Project Brief

Subject and Code	ARI2204 – Reinforcement Learning
Assessment	30% of subject marking
Deadline	Monday 26th May 2025 – 15:00hrs CET

Read the instructions carefully, thoroughly and completely before attempting this project. Failure to comply with the requirements can result in unnecessary loss of marks or potentially disqualification.

This is a group project. Make sure you declare the members of your group beforehand. Each group should be made of **3 students** (with exceptions only where there are not enough students). **Make sure you submit your deliverables before the deadline expires.** Late submissions will not be accepted.

1. Overview

In this project you will be implementing a simplified **Blackjack** environment that will be used to train an agent to learn how to play the game using different Reinforcement Learning algorithms. This is similar (but not the same) to the one discussed in class and described in Section 5.1 of the book [1].

Figure 1: The game of Blackjack

1.1 Rules of the game

In this version of the game a **single** deck of cards is used, containing 52 cards, i.e. 4 cards of each of the following $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$. The player plays against the dealer. The player gets dealt 2 cards, face up. The dealer gets one card, face up. The objective of the game is that the player and the dealer must get as close to 21 (inclusive) without going over 21. If neither goes over 21, the hand with the highest sum wins. If both hands are equal and not over 21, then it is a draw.

The face cards, $\{J, Q, K\}$ each have a value of 10. The Ace card, A, can take two values, 1 or 11, whichever brings the total sum of cards held by the player or dealer closer to 21 without going over.

1.1.1 Player Actions

After the player sees their two cards and the card of the dealer, they must decide whether to:

1. **HIT** – Take another card, hoping to get closer to 21 without going over.
2. **STAND** – Stop and hope the sum of the hand will be higher than that of the dealer.

If the sum of cards is less than 12 it always makes sense to HIT. If the sum of cards is 21 it always makes sense to STAND.

Your agent will learn what to do if the sum is any number in $[12 \,..\, 20]$, given the card of the dealer $[2 \,..\, 11]$ and whether the player has an Ace or not.

If the player has an Ace that is counting as 11, the risk of going over 21 when doing a HIT is lower, because if a high value card is received, the value of the Ace can change to 1.

1.1.2 Dealer Policy

The dealer only plays after the player chooses to STAND. If the player goes over 21, the dealer wins automatically.

The dealer has a very strict policy:

1. **HIT** - if the sum of dealer cards is *less* than 17.
2. **STAND** otherwise (the sum is *greater or equal* to 17).

Even if the player has a higher sum than the dealer, the dealer must still STAND at a value of 17 or higher, thus handing the win to the player.

2 Implementation

You can use any programming language of your choice, as long as you satisfy the requirements.

2.1 Blackjack Environment

Develop a very simple Blackjack environment that consists of the following components:

1. A card deck of 52 cards, with functionality to shuffle the deck randomly and get the next card from the deck.
2. A Blackjack round, which consists of the current *environment state* (cards dealt to each player, or sum together with a flag to indicate whether an ace was used). It must have the functionality to HIT (give another card to the player from the deck, as long as the current sum is less than 21), STAND (execute the dealer policy, as long as the player has not gone over 21), and determine whether the outcome was a Win (for the player), a Loss, or a Draw after both player and dealer complete their actions.

2.2 Reinforcement Learning Framework

Implement an agent that improves its policy using Reinforcement Learning. The implementation is going to learn the $q(s, a)$ values for each agent state-action pair $\langle s, a \rangle$ where $s \in S$ and $a \in \mathcal{A} = \{HIT, STAND\}$. Remember that your policy only needs to decide between the two actions when the player sum in the range $[12 \,..\, 20]$. The policy should always HIT when the sum is less than 12, and STAND when the sum is 21 (i.e. you do not need to learn the $q(s, a)$ for these state-action pairs).

2.2.1 Agent State

For each Blackjack environment state, generate an agent state to be used by the RL algorithm. This consists of the player sum $[12 \,..\, 20]$, the dealer card $[2 \,..\, 11]$ and a Boolean flag to indicate whether the player sum currently includes the use of an Ace card as 11 (i.e. if the player goes over 21, there is the option of reducing the sum by 10 and switching this Boolean flag off).

2.2.2 State-Action Values and Counts

Choose an efficient data structure (such as an array or a hash table/dictionary) to store and efficiently look up the following:

1. The number of times an action $a \in \mathcal{A}$ was selected in a state $s \in S$.
2. The current estimated expected value of taking an action $a \in \mathcal{A}$ in a state $s \in S$.
3. Functionality to update the estimated expected value and increment the count.

2.2.3 Episodes, Policies, and Rewards

Each Episode represents a single round of Blackjack. Before each episode, initialise and shuffle a new deck. At each step of the episode your policy must decide whether to HIT or STAND. STAND or going over 21 both lead to a terminal state.

If the outcome of the round is a win, the reward is +1, if the outcome is a loss, the reward is -1, otherwise in case of a draw the reward is 0. Since the episodes are very short, assume no discount factor ($\gamma = 1$).

For each episode $(S_0, A_0, S_1, A_1, \ldots, S_T)$ you will need to extract each $\langle S_i, A_i \rangle$ for all $0 \leq i < T$ and update its estimated state action value $q(s, a)$.

2.3 Monte Carlo On-Policy Control

Implement the **Monte Carlo On-Policy Control** algorithm to estimate the state-action value function $q(s, a)$ for Blackjack. Use an ***ϵ*-Greedy policy** where k is the episode count so far (including the current one).

Implement the following ***two*** variants of Monte Carlo On-Policy Control:

1. *Exploring Starts*: when the player sum is in $[12 \,..\, 20]$ the first state of the episode always gets a random action. In subsequent states (with sum in $[12 \,..\, 20]$) of the same episode, the action is selected randomly with probability $\epsilon = \frac{1}{k}$ and greedily with probability $1 - \epsilon$.

2. *No Exploring Starts*: when the player sum is in $[12\,..\,20]$ the action is selected randomly with probability ϵ, and greedily with probability $1-\epsilon$. Run this algorithm with ***three*** different configurations: $\epsilon = \frac{1}{k}$, $\epsilon = e^{-k/1000}$ and $e^{-k/10000}$.

So, in total you will have **four** different configurations. One with exploring starts, and three without.

2.4 SARSA On-Policy Control

Implement the **SARSA On-Policy Control** algorithm to estimate the state-action value function $q(s,a)$ for Blackjack, using an ***ϵ*-Greedy policy**, where k is the episode count so far (including the current one).

Run it using ***four*** configurations with $\epsilon = 0.1$, $\epsilon = \frac{1}{k}$, $\epsilon = e^{-k/1000}$ and $\epsilon = e^{-k/10000}$, for 100,000 episodes each. Set the step size, $\alpha = \frac{1}{N(s,a)+1}$, where $N(s,a)$ is the number of times the $\langle s,a\rangle$ pair was encountered (including the current time step) so far across all episodes.

2.5 Q-Learning (SARSAMAX) Off-Policy Control

Implement the **Q-Learning / SARSAMAX Off-Policy Control** algorithm to estimate the state-action value function for $q(s,a)$ for Blackjack, using an ***ϵ*-Greedy policy**, where k is the episode count so far (including the current one).

Run it using ***four*** configurations with $\epsilon = 0.1$, $\epsilon = \frac{1}{k}$, $\epsilon = e^{-k/1000}$ and $\epsilon = e^{-k/10000}$, for 100,000 episodes each. Set the step size, $\alpha = \frac{1}{N(s,a)+1}$, where $N(s,a)$ is the number of times the $\langle s,a\rangle$ pair was encountered (including the current time step) so far across all episodes.

2.6 Double Q-Learning Off-Policy Control

Implement the **Double Q-Learning Off-Policy Control** algorithm to estimate two state-action value functions, $q_1(s,a)$ and $q_2(s,a)$ for Blackjack, using an ***ϵ*-Greedy policy**, where k is the episode count so far (including the current one). Remember that at every timestep, you should either update q_1 or q_2, but not both. Choose which one to update randomly with 0.5 probability at each timestep. After running all episodes, the final q-value for a state-action pair $\langle s,a\rangle$ will be the mean of both functions:

$$q(s,a) = \frac{1}{2}(q_1(s,a) + q_2(s,a))$$

Run it using ***four*** configurations with $\epsilon = 0.1$, $\epsilon = \frac{1}{k}$, $\epsilon = e^{-k/1000}$ and $\epsilon = e^{-k/10000}$, for 100,000 episodes each. Set the step size, $\alpha = \frac{1}{N(s,a)+1}$, where $N(s,a)$ is the number of times the $\langle s,a\rangle$ pair was encountered (including the current time step) so far across all episodes.

So, you need to implement 4 algorithms, and for each algorithm you are going to have 4 exploration configurations, for a total of 16 runs.

3 Evaluation

Run each algorithm configuration (four Monte Carlo, four SARSA, four Q-Learning, and four Double Q-Learning configurations) for 100,000 episodes, and extract the following information:

1. For every 1000 episodes, keep a count of how many of the previous 1000 episodes ended up in a Win for the player, a Loss for the player, and a Draw.
2. The number of unique state-action pairs, $\langle s, a \rangle$ explored after running all episodes.
3. The counts of how many times each unique state-action pair $\langle s, a \rangle$ was selected, after running all episodes.
4. The estimated Q values for each unique state-action pair $\langle s, a \rangle$ after running all episodes.

Use this information to plot and analyse the performance and effects of different configurations.

1. Plot the number of wins, losses and draws on **one line-chart**, with the number of episodes on the x-axis (a separate chart for each algorithm and algorithm configuration).
2. Plot the counts of each unique state-action pair on a **bar chart** sorted by **highest count first**, (a separate chart for each algorithm and algorithm configuration), to show how often each state-action pair was executed.
3. Plot the total number of unique state-action pairs, $\langle s, a \rangle$, as a **bar chart across all configurations of each algorithm** (so each histogram will have 4 bars), to show the effect of the different configurations on *exploration*.

Using the estimated Q values from each configuration, find the best action to take in each state, and build a *Blackjack Strategy table*, with the columns being the dealer's card $2, 3, 4 \ldots 10, A$ and the rows being the player's sum from 20 down to 12. The cells should contain an **H** if the best action is a HIT, and **S** if the best action is a STAND. You should have **two** tables for each algorithm configuration, one for when the player is using an Ace as 11, and one when not.

For each policy obtained using each algorithm configuration, calculate the *mean* number of wins, draws and losses over the **last 10,000 episodes**, and calculate the advantage of the dealer using the following formula, where l is the mean number of losses and w is the mean number of wins:

$$\frac{l - w}{l + w}$$

Plot the dealer advantage of each algorithm configuration on one chart and find out which algorithm managed to find a policy that minimises the dealer advantage the most.

4 Deliverables

You should submit all deliverables in **one (1) zip file** on VLE with your name. It should contain the following deliverables:

1. Report

A report of around **3000 words** (+/- 10%) in **PDF** format.

In the report you should give an overview of the algorithms used, describe how you implemented them and analyse the results and observations. Discuss the differences between each algorithm and algorithm configuration, and how ***exploration*** and ***exploitation*** had an effect on the performance of each algorithm. Any citations and references to any literature you may need to quote or refer to should be done using either the ACM, AAAI or IEEE reference style.

2. Source Code

Put the source code in the zip file. Make sure it is adequately structured and understandable.

Any programming language can be used, but the algorithms discussed during the study unit must be implemented as part of this project. ***The code will be checked for copying or high similarity with online sources using automated tools or manual spot checks.***

If you prefer you can submit the code as a Jupyter Notebook, and include the necessary plots in it. **However, the plots must still be in the report.**

5 Distribution of Work

Each team must include a "Distribution of Work" page, at the back of the report. (This does not count towards the word count limit and can be included after the reference list). In this page put the names of each team member and a very short description (a few bullet points) of the tasks that each person was responsible for. **Each member must sign this page**, indicating that they agree with the reported distribution of work.

6 Plagiarism, Collusion and use of Generative AI Tools

Each group is expected to work on its own.

Plagiarism is taken very seriously and any suspicion of plagiarism, collusion, copying, or claims of the work of others to be one's own will be thoroughly investigated and if confirmed, severe action will be taken according to the regulations of the University of Malta.

You should sign this form and it must be scanned and included with your submission.
https://www.um.edu.mt/media/um/docs/faculties/ict/mainfacultyofictfiles/PlagiarismForm.pdf

For more details about plagiarism and collusion you can consult the university guidelines at:
https://www.um.edu.mt/__data/assets/pdf_file/0007/436651/UniversityGuidelinesonPlagiarism.pdf

Furthermore, the Department of AI has its own policy on the use of Generative AI tools in assessed coursework, which requires that you declare their use and to what extent they were used. Refer to the accompanying document for more details.

7 Bibliography

[1] A. G. Barto and R. S. Sutton, Reinforcement Learning: An Introduction, 2nd ed., MIT Press, 2018.

Appendix A: Marking Criteria

Table 1 serves as a guide to indicate what criteria will be used to assess each component. The distribution of marks for each Part will be assigned as follows:

	Weak	***Satisfactory***	***Excellent***
Implementation (40%)	Program doesn't work or does not run. Submitted source code does not compile. Missing essential parts. Program does not do what it is intended.	Program works as intended. Some bugs, or missing a few aspects.	Program works as intended. All requirements implemented and demonstrates the intended concepts well.
Evaluation (40%)	Little or no evaluation metrics included in the report.	Most of the evaluation metrics included in the report. Some discussion about the results.	All required evaluation metrics included in the report, together with a clear analysis and discussion of the observed results.
Writing (15%)	Language of the report is unclear. Bad grammar or incoherent writing, reflecting lack of proof reading. Work not described in enough detail.	Language of the report is coherent, with minor grammatical or language issues. The document is properly structured and some details about the implementation and results have been provided.	Language of the report is clear, flowing and grammatically correct, showing evident proof reading. The document is properly structured and all the expected details and discussions about results have been provided.
References (5%)	Minimal or no references to literature included in the report. Not following the required conventions.	A few references included in the report, mostly following one of the specified referencing conventions.	A good number of references and in-line citations included in the report, using one of the specified referencing conventions.

Table 1: Marking Criteria