
@Override
public void observe_enviroment(GameBoard board) {
    int min = Integer.MAX_VALUE;
    Node temp = null;
    Set<Node> nodes = board.getPlayerNodesSet(player);
    for (Node node: nodes) {
        if (min > node.getArmies() ||
                (min == node.getArmies()
                && (temp != null && temp.getId() > node.getId() || temp == null))) {
            temp = node;
            min = node.getArmies();
        }
    }
    placeNode = temp;
}
```

II. DATA STRUCTURE

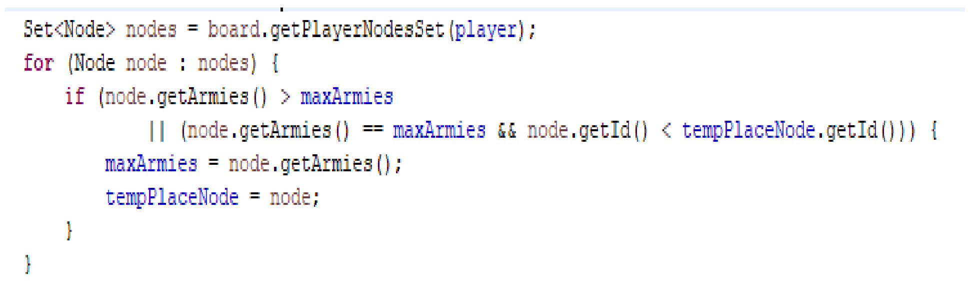
It only uses a **HashSet** for the current player’s territories.

3) AGGRESSIVE AGENT

That non-**AI** *agent* places all its bonus armies on the strongest vertex. It wants to cause the maximum damage to the opponent player as well.

I. ALGORITHM

For placement stage, it searches for the node with most armies to place its bonus armies on it, breaking ties by lower ID.



```
Set<Node> nodes = board.getPlayerNodesSet(player);
for (Node node : nodes) {
    if (node.getArmies() > maxArmies
            || (node.getArmies() == maxArmies && node.getId() < tempPlaceNode.getId())) {
        maxArmies = node.getArmies();
        tempPlaceNode = node;
    }
}
```

For attack stage, it attacks the vertex that can cause the maximum damage for the opponent. The maximum damage here means attacking the country with most possible number of armies to be attacked in a complete partition, so the opponent loses both armies and the bonus of this partition.

II. DATA STRUCTURE

This algorithm uses an array to store the edges of end points belong to different players, an array of sets of partitions and two sets for the players' territories.

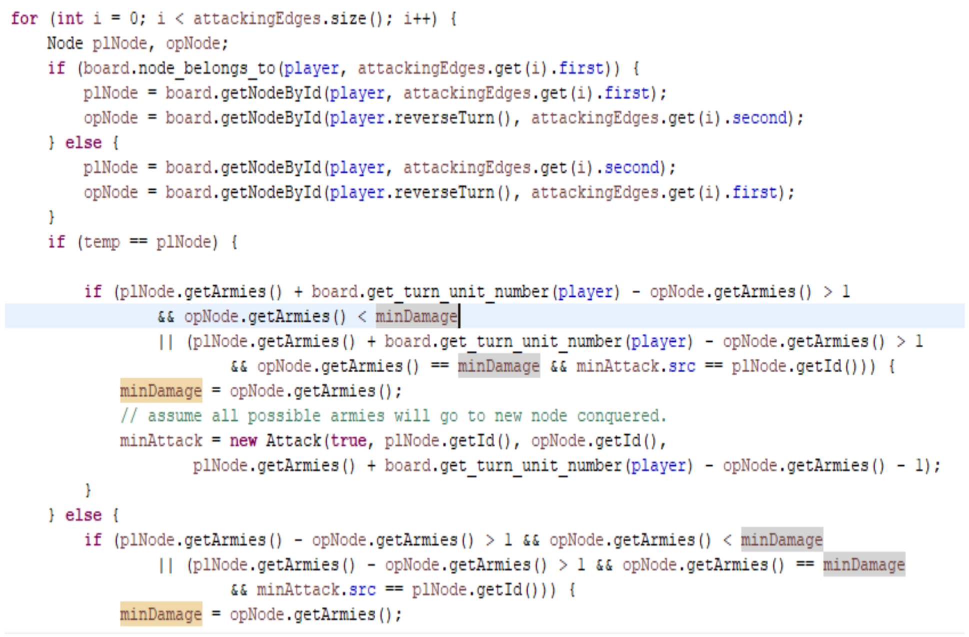
4) PACIFIST AGENT

That non-**AI** agent imitates the previously-mentioned passive one for the placement stage. On the other hand, it takes the safest side for attack stage, and attacks the vertex with the least number of armies.

I. ALGORITHM

For placement stage, the same as the passive agent.

For attack stage, it attacks the vertex that causes the minimum damage for him.



```
for (int i = 0; i < attackingEdges.size(); i++) {
    Node plNode, opNode;
    if (board.node_belongs_to(player, attackingEdges.get(i).first)) {
        plNode = board.getNodeById(player, attackingEdges.get(i).first);
        opNode = board.getNodeById(player.reverseTurn(), attackingEdges.get(i).second);
    } else {
        plNode = board.getNodeById(player, attackingEdges.get(i).second);
        opNode = board.getNodeById(player.reverseTurn(), attackingEdges.get(i).first);
    }
    if (temp == plNode) {

        if (plNode.getArmies() + board.get_turn_unit_number(player) - opNode.getArmies() > 1
                && opNode.getArmies() < minDamage
                || (plNode.getArmies() + board.get_turn_unit_number(player) - opNode.getArmies() > 1
                        && opNode.getArmies() == minDamage && minAttack.src == plNode.getId())) {
            minDamage = opNode.getArmies();
            // assume all possible armies will go to new node conquered.
            minAttack = new Attack(true, plNode.getId(), opNode.getId(),
                    plNode.getArmies() + board.get_turn_unit_number(player) - opNode.getArmies() - 1);
        }
    } else {
        if (plNode.getArmies() - opNode.getArmies() > 1 && opNode.getArmies() < minDamage
                || (plNode.getArmies() - opNode.getArmies() > 1 && opNode.getArmies() == minDamage
                        && minAttack.src == plNode.getId())) {
            minDamage = opNode.getArmies();
```

II. DATA STRUCTURE

The same as the aggressive agent.

5) GREEDY AGENT

That **AI** agent picks the move with best immediate heuristic value.

I. ALGORITHM

It applies the greedy search algorithm, such that it searches for the possible placement moves, then searches for the possible attacks. The goal here is to find the move with the lease heuristic cost.

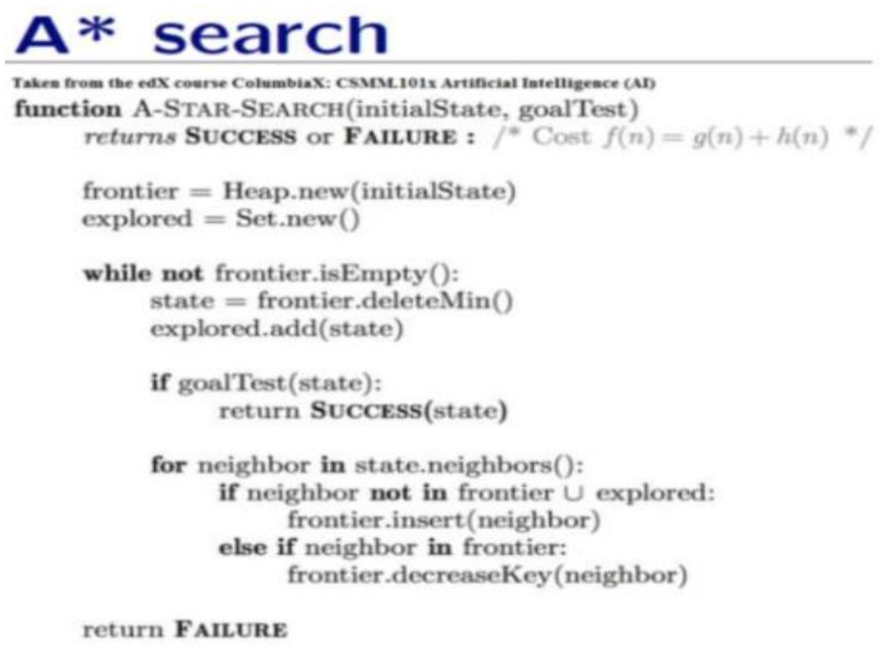
DATA STRUCTURE

A collection of sets and lists to implement the best search algorithm.

6) A* AGENT

That **AI** agent applies A* search algorithm to decide the next move.

I. ALGORITHM



A* search

Taken from the edX course ColumbiaX: CSMM.101x Artificial Intelligence (AI)

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE : /* Cost $f(n) = g(n) + h(n)$ */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

DATA STRUCTURE

A collection of HashSets, HashMaps and lists to implement the A* search algorithm.

7) RTA* AGENT

That **AI** agent applies A* search algorithm to decide the next move. However, it only works for up to 4 search levels not to result in long response time.

I. ALGORITHM

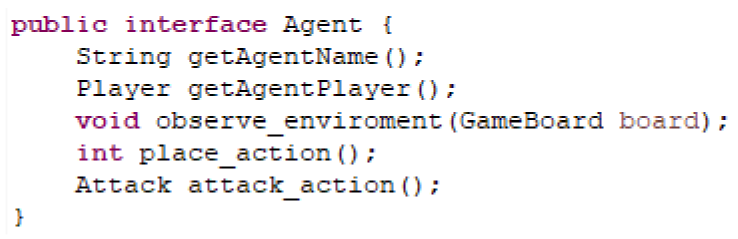
The same as A*.

DATA STRUCTURE

The same as A*.

CODE ORGANIZATION

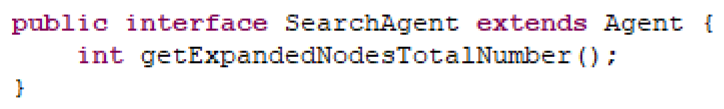
1) AGENT INTERFACE



```java
public interface Agent {
    String getAgentName();
    Player getAgentPlayer();
    void observe_enviroment(GameBoard board);
    int place_action();
    Attack attack_action();
}
```

All agents have to implement this interface to provide the minimal functionality.

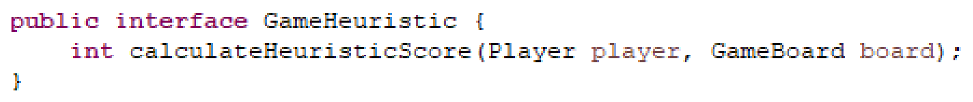
2) SEARCH AGENT INTERFACE



```java
public interface SearchAgent extends Agent {
    int getExpandedNodesTotalNumber();
}
```

AI agents also have to add this function to provide information about the run time.

3) GAME HEURISTIC INTERFACE



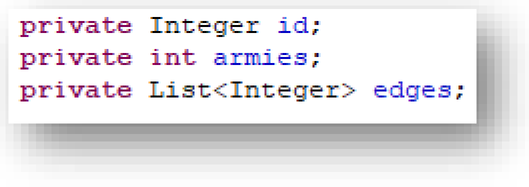
```java
public interface GameHeuristic {
    int calculateHeuristicScore(Player player, GameBoard board);
}
```

The heuristic function for the player, used to implement **AI** agents, follows this formula:

$$h(n) = 1 * \#can\ be\ attacked + 2 * \#can't\ be\ attacked + 1 * \#not\ adjacent$$

4) NODE

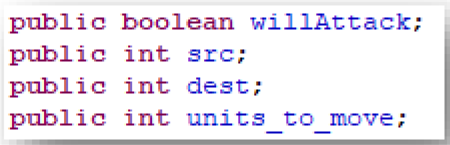
The internal representation for each territory. It has the following attributes:



```java
private Integer id;
private int armies;
private List<Integer> edges;
```

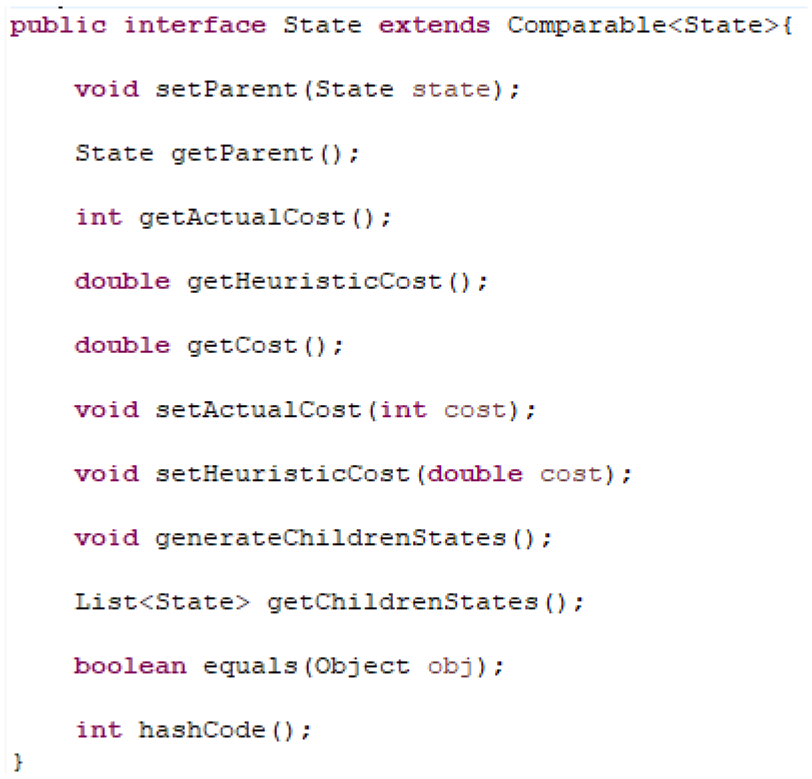
5) ATTACK

The internal representation for each attack taken by any node. It keeps information about the source and destination territories, in addition to the number of armies to be moved after the attack.



```
public boolean willAttack;
public int src;
public int dest;
public int units_to_move;
```

6) STATE INTERFACE



```
public interface State extends Comparable<State>{

    void setParent(State state);

    State getParent();

    int getActualCost();

    double getHeuristicCost();

    double getCost();

    void setActualCost(int cost);

    void setHeuristicCost(double cost);

    void generateChildrenStates();

    List<State> getChildrenStates();

    boolean equals(Object obj);

    int hashCode();
}
```

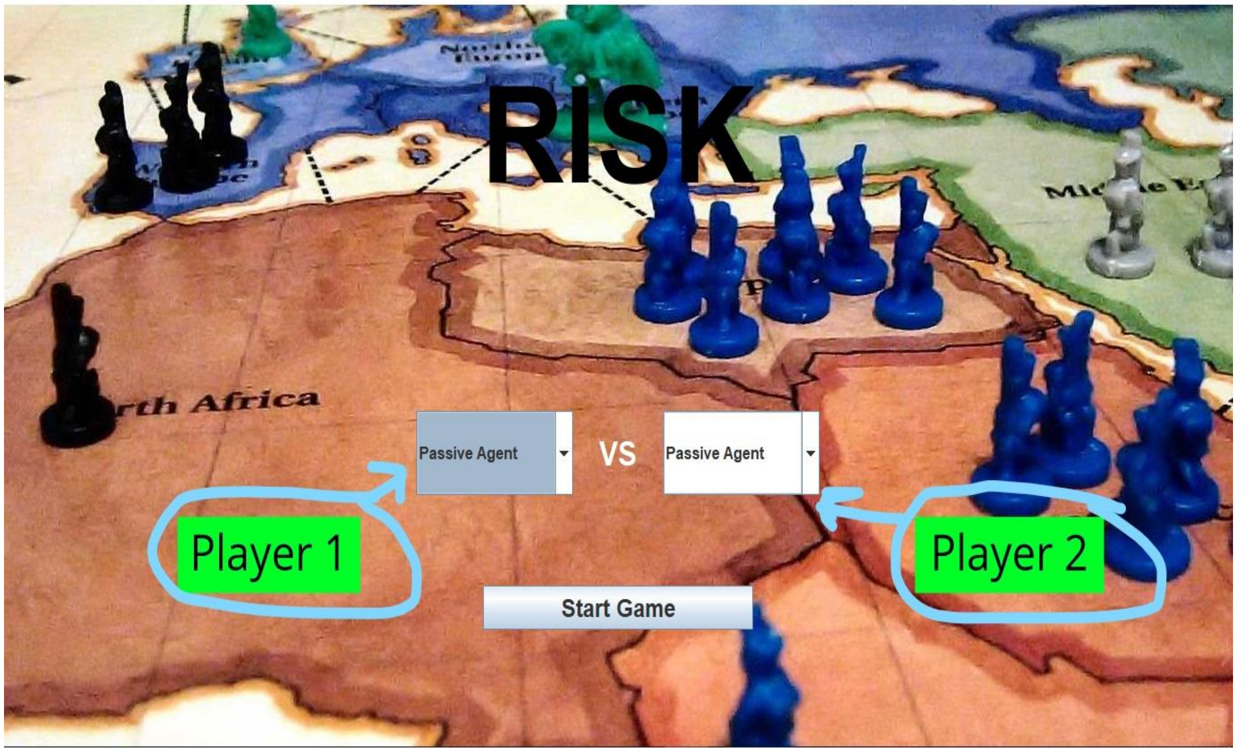
GUI APPLICATION

We provide a 2-player GUI game, implemented in **Java Swing**, that enables the user(s) to play the **RISK** game for different purposes.

1- 2 human players.
2- One player plays against the computer.
3- Scientific and educational issues to learn from computer agents or measure the performance of each of them.

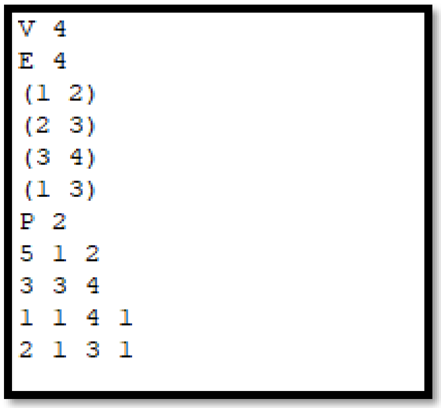
In addition, the graph representation is implemented using GraphStream external JAR.

Here's a screenshot of the game intro:



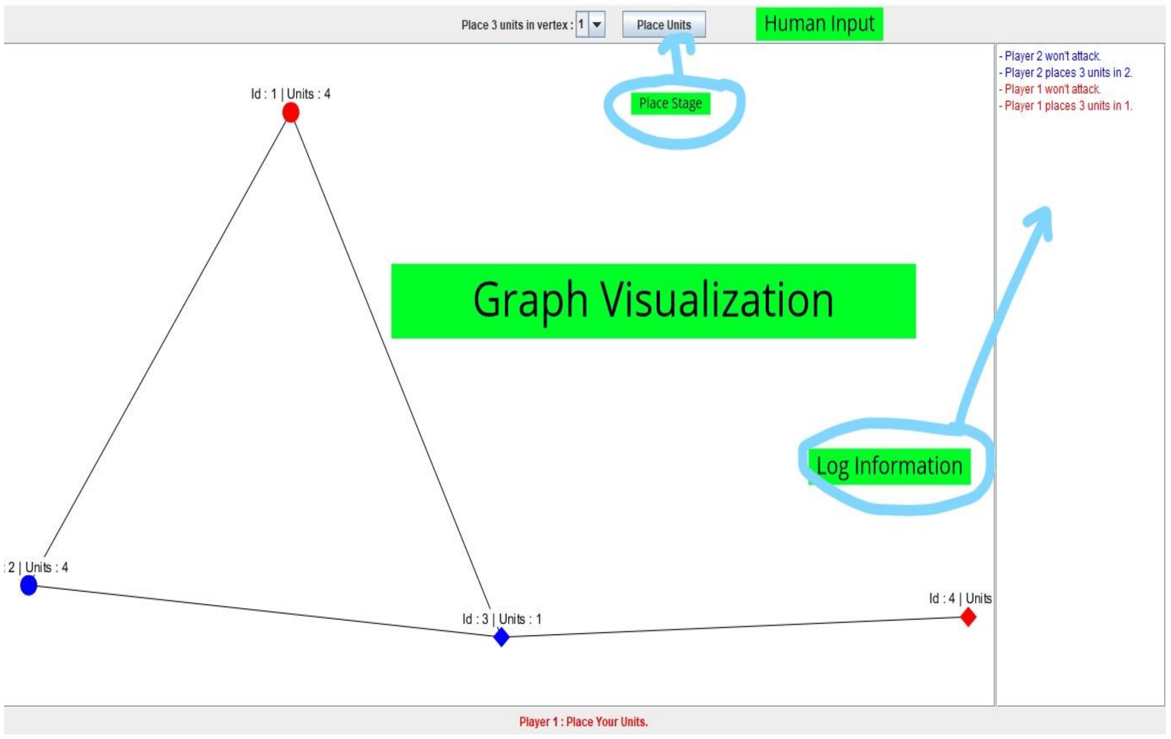
SAMPLE RUNS

The text file used for that sample has the following specifications:



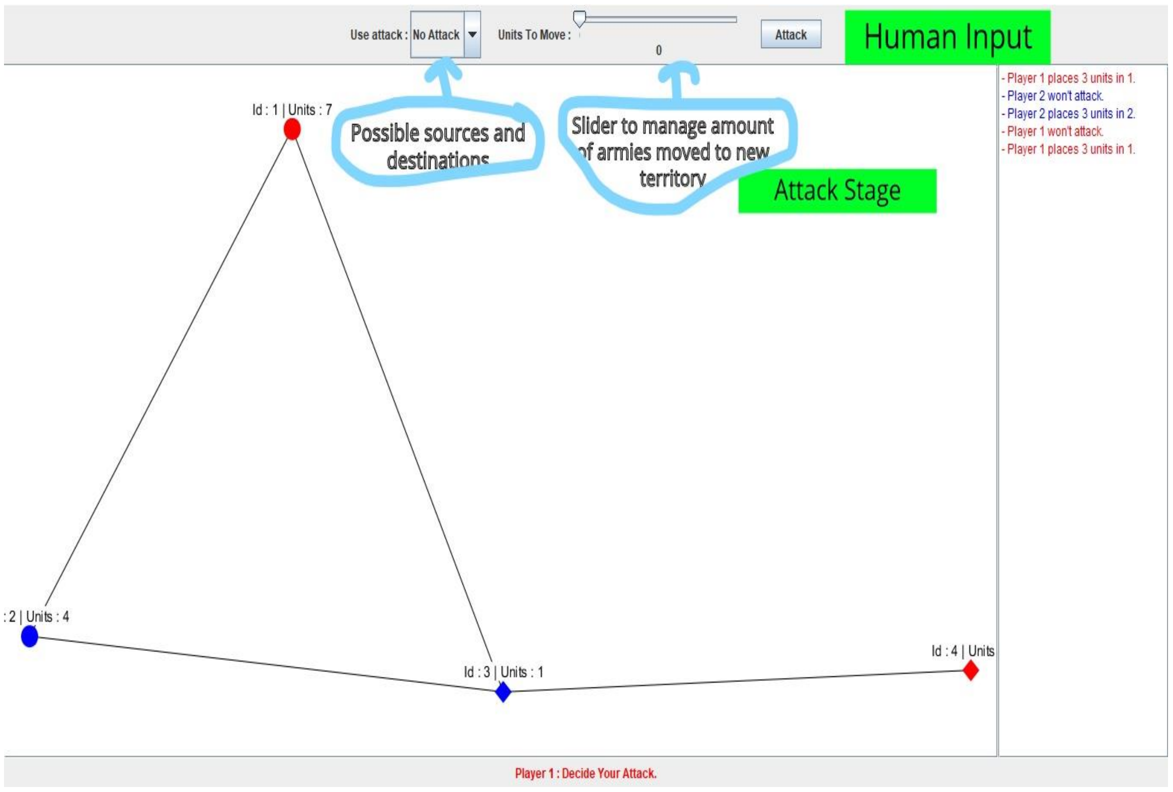
```
V 4
E 4
(1 2)
(2 3)
(3 4)
(1 3)
P 2
5 1 2
3 3 4
1 1 4 1
2 1 3 1
```

And that's the GUI game:



- Player 2 won't attack.
- Player 2 places 3 units in 2.
- Player 1 won't attack.
- Player 1 places 3 units in 1.

Player 1 : Place Your Units.



- Player 1 places 3 units in 1.
- Player 2 won't attack.
- Player 2 places 3 units in 2.
- Player 1 won't attack.
- Player 1 places 3 units in 1.

Player 1 : Decide Your Attack.

ASSUMPTIONS & EXPLANATIONS

- The Initial placement of armies is determined as part of the input, such that the given file format is extended by extra two lines. The first line is for the first player, and the second one is for the other. Each line follows the format shown below: $\text{territory}_1\ \text{armies}_1\ \dots\ territory_n\ armies_n$, where n is the number of territories owned by that player.
- For pacifist and aggressive agents, it’s assumed that the agent leaves only one army in the attacking territory and moves all the remaining armies into the new one.
- **RTA*** uses a maximum search limit of 4 levels.
- **AI** agents are programmed to search against a passive agent.
- The actual cost is the number of turns from the game start till the current one.