L-Università ta' Malta
Faculty of Information & Communication Technology
Department of Computer Information Systems

IAPT Assignment - A Smart Strategist for Risk

Edward Thomas Sciberras (274402L)
B.Sc. (Hons) Information Technology (AI)

Study-unit: **Individual Assigned Practical Task**
Code: **ARI2201**
Supervisor: **Dr Ingrid Vella**

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as “the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

We, the undersigned, declare that the assignment submitted is my / our* work, except where acknowledged and referenced.

We understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected and will be given zero marks.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Edward Thomas Sciberras 

Student Name Signature

ARI2201 A Smart Strategist for Risk

Course Code Title of work submitted

05/06/2022

Date

Contents

1. Introduction

1.1 What is Risk?

Risk is a two to six player strategic board game about diplomacy, combat, and conquest. The conventional version is played on a board with a political map of the world divided into forty-two regions and six continents. Players take turns controlling armies of playing pieces in an attempt to acquire regions from other players, with dice rolls determining the outcome. Throughout the game, players can create and break alliances. The object of the game is to occupy all areas on the board and eliminate the other players in the process. The game might take anywhere from a few hours to many days to complete.

Risk was created by Albert Lamorisse, a French filmmaker, in 1957 and has since become one of the most popular board games in history, generating games like Axis & Allies and Settlers of Catan. It appeals to adults as well as children and families due to its simple rules yet intricate relationships. Hasbro continues to produce it in a variety of editions and variants with popular media themes and rules, including PC software, video games, and mobile applications.

1.2 How can AI be implemented?

In this situation, an artificial intelligence model can be implemented that would learn from playing a plethora of games using different strategies and choices and developing an optimal strategy over time. Since it is a multi-player game and there is an element of luck involved in the winning of battles and territories, there is no method or strategy that is capable of winning every single game that it plays.

With this being said, there are machine learning methods that fall under the category of reinforcement learning which can aid a model in learning after processing a large number of games and data that would be close to impossible for a human to play all of them and gather all the information needed.

Reinforcement learning is all about taking the right steps to maximize your benefit in a given circumstance. It is used by a variety of software and computers to determine the best feasible action or path in a given scenario. Reinforcement learning differs from supervised learning in that supervised learning includes the answer key, allowing the model to be trained with the correct answer, whereas reinforcement learning does not include an

answer and instead relies on the reinforcement model to decide what to do to complete the task. It is obligated to learn from its experience in the absence of a training dataset.

1.3 Scope & Approach

The scope of this project is to investigate the learning capabilities of certain reinforcement learning methods concerning a classic board game like the one in question. From the perspective of AI, Risk is an intriguing game to teach as it is not like other board games such as chess or checkers, in which a player makes one move in a turn, in Risk, each player's turn involves three distinct stages, these being: receiving & placing troops, attacking and fortifying.

This project was limited to only the learning of the game in a four-player game. In which one player will be the learning model and the other three players will be playing each in their own distinct way. This shall be further explained later on. The learning part of the algorithm will also just be limited to the playing phase of the game and the initialising of the board will be mostly randomised.

2. Literature Review

Recently, techniques similar to the ones mentioned above have been used to handle the same, or a variant of the one in question. For instance, Erik Blomqvist [1] also recreated the Risk environment in code and applied different machine learning algorithms to this problem to see which performs the best. However, his environment was a one vs one situation with the model playing against itself over and over again. The algorithms that he used are the following: reinforcement learning, Monte Carlo tree search, neural networks and zero learning. He concluded that the zero learning approaches, minimally supplemented by human expertise and adjustments, may be utilized to train a neural network such that the network action rules increased the Monte Carlo Tree Search decision making algorithm's playing performance. However, after further iterations of network training, agent performance did not improve.

Another study done by Michael Wolf [2] uses Temporal Difference Learning (TD). This type of learning is a subsection of reinforcement learning and involves looking at the potential steps ahead of it and choosing the one with the highest expected value of each state. The number of steps to look ahead along with the importance to give them can be adjusted to see which configuration would produce the best results.

Manuela Lutolf [3] also takes the approach of using a TD(λ) algorithm that will consider a multitude of features when making the next decision, some of these being: the total number of territories owned, number of unique enemy neighbours, pairs of friendly neighbours and the number of countries in a specific continent. These are not all of the features involved; however, they are the ones that are not noticed easily and would need some thought to be put behind them to implement them and are not surface level.

3. Implementation of Risk Environment

This implementation was done using Python as there is a multitude of easy access libraries that can be used for machine learning algorithms. One of these is OpenAI Gym, which was used to perform the reinforcement learning environment and will be explained further on. In addition, Jupyter Notebook facilitates the easy use of modularity in the code and tests individual pieces of code easily.

3.1 Initialisation of Board

The first step in creating the Risk environment in code was to create classes that would define each component in the game. Six basic classes are used as the building blocks in the entire game. These are the following along with their attributes:

- Territory (index, name, ownedBy, parentContinent, currTroops, connections)
- Continent (index, name, territories, troopsWhenFull, noOfCountries)
- Card (troop, territory)
- Player (index, troopTotal, territories, cards)
- SimpleAgent (index, troopTotal, territories, cards, strategy)
- Board (players, noOfPlayers, terrList, continents, cards)

Moreover, Enums were used for each territory and continent as well as troop type (infantry, cavalry or artillery) to keep track of indexes easier and by using names rather than a number with little to no meaning attached to them.

The territory class uses the index to keep track of each individual territory without having to pass the territory whole class each time for a process, as that would be slow as opposed to just passing a single number. The “ownedBy” attribute is used to hold a record of which player owns that territory at any given moment. The “currTroops” feature was made to keep track of how many troops are currently being held on that territory. “connections” is

an array of other indexes of other territories that have a direct link on the Risk map to the territory in question.

Continent has the data “troopsWhenFull” which is a variable that is keeping the number of troops that a player will receive if they hold all the territories in said continent. The card class just holds the data of what troop type it is and what territory it holds.

The difference between the player and “simpleAgent” class is that the player is the model that will be trained and the “simpleAgent” are the other three players that will be a part of the same game. Finally, the board class holds all the information and all the classes mentioned above as a central data source that can be accessed at any time. The board class is interesting as it does not take any parameters to be called as all the data is gathered from external methods.

```
class Board:
    def __init__(self):
        self.players = initialisePlayers()
        self.noOfPlayers = len(self.players)
        self.terrList = initialiseTerritories()
        self.continents = initialiseContinents(self.terrList)
        self.cards = initialiseCards()
```

Figure 1 - Board class

The initialisation of the players is a simple method as all it does is create a singular player class with no troops or territories and three “simpleAgent” classes, also with no troops or territories, but with the extra “strategy” attribute that will be explained later on.

```
def initialisePlayers():
    player0 = Player(0, 0, [], [])
    player1 = SimpleAgent(1, 0, [], [], random.randint(0, 2))
    player2 = SimpleAgent(2, 0, [], [], random.randint(0, 2))
    player3 = SimpleAgent(3, 0, [], [], random.randint(0, 2))

    playerList = [player0, player1, player2, player3]

    return playerList
```

Figure 2 - Method to initialise players

The territory initialisation was a straightforward implementation as there were no other options as opposed to having to hardcode each territory with its own attributes. The following is an example of the South American territories getting initialised and added to a list to be returned to the board.

```
# south american territories
terrList.append(Territory(TerrEnums.venezuela, "Venezuela", None, ContEnums.southAmerica, 0, [TerrEnums.centralAmerica, TerrEnums.brazil, TerrEnums.peru]))
terrList.append(Territory(TerrEnums.brazil, "Brazil", None, ContEnums.southAmerica, 0, [TerrEnums.venezuela, TerrEnums.peru, TerrEnums.argentina, TerrEnums.northAfrica]))
terrList.append(Territory(TerrEnums.peru, "Peru", None, ContEnums.southAmerica, 0, [TerrEnums.brazil, TerrEnums.venezuela, TerrEnums.argentina]))
terrList.append(Territory(TerrEnums.argentina, "Argentina", None, ContEnums.southAmerica, 0, [TerrEnums.peru, TerrEnums.brazil]))
```

Figure 3 - South American territories getting initialised

Continents were also relatively easy to initialise, all that was needed is to loop through all the territories and for each territory that had its “ownedBy” attribute equal to the index of the current continent in the loop then add it to a list that will eventually be added to the final initialisation of a single continent.

```
def initialiseContinents(terrList):
    contList = []
    for continent in ContEnums:
        tempList = []
        for territory in terrList:
            # if terr continent value is equal to current continent in loop then add
            if territory.parentContinent.value == continent.value:
                tempList.append(territory)

        if continent.value == 0:
            contList.append(Continent(continent, "North America", tempList, 5, len(tempList)))
        elif continent.value == 1:
            contList.append(Continent(continent, "South America", tempList, 2, len(tempList)))
        elif continent.value == 2:
            contList.append(Continent(continent, "Europe", tempList, 5, len(tempList)))
        elif continent.value == 3:
            contList.append(Continent(continent, "Africa", tempList, 3, len(tempList)))
        elif continent.value == 4:
            contList.append(Continent(continent, "Asia", tempList, 7, len(tempList)))
        elif continent.value == 5:
            contList.append(Continent(continent, "Australia", tempList, 2, len(tempList)))

    return contList
```

Figure 4 - Method to initialise the continents

The initialisation of cards was slightly different from the real game rules, in the sense that instead of hard coding each card with its values, the territories were randomised each game. This does not make a difference in the grand scheme of things as each territory has its card either way. There then was a loop which essentially looped forty-two times (number of territories in the game) and if the index was between zero and thirteen then it was an infantry card, if the index was between fourteen and twenty-seven then it would be a cavalry card and the remaining would be artillery cards. This way each troop type would receive fourteen cards equally.

```
def initialiseCards():
    cardList = []
    troopTypeCounter = 0
    territories = list(range(0, len(TerrEnums)))
    random.shuffle(territories)
    while territories:
        # random territory each time
        randomTerrIndex = territories.pop()
        if troopTypeCounter <= 13: # for infantry
            cardList.append(Card(TroopEnums.infantry.value, TerrEnums(randomTerrIndex).value))
        elif troopTypeCounter <= 27: # for cavelry
            cardList.append(Card(TroopEnums.cavalry.value, TerrEnums(randomTerrIndex).value))
        elif troopTypeCounter <= 41: # for artillery
            cardList.append(Card(TroopEnums.artillery.value, TerrEnums(randomTerrIndex).value))

        troopTypeCounter += 1

    return cardList
```

Figure 5 - Algorithm to make all cards needed for the game

After all the important items were initialised and added to the board, the playable game setup can begin. This starts by rolling a dice for each player to determine the player order for the rest of the game. The algorithm will continue rolling four dice until the numbers are all different to avoid any confusion and unnecessary code, as shown in the following snippet.

```
startingRolls = []
# making sure that all values are different to avoid confusion with starting order
while len(list(set(startingRolls))) != self.board.noOfPlayers:
    startingRolls = rollDice(self.board.noOfPlayers)
```

Figure 6 - Code to get four distinct numbers from dice rolls

Once all the numbers were different, the logic to find the order was to sort the values by their order in descending order and save their index, which in this case, is the player that the number is representing. Then a new list called “turnOrder” was made and the index of the player is added to said list. The result of this is the order of players that the game will proceed with, starting with the player that rolled the highest number and ending with the player that rolled the lowest number.

```
# sorting rolls with their values and getting the turn order
sortedRolls = sorted(((value, index) for index, value in enumerate(startingRolls)), reverse = True)
self.turnOrder = []
for i in range(len(sortedRolls)):
    self.turnOrder.append(sortedRolls[i][1])
```

Figure 7 - Code to determine the order of the players according to their dice roll

The next step in the process is to randomise the giving out of all the territories to each of the players. Naturally, the player that rolled the highest will start getting the territories and the list of territories will keep getting randomised to the player that has his current turn, as seen in the snippet below. The territories were given out randomly as it makes the players have to strategies more as opposed to picking their own as they will have to work with what they have. Naturally, since there are forty-two territories and four players, the first two players will have eleven territories starting whilst the latter two will have 10 each.

```
# spreading territories between players
tempTerrList = self.board.terrList.copy()
i = 0
while len(tempTerrList) != 0:
    self.currPlayerTurn = self.turnOrder[i % self.board.noOfPlayers] # getting the current player turn from rolls
    randomTerrIndex = random.randint(0, len(tempTerrList) - 1) # getting a random territory index
    currTerr = tempTerrList.pop(randomTerrIndex) # getting actual territory
    self.board.players[self.currPlayerTurn].territories.append(currTerr) # adding territory to players

    i += 1
```

Figure 8 - Giving out territories randomly

After the territories have been divided as equally as possible, each player has to set at least one troop on each. Once this is done the left-over troops would be spread out randomly between all their territories, this was done to keep things more straightforward.

```
for player in self.board.players:
    # putting at least 1 troop on every territory
    for terr in player.territories:
        terr.ownedBy = player.index
        terr.currTroops = 1
        player.troopTotal += 1

    # randomizing the places of the leftover troops
    leftoverTroops = STARTING_TROOPS - player.troopTotal
    for i in range(leftoverTroops):
        randomTerrIndex = random.randint(0, len(player.territories) - 1)
        player.territories[randomTerrIndex].currTroops += 1
        player.troopTotal += 1
```

Figure 9 - Code that is placing troops on starting territories of each player

Once these last steps of initialisation are taken care of, the game can start with each player doing their turns that are divided into three different phases; receiving and placing troops, attacking and fortifying.

3.2 Receiving & Placing Troops

Receiving and placing troops is the first part of a player's turn and involves acquiring several troops through different potential sources such as the number of held territories and trading in cards, however, there are also other sources.

The first course of income troops can be attributed to the number of territories that the player has. The rule is that the player will receive the number of territories it is currently holding, divided by three and ignoring any decimal. However, if the result of this small equation is less than 3, then it will be set to three automatically as per the game rules. The code to achieve is shown below.

```
# using int to ignore decimal
troopsRecieved = int(noOfTerrs / 3)

# can not recieve less than 3 troops in a turn
if troopsRecieved < 3:
    troopsRecieved = 3
```

Figure 10 - Snippet to receive troops depending on territory count

The second fount for troops to be acquired would be having a continent bonus, which is essentially the number of troops that a player would be presented if they hold an entire continent with no interruptions. Different continents have a different number of troops given according to their size and the number of territories that the continent holds. To check if a player owns a continent, the following approach was taken, loop through each of the continents and use the in-built function in python of sets to check whether all the territories of a continent are a subset of the list of the territories that the player owns, as depicted in the small algorithm below.

The final source of potential troops is to trade in cards. In this implementation, it was decided to automatically trade in cards when it is possible. In addition, instead of the traditional weights of trading in cards with the increasing value over time, a more straightforward approach. This approach is that if a player has three cards of the same troop type (infantry, cavalry or artillery) they can be traded in for varying values. Infantry would receive four, cavalry would receive six and artillery eight. However, the player could also trade in one of each troop type, which would yield ten troops.

The algorithm will always prioritise a trade-in set that would make the player receive the most troops as possible. Below is an example of checking for the set of all three troop types at once.

```
# checking for one of team type since highest value
if cardTypeList.count(0) >= 1 and cardTypeList.count(1) >= 1 and cardTypeList.count(2) >= 1:
    infantryCardIndex = cardTypeList.index(0)
    cardTypeList.pop(infantryCardIndex)
    poppedCard = cards.pop(infantryCardIndex)
    tradedTroops += checkCardTerritory(poppedCard, player)

    cavalryCardIndex = cardTypeList.index(1)
    cardTypeList.pop(cavalryCardIndex)
    poppedCard = cards.pop(cavalryCardIndex)
    tradedTroops += checkCardTerritory(poppedCard, player)

    artilleryCardIndex = cardTypeList.index(2)
    cardTypeList.pop(artilleryCardIndex)
    poppedCard = cards.pop(artilleryCardIndex)
    tradedTroops += checkCardTerritory(poppedCard, player)

    tradedTroops += 10
    return tradedTroops
```

Figure 11 - Checking for trade in set

The decisions made on how to split the troops between the players' territories will be explained later on in the report.

3.3 Attacking

The second portion of a turn is the attacking phase, which is arguable the most important phase, as it is the part of a turn in which a player can gain more territories and thus be closer to winning. In this implementation, it was decided to use blitz battling. Once a battle is chosen from the attacker, blitz battling will commence, which essentially is the repetition of the chosen battle until the attacker has taken over the territory or the defender has successfully fended off the attacker until they have no more troops left to use. This is common practice in simulations of Risk [1] as it is easier to implement and will speed up the game greatly.

The way that the attacking works starts with a loop over every territory that the player owns and filters out the territories with just one troop on them as due to the rules of Risk, a player can not attack another territory with just one troop. After the filter is done each of the territories with two troops or more are then also looped to get a list of their connections to other territories. Another filter is done to make sure that the connection in question is not a territory that the player also owns, as it does not make sense to attack

yourself. One more feature that was implemented was the detection of whether a defending play is a small player or not. In the sense that, in a real game, a player would be more likely to pursue attacking a player if that player does not have a lot of territories left in the game, as if they are out of the game then it is one less player to worry about. Therefore, the “smallPlayer” flag was added to check if the player in the possible attack is a player with less than ten territories, making them more lucrative to attack.

```
# smaller nations should be targetted
if defendingPlayerTerrs >= 10:
    smallPlayer = False
else:
    smallPlayer = True

for terr2 in playerTerrList:
    if terr2.index == connection:
        isOwnTerr = True
```

Figure 12 - Implementation of the "smallPlayer" flag

Once all these filters and checks are passed through two lists are filled up, “possibleAttacks” and “troopCount”. The former is simply a list of tuples with the format of (attacking territory index, defending territory index, small player flag). This keeps track of the possible attacks that a player can make in that turn. The latter is almost identical besides the fact that instead of the indexes of the territories, there is the number of troops on the territory.

```
# making sure that you can't attack your own terr
if isOwnTerr == False:
    # using a tuple to show which territory to attack from
    possibleAttacks.append((terr.index.value, connection.value, smallPlayer))
    # mirror array with how many troops there are for each territory above
    troopCount.append((terr.currTroops, board.terrList[connection.value].currTroops, smallPlayer))
```

Figure 13 - How the two tuples get filled in the loop

The attacks are then split into the ones that have the “smallPlayers” flag being true and the others having this flag as false. If there is a possible attack on a player that has less than ten territories, then the player will take that one, if not a normal attack will be performed. The algorithm will decide which attack to do if there are multiple attacks by getting the difference between the troops on the attacking territory and the number of troops on the

defending territory. The chosen attack will be the one that has the biggest difference between these two numbers.

```
# choose to attack the territory that has the biggest discrepiancy
troopDifference = []
for possAttack in troopCount:
    troopDifference.append(possAttack[0] - possAttack[1])

maxTroopDifference = max(troopDifference)
chosenBattleIndex = troopDifference.index(maxTroopDifference)
randomBattle = random.randint(0, len(possibleAttacks) - 1)
chosenBattle = possibleAttacks[chosenBattleIndex]

blitzBattle(chosenBattle, board, player)
```

Figure 14 - Choosing attack with biggest discrepancy between troops

The actual attacking process is like the real game except it is blitz (as explained previously). A small change from the real game is that a player will always attack and defend with the greatest number of troops possible. This was done to make the decisions easier. After this, it is the same as vanilla Risk and the players will roll dice according to how many troops they put up and the amount of dice the attacker has higher will remove troops from the defending territory and vice versa. However, if there is a draw then the defender will have the edge. This will keep on going until the defence has no more troops or the attackers have an insufficient number of troops remaining to attack.

If the attacking player wins, then the ownership of the territory switches hands and the attacker gets another card for a possible trade-in, in their next turn.

```
# if attacking player wins 1 terr in round then recieve card
endingTerrCount = len(player.territories)
if endingTerrCount > startingTerrCount and len(board.cards) != 0:
    randomCardIndex = random.randint(0, len(board.cards) - 1)
    player.cards.append(board.cards.pop(randomCardIndex))
```

Figure 15 - If player wins a territory in a turn, then receive a card

3.4 Fortifying

The bulk of fortifying will be explained later on, however, to check if there is a link between two nodes that the player might want to move troops between, a breadth-first search was used. Using the territories at nodes and the connections that they have as edges. The "defaultdict" library from the collections set was used to perform this task to use the indexing capabilities of a normal "dict" but with an easier syntax point of view.

```
# checks if there is a route between terrs that the player wants to move troops between
def isReachable(player, a, b):
    edgeDict = defaultdict(list)
    for terr in player.territories:
        for connection in terr.connections:
            edgeDict[terr.index.value].append(connection.value)

    noOfTerrs = len(edgeDict)
    visited = [False] * (42)
    queue = []

    queue.append(a.index.value)
    visited[a.index.value] = True

    while queue:
        n = queue.pop(0)
        if n == b.index.value:
            return True

        for i in edgeDict[n]:
            if visited[i] == False:
                queue.append(i)
                visited[i] = True

    return False
```

Figure 16 - Code to check if two territories have a clear link between them

4. Implementation of AI

4.1 OpenAI Gym

Gym is a Python library for constructing and comparing reinforcement learning algorithms. It provides a standard API for communicating between learning algorithms and environments, as well as a set of environments that are consistent with that API. Gym's API has become the industry standard for achieving this.

Gym is usually used for their in-built and ready to go games that can be imported with a simple line of code. However, it also supports custom games and environments but uses their API. This starts with the declaration and initialization of the custom environment that

will inherit from the abstract class “gym.Env”. This will house all the important elements that are needed for this library to work.

```
# openAI gym environment is extended
class RiskEnv(Env):
```

Figure 17 - Creating new environment that will inherit from OpenAI Gym

The first compulsory method that is needed is the “__init__” method. This is the first method that will be called when the whole environment is called, essentially, in this case, it will set up the Risk environment and have it ready for the players to play. Moreover, this method holds two important variables that are needed for this API; “observation_space” and “action_space”. The action space is the number of possible actions that a player can take during a turn, in this case, it is three. The observation space is what we can observe in each state that will be used to make a decision.

Since this part is the initialisation stage of the game, all the starting methods and applications were performed here, these being; deciding turn order, spreading territories to the players and spreading the troops for each player within their territories.

```
# init the game when called
def __init__(self):
    # shows different strategies that can be used
    self.action_space = Discrete(3)
    # will hold percentage of board owned
    self.observation_space = Box(low = np.array([0]), high = np.array([100]))
```

Figure 18 - __init__ function

The second compulsory function which is necessary is the “step” function. This method represents a single turn or environment change after an action in the environment. In this case, this method will represent one turn for a singular player, as whenever some player will make a move, the environment will change and decisions made differ depending on moves that were just made.

In addition to a move being made in a single run of this function, the rewards are also given out for the reinforcement part of the algorithm. How the rewards were split and decided will be explained later on.

```
if currPlayerTurn == 0:
    receiveAndPlaceTroops(self.board.players[0], self.board, recievingStrat)
    attacking(self.board.players[0], self.board, attackingStrat)
    fortifying(self.board.players[0], self.board, fortifyingStrat)

    if len(self.board.players[0].territories) == 42:
        gameOver = True
```

Figure 19 - Example of player going through their turn

Thirdly, a method called “render” is needed for the API to work. This is simply the drawing of the environments to make the game more visual. This project does not have a visual component so this method was left empty and as a placeholder so that the API would work with no issues.

```
# used by openAI gym when rendering is involved
def render(self):
    # not used since there is no GUI
    pass
```

Figure 20 - Render method being left empty

Finally, the last method that is needed to run the Gym environment is the “reset” method, essentially, this is the same as the “__init__” method that was previously explained except that it resets the environment between games so that it can be played over and over again with no hiccups.

4.2 Learning Model

The way that the machine learning takes place is that there are three modes that the AI can use in a single turn, these being; passive, pacifist and aggressive. The simple agents that the player will play against also have similar settings. The only difference is that the player can choose one setting every turn and different between strategies when needed and the simple agents will get a random one at the beginning of the game and have to stick with that throughout the whole game. This also gives the player different games as they will be playing against different strategies every game.

The difference between the three strategies takes place in every part of a player's turn. Starting with the receiving troops and placing stage. If the player or agent is in a passive

state, then the player will place all troops in the territory with the least troops in an attempt to defend as well as possible and leave no weak spots in the nation. The aggressive strategy will do the opposite of this and place all troops in the territory that has the most troops already, to try and have a strong territory that is worthy of attacking and winning over multiple territories in a single turn. The pacifist strategy is a sweet medium between these two and will simply give out troops to all its territories one by one. This will build a well-rounded nation that can be a Swiss-army knife for defending and/or attacking.

```
if strat == 0: # passive strat
    # always places recieved troops in terr with least troops
    troopCountList = []
    for terr in player.territories:
        troopCountList.append(terr.currTroops)
    leastTroopTerr = min(troopCountList)
    leastTroopIndex = troopCountList.index(leastTroopTerr)

    player.territories[leastTroopIndex].currTroops += troopsRecieved
```

Figure 21 - Passive strategy for placing troops in the beginning of a turn

Regarding attacking, arguable the most important part of a turn, the difference in strategy is that the passive agent will simply not ever attack and keep its troop for defending all the time. The pacifist agent will only attack between one and three times during a turn and the aggressive will attack anywhere between seven and nine times during their turn.

```
if strat == 0: # passive strat
    # does not attack
    return
elif strat == 1: # pacifist strat
    # only attacks 1 to 3 times
    attackingRounds = random.randint(1, 3)
elif strat == 2: # aggressive strat
    # attacks 7 to 9 times
    attackingRounds = random.randint(7, 9)
```

Figure 22 - Different strategies for attacking

Finally, for the fortifying stage, the passive strategy will divide the number of troops of the territory that has the most troops with the territory that has the least troops in a further try to have an all-around defence nation. The pacifist agent will simply just pass this part and the aggressive will get the territory with the most troops and the territory with the second most troops and divide the troops equal between them. It is important to keep in mind that any transitions will only take place if there is a valid link between each territory.

```
# checking if there is valid path
if isReachable(player, maxTroopTerr, max2TroopTerr) == True:
    # moves troops from terr with most troops and 2nd most
    if totalTroops % 2 == 0:
        maxTroopTerr.currTroops = totalTroops / 2
        max2TroopTerr.currTroops = totalTroops / 2
    else:
        maxTroopTerr.currTroops = int(totalTroops / 2) + 1
        max2TroopTerr.currTroops = int(totalTroops / 2)
```

Figure 23 - Aggressive strategy for the fortifying stage

Now that the possible actions were explained, the actual machine learning can be explained. A sequential model was built with three distinct layers, all dense. Two of these had twenty-four nodes and the last with three (to represent the number of actions).

```
def build_model(states, actions):
    # building model with 3 dense layers
    model = Sequential()
    model.add(Dense(24, activation = 'relu', input_shape = states))
    model.add(Dense(24, activation = 'relu'))
    model.add(Dense(actions, activation = 'linear'))

    return model
```

Figure 24 - Model being built

The agent that will perform the actions in the game was developed using a Boltzmann Q Policy which is a soft-maxing Q learning method. There will be a total of five hundred thousand games played for the agent to learn from over and over. The games will be split into batches of ten thousand for easier analysis and calculating rewards.

```python
def build_agent(model, actions):
    policy = BoltzmannQPolicy() # soft-maxing Q learning
    memory = SequentialMemory(limit = 500000, window_length = 1)
    dqn = DQNAgent(model = model, memory = memory, policy = policy, nb_actions = actions, nb_steps_warmup = 1000, target_model_update = 1e-2)

    return dqn
```

Figure 25 - Building agent

The rewards were processed as follows:

- If the player wins, a large positive reward
- If the player loses but is still in the game, a small negative reward
- If the player loses and is knocked out of the game, a large negative reward
- Tiny negative reward every turn to deter from stalling a win
- If the game exceeds a certain number of turns, end game
    - Varying awards depending on the number of territories held at end of the game

```python
# if game lasts more than 200 turns take into account how many territories player had
playerTerrs = len(self.board.players[0].territories)
agent1Terrs = len(self.board.players[1].territories)
agent2Terrs = len(self.board.players[2].territories)
agent3Terrs = len(self.board.players[3].territories)

terrsList = [playerTerrs, agent1Terrs, agent2Terrs, agent3Terrs]
terrsList.sort()

if playerTerrs == terrsList[3]:
    self.reward += 250
elif playerTerrs == terrsList[2]:
    self.reward += 100
elif playerTerrs == terrsList[1]:
    self.reward += 0
elif playerTerrs == terrsList[0]:
    self.reward += -50
done = True
```

Figure 26 - Varying rewards at end of games relative to number of territories held at end of game

5. Results & Discussion

5.1 Results Obtained

The ideal situation would have been that the agent slowly learns over time, depending on the situation and the current stage of the game whether it is better to be passive, pacifist or aggressive. If this was the case then the rewards would slowly go up over time, but not always be perfect, as after all Risk is a game that has an element of luck attached to it that cannot be ignored.

However, the results obtained after running the model five hundred thousand times did not yield an improvement overall. After the model was trained, it played another one hundred games, and from the average reward, it can be seen that there was little to no improvement in the results of the games for this agent, and it is impossible to just chalk this up to chance.

```
Episode 96: reward: -200.000, steps: 200
Episode 97: reward: -250.000, steps: 200
Episode 98: reward: -200.000, steps: 200
Episode 99: reward: -200.000, steps: 200
Episode 100: reward: -250.000, steps: 200
-211.0
```

Figure 27 - Showing last few episodes and average score

5.2 Analysing Results

The results obtained could be attributed to a few variables. One of these is that the agent did not have enough freedom to choose the number of options it needed, such as the number of times it can attack in a turn. In addition, the opponents are not the best representation of a real-life human player of the game. One could also argue that the number of repetitions was not enough for the agent to learn everything, however, five hundred thousand is still several games that a human will likely never play throughout their entire life, let alone the forty-five minutes it took this agent to train.

Another potential reason could be that the reward system with the absolute points given out was skewed and the data is not a good representation of real-life values and rewards. Moreover, the neural network and reinforcement learning methods and classes are chosen might have not been the optimal ones.

6. Conclusion

6.1 Conclusion

In conclusion, although the original goal that was set out was not achieved very well. This project is still a good example of how a Risk environment can be implemented and libraries used for reinforcement learning of a game environment using the OpenAI Gym library. In addition, it also displays the capabilities of computers and artificial intelligence in general. Simply to have the capability to run hundreds of thousands of simulated games of a cult-classic board game in a matter of minutes is a great achievement in itself.

7. Bibliography

[1] Blomqvist, E. (2020). *Playing the Game of Risk.* Stockholm.

[2] Wolf, M. (2005). *An Intelligent Artificial Player for the Game of Risk.* Darmstadt.

[3] Lutolf, M. (2013). *A Learning AI for the game Risk using the TD(λ)-Algorithm.* Basel.