L-Università ta' Malta
Faculty of Information & Communication Technology
Department of Artificial Intelligence

# Individual Assigned Practical Task

Cristina Cutajar* (230802L)

*B.Sc. (Hons) Artificial Intelligence

---

Study-unit: **Individual Assigned Practical Task**

Code: **ARI2201**

Lecturer: **Dr Ingrid Vella and Dr Kristian Guillaumier**

# Table of Contents

Plagiarism Form

## 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).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / 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.

* Delete as appropriate.

(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).

| | |
|---|---|
| Cristina Cutajar |  |
| Student Name | Signature |
| | |
| Student Name | Signature |
| | |
| Student Name | Signature |
| | |
| Student Name | Signature |

| | |
|---|---|
| ARI2201 | A smart strategist for Risk IAPT |
| Course Code | Title of work submitted |

19/05/2022

Date

# APT Introduction

## Overall brief of project

For the Individual Assigned Practical Task project, the 'A smart strategist for Risk' proposal offered by Dr Ingrid Vella and Dr Kristian Guillaumier was chosen. This proposal focuses on Machine Learning where the smart strategist is able to learn and suggest which actions should be taken in order to maximise its chances of achieving the goal.

## Aim and Objective/s

The aim of this project is for a smart strategist to be able to be developed for the game of Risk. Therefore, the game of Risk must be implemented along with the smart strategist for it. The smart strategist should be able to correctly suggest which actions the player should take in order to maximise their winning chances. Hence, the objective of this project is to correctly implement the game of Risk with all its features and rules as well as to develop an agent for it. The implementation of a reinforcement learning algorithm is a must for the agent to effectively learn which actions it should take for each state by interacting with the environment. The agent should be able to choose the best actions that should be taken during gameplay for each player state based on its learnt state action values.

## Summary of Functionality Developed

This project was implemented in Python. The entire game of Risk was implemented along with an agent using the On-Policy Monte Carlo reinforcement learning algorithm to learn and select the best attack and defend actions to increase the winning chances. The full implementation consists of the game being able to have between 2 to 6 players. Each player starts off with the number of troops, according to how many players are playing, as specified in the Risk rules [6]. In order for the game to start, all the players roll a dice and the player who rolls the highest dice starts first. The players will take turns to play starting from the initial player and moving clockwise. Then, each player, according to the order, will place 1 troop to claim a territory and when all territories are claimed, they will place 1 troop in one of their own territories until all initial troops are placed. The placement of the troops to claim territories was implemented to happen in a way where territories are randomly chosen from the same continent if the player already owns territories. Otherwise, if the player does not own any territories or there are no unclaimed territories left in the same continent, a random unclaimed territory is chosen. After all territories have been claimed, the rest of the troops are randomly placed at territories with adjacent opponent territories.

According to the player order, the players will then each play a 3-step turn until one of the players manages to win the game by claiming all territories.

The first step of the 3-step turn consists of calculating the number of troops that the player shall receive. This is done by dividing the sum of the player's claimed territories by 3 as well as adding any continent bonus if they own entire continents. Apart from this, the player can get more troops if they return a full set of cards. According to the total set of cards returned, the

player will get a different amount of troops. In the implementation, if a full set of cards is found, the set of cards will be returned immediately. Also, if the player has 5 or 6 cards, some cards will be randomly removed, as according to the rules. When returning a full set of cards, if the player owns any of the territories that are marked on the returned cards, then 2 troops will be added in the particular territory/territories. As per the rules of Risk [6], no matter how many territories a player owns, they will always receive 3 troops.

The second step of the 3-step turn consists of the attack phase. The algorithm loops through the player's owned territories and for each territory, it will check whether it has at least 2 troops and if it has adjacent opponent territories. If the territory contains at least 2 troops and an adjacent opponent territory is found, then the agent will take into consideration the number of troops on both territories and choose whether or not it should attack and with how many dice between 1 and 3. The number of dice has to always be at least 1 less than the number of troops in the territory [6]. The same territory can be attacked more than once and there is no limit for the number of attacks performed per turn [6]. If the player decides to attack, the defender chooses whether to defend with 1 or 2 dice. The agent will make use of another Q table in order to decide whether to roll 1 or 2 defender dice based on how many dice the attacker is rolling and how many troops are in the defender territory. The defender can only attack with 2 dice if they have at least 2 troops in their territory. If the attacker wins a territory, n number of troops should be placed in the new territory where n is the number of dice rolled by the attacker. If the attacker manages to win at least 1 territory in their turn, they draw one card from the shuffled deck. [6]

The third and final step of the 3-step turn consists of the move phase. For each territory owned by the player, if the territory does not have any adjacent opponent territories, all troops except for one will be moved to random adjacent territories owned by the same player which have adjacent opponent territories. 1 troop will always be left in a territory with no adjacent opponent territory.

For the reinforcement learning algorithm, On-Policy Monte Carlo with no exploring starts was implemented to choose the best attack and defend options. On-Policy Monte Carlo either selects a random action with probability $\epsilon$ or greedily with probability $1 - \epsilon$ and epsilon was set to be 1-k where k is the number of episodes performed so far, including the current episode. The reward given to the attacker state-action pair was calculated to be the number of troops that the attacker defeated from the defender minus the number of troops that the attacker lost in battle. Meanwhile, the reward given to the defender state-action pair was calculated to be the number of troops that the attacker lost in battle minus the number of troops that the attacker defeated from the defender. Therefore, for example, if the attacker defeated 2 troops and lost 0 troops, the attacker reward would be 2 and the defender reward would be -2. Another example would be that if the attacker defeated 1 troop and lost 1 troop, a reward of 0 will be given to both the attacker and defender.

# Research

Prior to starting the implementation, research was performed to gain information on how to go about implementing the game of Risk. As well as which reinforcement learning algorithm would be best suited for the smart strategist to play the game and which programming language would be best to use. Research was also performed on previous attempts of developing a smart strategist for the game of Risk to get a better understanding of how it should be implemented, what the challenges and limitations were and how to best overcome them in order to develop a successful implementation.

Firstly, one must understand what reinforcement learning is and how it can be used. Reinforcement learning refers to the training performed on machine learning models to be able to choose which actions should be taken to maximise the goal [18]. It can be used to solve very complex problems that otherwise would not be able to be solved by conventional techniques [19]. Reinforcement learning occurs during interaction with the environment by recognising characteristics and similarities of different situations, mapping these situations to actions and maximising the desired utility [17]. There are two objectives to Reinforcement Learning. These two objectives are to find solutions to a stable environment where the rules do not change, as well as being able to adapt online to a dynamic environment where an agent has to adapt its policy to different opponent playing styles. Reinforcement Learning makes use of the Reward Hypothesis where goals can be formalised as the maximisation of the expected cumulative reward [17]. Through trial and error, the agent performs exploration or exploitation to learn which actions would be best to take for each state action pair. The agent does this by continuously updating the Q values, either at the end of the episode or during the episode, based on the given reward after applying a particular action on a particular state.

In the past, applications of reinforcement learning were very limited due to weak computer infrastructure, however, after the backgammon AI super player [20] was developed in 1992, by Gerard Tesauro, progress was made. Advances in this field are still being rapidly developed since newer technological advances resulted in powerful computational technologies and computer infrastructure which allow machine learning to take place efficiently. [18]

Before moving on to choose the best fitting reinforcement learning algorithm for the game of Risk, one must fully understand all the rules and requirements of the game. Extensive research was performed in order to not miss or misunderstand any features [5]-[13]. This also included gathering information to be able to plan out the implementation by making a list of all the tasks and in which order they need to be implemented. Apart from this, research was performed in order to gather all the continent names, territory names and each territory's adjacencies [7]-[9], [21]. Some research on which programming language would be best suited for this project also took place. The top three potential programming languages for this project were Python, Java or C. At first, the project was going to be implemented in Java however then the final decision was to implement it in Python.

After fully understanding how the game of Risk should be implemented, research was performed to choose a reinforcement learning algorithm. The research performed to choose

the best fitting reinforcement learning algorithm included looking into various reinforcement learning algorithms. Some examples of the reinforcement learning algorithms that were taken into consideration are Temporal-Difference (TD) Learning, SARSA, Q-Learning and Double Q-Learning. After some thought about how these would need to be implemented to work in the game of Risk, On-Policy Monte Carlo was chosen as the best reinforcement learning algorithm for this project. [17]

The research papers [1]-[4], gave great insight on which tasks should the agent be implemented on and how and which reinforcement learning algorithm should be implemented. They also provided an idea of what can be achieved throughout this project.

The On-Policy Monte Carlo reinforcement learning algorithm makes use of an agent to learn from experience by looking at complete episodes. The agent uses sampling to discover knowledge about the environment and uses the state-action value function of the policy in order to calculate the values for the state-action pairs. It calculates the values based on the average return. Since Monte Carlo looks at complete episodes, the values are updated after the episode terminates. In order to do this, all the states that were encountered, as well as the actions performed on them and the reward returned after performing the action on the state, need to be kept track of until the episode terminates. [17]

On-Policy Monte Carlo makes use of a Q table to store the values of the state-action pairs, as well as an N table to store the number of times a state-action pair was performed, in order to be able to calculate the values for the Q table at the end of the episode. During the episode, Monte Carlo either selects a random action with probability $\epsilon$ or greedily with probability $1 - \epsilon$. For this project epsilon was set to be 1-k where k is the number of episodes performed so far, including the current episode. This means that a random float number between 0 and 1 is generated and if the random number is less than epsilon, then a random action will be chosen for the state. This is done to perform exploration in order to gain more knowledge rather than always performing exploitation. Otherwise, if the random number is greater than or equal to epsilon, then the algorithm will search the Q table for the matching state and choose the action with the highest value. If the action values in the Q table for the particular state are equal, then a random action will be chosen. After performing the action, a positive or negative reward will be given based on the outcome of applying the action on the state. The state, the action chosen and the given reward will be stored so that when the episode terminates, the values in the Q table are updated accordingly by backtracking. The values are updated with the equation Q(s,a) = Q(s,a) + ((1/N(s,a)) * (G-Q(s,a))). where G is the discounted cumulative reward with the discount factor y. [17] When y is set to 0, G is taken as the reward given when applying the action to a state. Apart from this, On-Policy Monte Carlo can either have exploring starts or no exploring starts. Exploring starts means that for the first turn of the episode, a random action is always chosen. [17] For this project, the no exploring starts version of On-Policy Monte Carlo was implemented.

# Implementation and Testing

This project was implemented in Python. In order to fully implement the requirements, the following classes were implemented: Territories, Continents, Map, Player, Cards and Risk, as well as the main function.

Firstly, the Territory and Continents classes were implemented. The Territory class consists of the __init__(), the set_territory_adjacencies() and the print_details() functions. The __init__() function takes in the name, continent id and Map as parameters. It sets the territory id to be equal to the number of territories already appended to the Map territories array and increments the Map's number of territories. It then sets the name of the territory as well as the corresponding continent id that the territory belongs to. It also sets the player id to be equal to -1, the number of troops to be equal to 0 and an empty array which will later on store the territory's adjacent territories' ids. The set_territory_adjacencies() function takes in an array of territories' ids which are adjacent to the particular territory. This function will loop through the given array and append each id to the territory's adjacencies array. The print_details() function prints all the details of the particular territory.

The Continent class consists of the __init__(), the check_owned() and print_details() functions. The __init__() function takes in the continent id, name, list of territory names, bonus and Map. This function will set the continent id, the name and the bonus according to the given details. It will set the number of territories by taking the length of the list of territories. Then it will loop through the list of territories and call the Territory class to create a Territory object for each territory. It will then append the Territory object to the territories arrays in the Continent class and Map class. The territory id will be appended to the unowned array in the Continent object. A variable called all_owned will be set to -1. This variable will be used to store the player id if the continent is all owned by the same player. The check_owned function will check if the continent is all owned by the same player by setting all_owned to be equal to the player id of the first territory in the continent. It will then loop through the rest of the continent's territories and if at least one of them doesn't match, all_owned will be set to -1. The value stored in all_owned will be returned. The print_details() function will print all the details of the particular continent.

The Map class is hardcoded to create the continents with the given 42 territories as well as to append the adjacent territories for each territory and set the bonus of each continent. The continents, continents' bonus, continents' territories and adjacent territories for each territory are all according to the specifications of the game of Risk. The Map class also contains a function called print_territories() which loops through the Map's territories and for each territory, it calls the print_details() function of the Territory class.

The Player class contains two functions; the __init__() function and the print_details() function. The __init__ function takes in the player id and initial number of troops. It will set the player id and number of troops to the given data. It will create empty arrays to later on keep track of the territories owned by the player, the full continents owned by the player as well as the player's cards. It also sets the player bonus to be equal to 0, the won_terr variable to be False and the

is_alive variable to be True. The print_details() function will display the important details of the player.

The Cards class contains three functions which are the __init__(), shuffle() and deal() functions. The __init__() function takes in the Map as a parameter and loops through the territories stored in the Map object and appends each territory id to an array named t_ids. This array will be used to assign territories to cards. The function will then loop the following 3 times, as according to the specifications of Risk, there are 42 cards containing a territory each and either an Infantry, a Cavalry or an Artillery. Therefore, for each loop, the function will loop for the number of territories divided by 3 times and it will randomly choose a territory id. It will then create a card with the chosen territory id and either 'I' for Infantry, 'C' for Cavalry or 'A' for Artillery according to which loop it is in. The chosen territory will then be removed from the t_ids list. After all the territories have been assigned to a card, two wild cards will be added to the card deck. The cards will then be sorted in order. The shuffle() function will randomly shuffle the cards if there are more than 1 card in the deck. The deal() function will return the card found at the beginning of the deck and it will remove it from the cards array (deck).

The Risk class contains the __init__(), roll_dice(), claim_continents(), check_continents(), choose_new_territory(), claim_territories(), choose_territory(), place_army(), calculate_new_armies(), check_for_sets(), place_card_armies(), check_for_attack(), attack() and move_troops() functions.

The __init__() function takes in the number of players, the attacker and defender Q tables and the number of episodes performed so far including the current episode. The function first checks that the number of players is between 2 and 6, including both numbers. Then the Map and Cards classes are called and the cards are shuffled. Various empty arrays are initialised that will be used throughout the game. Depending on the number of players, the number of initial troops is set according to the rules of Risk. All the players will then roll a dice and the player who rolls the highest dice will be set as the initial player. According to the rules of Risk, the player order starts from the initial player and then moves clockwise. To do this, the function starts from the initial player id and loops and appends the player ids until it reaches -1. Then it will loop from the last player id and appends the player ids until it reaches the initial player id. When the order has been established, the players will take turns to place one troop down on a territory until all troops have been placed. If there are unclaimed territories, the players will claim a territory by placing a troop on an unclaimed territory. This is done by calling the choose_new_territory() function.

This function will choose an unclaimed territory for a player to claim. It does this by checking if a player already owns a territory. If so, it will loop through the player's owned territories and take the continent id of the owned territory. If the continent has unclaimed territories, then the function will store the continent id and break out of the loop. If the player does not own any territories yet or the player's owned territories are all part of a continent with no more unclaimed territories, then a random continent from the unclaimed continents will be chosen and if the continent has unclaimed territories, it's id will be stored, and the function will break

out of the loop. Using the continent id, the function will choose a random unclaimed territory from that continent, and it will return the chosen territory id.

The claim_territories() function will then be called with the returned territory id in order to perform all the required actions to mark the territory as claimed. This function will check to make sure that the player has more than 0 troops and that the given territory is truly not yet owned by any player. If both are true, the function will set the territory to be owned by the player and it will place 1 troop on the territory and decrease one troop from the player. It will also append the territory id to the player's territories_owned array. The function will also get the territory's continent id and remove the territory from the continent's unowned array and from the unclaimed_territories array.

If all territories are claimed, then the players will place 1 troop on one of their own claimed territories. The choose_territory() function is called and then the place_army() function is called with the given territory id. The choose_territory() function will loop until a territory is found. It will randomly choose a territory from the player's owned territories and if the territory has adjacent opponent territories, then the function will return the territory id. If the player does not have any territories left, the function will return -1. The place_army() function will first check to make sure that the player has more than 0 troops and that the territory is already owned by the player. If both are true, the function will increase the territory's number of troops by 1 and decrease the player's number of troops by 1.

The claim_continents() function is called to check whether all the territories in a continent have been claimed and if so, it will remove the continent from the unclaimed_continents array. It will also call the check_owned() function from the Continent class and if the continent is all owned by the same player, then it will append the continent id to the player's continents_owned array and add the continent bonus to the player's bonus. It will also remove the continent id from the not_owned_continents array.

The __init__() function will then loop the following code until a player wins the game by claiming all the territories. The players, in the decided order, will each play a 3-step turn which includes getting and positioning new troops, attacking, and moving troops.

In order to perform the first step, which is calculating the new troops for the player and placing them on the player's claimed territories, the calculate_new_armies() function will first be called. This function will calculate the number of troops that the player should receive based on the Risk rules. First, the number of territories that the player owns is divided by 3, ignoring any remainder. If the result is less than 3, it will be set to 3 as the rules specify that a player shall always receive at least 3 troops. The function will then check if the player has more than 2 cards. If this is the case, the check_for_sets() function is called to check if the player has any full sets to return. Then the place_card_armies() function is called. If a set has been returned, the player will receive the appropriate bonus number of troops. Apart from these, if the player owns entire continents, they will also receive the specified number of troops.
The check_for_sets() function first sorts the cards to get them in the order A, C, I and W. Then the function will loop through the player's cards and if a set is found, that set is returned. The

sets either need to be 3 cards of the same army type or 3 cards, one of each army. The wild card can replace any card. If after returning all the full sets, the player has more than 4 cards, random cards will be removed until the player no longer has more than 4 cards. The list of territories of the returned set will be returned by the function. The place_card_armies() function will then loop through the returned card's territories and if the player owns the territory, 2 troops will be placed on that territory.

According to the number of new troops, the function will loop so that all the new troops will be placed in the player's claimed territories. In each loop, the choose_territory() function will be called which returns a territory id and then the place_army() function will be called with the territory id. If the player lost all their territories, then -1 would be returned and game_ongoing would be set to be False and the function would break out of the loop.

In order to perform the attack phase, the check_for_attack() function is called which takes in the player, map, attacker and defender Q tables and number of episodes performed including the current episode. The check_for_attack() function will loop until the attack variable becomes False as there is no limit to how many attacks can be performed per turn, so the function will loop until the algorithm chooses to not perform any more attacks. It will loop through the player's territories and check if the territory has at least 2 troops. If the territory has at least 2 troops, the function will check if there are any adjacent opponent territories. If adjacent opponent territories are found, the On-Policy Monte Carlo algorithm will choose whether or not an attack should be made and with how many dice. In order to do this, epsilon is calculated to be 1 divided by the number of episodes performed so far including the current one. Then a random number between 0 and 1 will be generated. If the random number is less than epsilon, a random action will be chosen provided that the number of dice rolled does not exceed 3 and is always one less than the number of troops on the territory. Otherwise, the function will search through the Q table and select the action with the highest value. If the values are equal, a random action will be chosen. If the attacker decides to attack, the defender will choose whether to defend with 1 or 2 dice with the On-Policy Monte Carlo algorithm. Another random number will be generated and if the number is greater than epsilon, then the action with the highest value will be chosen provided that defending with 2 dice is only chosen if the player has at least 2 troops. If the values in the Q table are equal or if the random number is less than epsilon, the number of dice is chosen randomly; either 1 or 2. The attack() function is then called to perform the attack. After the attack() function terminates, the rewards are returned and the actions and their rewards will be appended to the attacker_actions and defender_actions arrays.

The attack() function rolls the chosen amount of attacker and defender dice by calling the roll_dice() function. It will then compare the dice in order, highest dice first. For each attacker and defender dice pair, if the attacker dice value is higher than the defender dice value, then the defender loses a troop. Otherwise, if the dice values are equal or the defender dice value is greater than that of the attacker, the attacker loses a troop. The attacker and defender can never lose more than 2 troops in each attack. The function will then remove the lost troops from their respective territories. If the defender territory ends up with 0 troops, that means that the attacker won the territory. If this is the case, the territory id will be removed from the

defender's territories_owned array and appended to the attacker's territories_owned array. The territory's player id will be set to the attacker id and the attacker's won_terr variable will be set to true. Depending on the number of attacker dice rolled, that amount of troops will be moved from the attacker territory to the newly claimed territory. If the defender claimed the continent of which the territory was part of, the continent bonus and continent id will be removed from the player's profile. The continent will be reset to unowned. The rewards are finally calculated, for the attacker and defender actions, by decreasing the number of troops lost from the number of opponent troops defeated. If the player won at least one territory, the player is given a card from the deck. If the deck is empty, then the previously returned cards will be shuffled and used as the new deck. The player's won_terr variable is reset to False and the player's claimed territories and continents are sorted.

The third and final phase happens by calling the move_troops() function in order to make any necessary changes to the troops placement. The move_troops() function loops through the player's territories and checks each territory's adjacent territories. If the territory has an opponent adjacent territory, then the function will not move the troops from that territory. If all the adjacent territories of a territory are all owned by the player, then the function will select a random adjacent territory and if that territory has adjacent opponent territories, the function will move all troops except for 1 to this adjacent territory.

Then, the function will loop through the players to check if a player was defeated. If the current player defeated another player in this turn, then the defeated player is removed from the player order and set to not alive. The cards of the defeated player are assigned to the current player.

After all the players perform their turn, the check_continents() function will then be called. The check_continents() function will loop through the continents found in the not_owned array and for each continent, it will call the check_owned() function from the Continent class. If the continent has been indeed claimed by a player, the function will append the continent id to the player's continents_owned array as well as add the continent's bonus to the player's bonus and remove the continent from the not_owned_continents array.

If all the continents are owned, the function will loop through the continents to check if they are all owned by the same player and if this is the case, game_ongoing will be set to False and the player who owns all the continents wins.

The roll_dice() function simulates rolling a dice by returning a random number from 1 to 6.

The main function initialises the two Q and N tables as dictionaries for the On-Policy Monte Carlo algorithm to use for the attack and defend state-action values.

The dictionaries for the attacker actions, contain the state as the number of troops in the attacker territory, the number of troops in the defender territory and either True, if the attacker territory has more troops than the defender territory, or False if otherwise. An attack can only happen if the attacker territory contains at least 2 troops and to roll the maximum

amount of dice, which is 3, the attacker territory needs to have at least 4 troops. Because of this, for the number of troops in the attacker territory, the dictionary has 2, 3 and 4 troops. Similarly, since the defender will always have at least 1 troop and will lose at most 2 troops, the dictionary contains 1, 2 and 3 troops. The other variable in the dictionary is used for the action values to be more accurate as if the defender has more troops than the attacker it may be different than if the attacker has more troops than the defender. The actions for each state consists of rolling either 0, 1, 2 or 3 dice. The values for the dice are initially all set to 0. For the states that have 2 attacker troops, the function will set the values for choosing 2 or 3 dice to -100 as these are not allowed according to the rules of the game. Similarly, when the attacker has 3 troops, the function will set the values for choosing 3 dice to be -100 as well.

The dictionaries for the defender actions, contain the state as the number of troops in the defender territory and the number of dice that the attacker chose to roll. Since in the attacker dictionaries, the defender number of troops is between 1 and 3 troops, both numbers included, this was implemented to be the same in the defender dictionary. The number of attacker dice rolled is between 1 and 3, both included, as if the attacker chooses to roll 0 dice, then the defender does not have the option to roll any dice. The values for the defender dice are initially all set to 0, however since the defender can only roll 2 dice if there are at least 2 troops in the defender territory, the function will set the values for choosing 2 dice to -100 for when there is only 1 troop in the defender territory.

Then, the function will loop the following according to the specified number of episodes. For each episode, the Risk class is called with the number of players, the number of episodes played so far and the two Q dictionaries: Q and Q_d. When the game ends, the function will loop through the attacker_actions and defender_actions arrays and it will update the values of the respective Q dictionary for each state-action pair with the reward. The state-action pair value stored in the respective N dictionary will be increased by 1. Then, it will make use of the equation Q(s,a) = Q(s,a) + ((1/N(s,a)) * (total_reward-Q(s,a))) to update the state-action pair value in the respective Q dictionary. Since the discount factor y is taken as 0 in this implementation, the total_reward is the reward of the state-action pair itself.

Testing was performed concurrently whilst developing the game to check that everything was being implemented correctly without any logic errors. For each part of the implementation, testing was performed by printing out the initial states and their results after performing any actions. The Q table was also printed, and some values were manually calculated based on the actions that were performed and their rewards to make sure that everything was correct.

# Evaluation and Critical Analysis

The implementation achieved all the discussed aims and objectives as well as all the rules and requirements of the game Risk. The implemented reinforcement learning algorithm maximises the chances of the player winning by choosing the attack options. However, since the defender dice is also controlled by another agent, the two agents try to maximise their own player's chances of winning and hence try to decrease the other's chance of winning. It is quite interesting to see and adds a greater learning experience. It also makes it a bit harder for the attacker to win territories.

One problem that occurred during training with a large number of episodes, for example 1000 episodes, was that sometimes the algorithm would get stuck in a loop where both players kept winning the same few territories from each other and hence kept drawing cards. Because of this, they kept getting full sets and the card bonus kept increasing to a very large number. This is because the risk rules specify that after 5 sets are returned, the next set will always be 5 more troops added to the number received from the previous set bonus. This was fixed by limiting the number of troops taken from the card bonus to 30. Therefore, the players can never take more than 30 new troops no matter how many sets are returned in the game.

This implementation could be mainly improved by adding graphics for the Map, territories, troops, and cards. It would certainly make the experience better and easier to see all the moves since it would be very visual. After adding graphics, some sound effects could also be added to further enhance the experience and immerse the player into the game.

Another improvement would be to add reinforcement learning for both the troop placement and troop movement. This would have to be done by using an algorithm which is able to adapt online to a dynamic environment as the agent would have to adapt its policy to different opponent playing styles and moves. An example of this algorithm would be the Monte Carlo Tree Search algorithm.

Finally, adding user interaction could also improve the implementation. One could have a player playing against the smart strategist. This would in turn make the player more interested long-term, as the player can become determined to improve their skills and defeat the smart strategist. The player can also learn from seeing the moves that the smart strategist chooses during gameplay.

# Conclusion

In conclusion, the proposal ‘A smart strategist for Risk’ was about implementing the entire game of Risk as well as developing and implementing a reinforcement learning algorithm for the smart strategist to be able to learn and maximise the winning chances. All these requirements were developed and achieved in the final implementation of the project. The final implementation consists of smart strategists playing the game of Risk against each other in order to learn what actions are best to take during gameplay.

During this project, knowledge was gained mainly about different reinforcement learning algorithms, how they work, how they are used and how they can be applied. Apart from this, a greater understanding of which reinforcement learning algorithms should be applied to certain scenarios was gained. Knowledge was also gained on how to go about implementing an entire complex multiplayer game from scratch and how to overcome any difficulties that can be encountered. A deeper understanding of all the rules and requirements of the game of Risk was also gained along with gaining knowledge about the available different versions of the game of Risk.

# References


[1] E. Blomqvist, “Playing the Game of Risk with an AlphaZero Agent”, Aug 23, 2010. Accessed: Feb. 17, 2022. [Online]. Available: http://kth.diva-portal.org/smash/get/diva2:1514096/FULLTEXT01.pdf

[2] J. Lozano and D. Bratz, “A Risky Proposal: Designing a Risk Game Playing Agent”. Accessed: Feb. 17, 2022. [Online]. Available: https://cs229.stanford.edu/proj2012/LozanoBratz-ARiskyProposalDesigningARiskGamePlayingAgent.pdf

[3] G. Robinson, “The Strategy of Risk”. Accessed: Mar. 17, 2022. [Online]. Available: https://web.mit.edu/sp.268/www/risk.pdf

[4] F. Hahn, “Evaluating Heuristics in the Game Risk, An Aritifical Intelligence Perspective”. Accessed: Mar. 17, 2022. [Online]. Available: https://project.dke.maastrichtuniversity.nl/games/files/bsc/Hahn_Bsc-paper.pdf

[5] Gather Together Games, How To Play Risk. (Mar. 3, 2020). Accessed: Mar. 17, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=5g48vXnnf30&t=151s

[6] “RISK” gamerules.com. https://gamerules.com/rules/risk-board-game/ (Accessed Mar. 19, 2022)

[7] “List of Regions” risk.fandom.com. https://risk.fandom.com/wiki/List_of_Regions (Accessed Mar. 19, 2022)

[8] A. Smith. “List of Territories in Risk” listsclick.com. https://listsclick.com/list-of-territories-in-risk/ (Accessed Mar. 21, 2022)

[9] “Play Risk Game” commodoregames.com. https://commodoregames.com/risk-online-game (Accessed Mar. 21, 2022)

[10] Board Game Museum, How To Play Classic Risk Board Game. (Mar. 19, 2017). Accessed: Jun. 1, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=d7zuAejNUsA

[11] Watch It Played, Risk - How To Play - A Complete Guide!. (Dec. 16, 2021). Accessed: Jun. 2, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=Xo8RSozX6Ac&t=785s

[12] “Risk Game Rules” ultraboardgames.com. https://www.ultraboardgames.com/risk/game-rules.php (Accessed Jun. 2, 2022)

[13] “Risk For two Players | Game Rules” ultraboardgames.com. https://www.ultraboardgames.com/risk/risk-for-2-players.php (Accessed Jun. 2, 2022)

[14] M. Sonesson. “Creating an AI for Risk board game” martinsonesson.wordpress.com. https://martinsonesson.wordpress.com/2018/01/07/creating-an-ai-for-risk-board-game/ (Accessed Mar. 17, 2022)

[15] Rahul_Roy. “ML | Monte Carlo Tree Search (MCTS)” geeksforgeeks.org. https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/ (Accessed Apr. 6, 2022)

[16] “Monte Carlo Tree Search” the-algorithms.com. https://the-algorithms.com/algorithm/monte-carlo-tree-search (Accessed Apr. 6, 2022)

[17] J. Bajada. Reinforcement Learning [PowerPoint Slides]. Available: https://www.um.edu.mt/vle/course/view.php?id=58619 (Accessed Apr. 6, 2022)

[18] B. Osiński and K. Budek. “What is reinforcement learning? The complete guide” deepsense.ai. https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/ (Accessed Jun. 9, 2022)

[19] A. Joy. “Pros and Cons of Reinforcement Learning” pythonistaplanet.com. https://pythonistaplanet.com/pros-and-cons-of-reinforcement-learning/ (Accessed Jun. 9, 2022)

[20] ”TD-Gammon” en.wikipedia.org. https://en.wikipedia.org/wiki/TD-Gammon (Accessed Jun. 9, 2022)

[21] tlmader. “risk” github.com. https://github.com/tlmader/risk (Accessed Mar. 18, 2022)

[22] arman-aminian. “risk-game-ai-agent” github.com. https://github.com/arman-aminian/risk-game-ai-agent (Accessed Mar. 18, 2022)

[23] kengz. “Risk-game” github.com. https://github.com/kengz/Risk-game (Accessed Mar. 18, 2022)

[24] godatadriven. “risk-analysis” github.com. https://github.com/godatadriven/risk-analysis (Accessed Mar. 18, 2022)

[25] Whysmerhill. “Risk” github.com. https://github.com/Whysmerhill/Risk (Accessed Mar. 19, 2022)

[26] chronitis. “pyrisk” github.com. https://github.com/chronitis/pyrisk (Accessed Mar. 19, 2022)

[27] sfucmpt105-beta. “risk” github.com. https://github.com/sfucmpt105-beta/risk (Accessed Mar. 21, 2022)

[28] osamasherif22. “RiskGamePython” github.com. https://github.com/osamasherif22/RiskGamePython (Accessed Mar. 21, 2022)