COMP20230 Assignment 2

Team Members:
Qingtian Ye - 23202064
Michael Glennon - 22202448
Yu Li- 23205800

Github:

https://github.com/Viskym/CardGame

1. Problem Statement

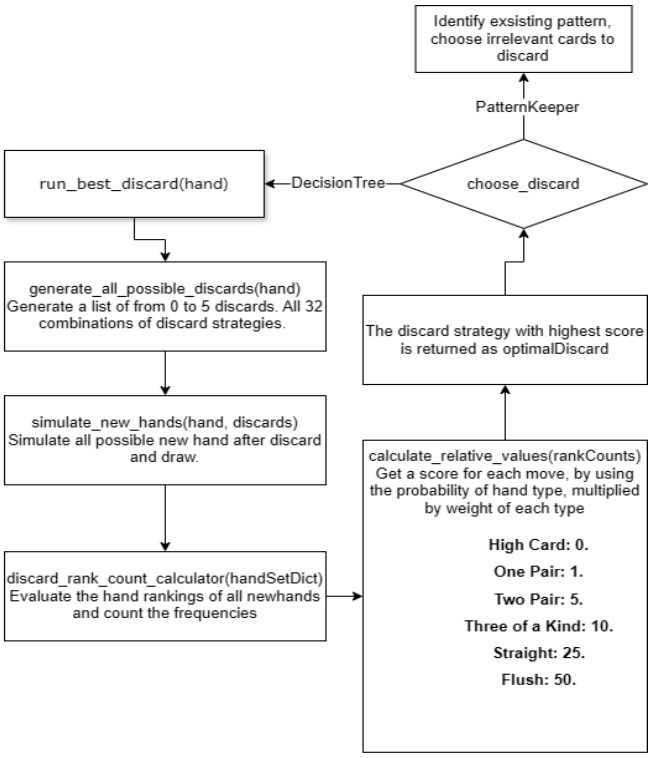
Five-card draw (also known as Cantredraw) is a poker variant that is considered the simplest variant of poker, and is the basis for video poker. We aim to develop an AI bot for the game, which will evaluate what's in hand, and make best discard/replace strategies.
Player: Our implementation is designed for 2 players. One of them will be played by an AI bot.
Deck: Standard 52-card deck.
Game summary:

a. The deal: Five cards face down are dealt to each player.
b. Betting Round: After the deal, the first betting round starts.
c. Play: Each player may discard and draw up to five cards.
d. Betting Round (Final): A second round of betting ensues.
e. Showdown: The winner of the hand is the player who holds the highest hand according to universal poker hand rankings.

Searching through all combinations is NP (Nondeterministic Polynomial time) class of problems.
The algorithm is efficient for the problem size (a five-card hand), as both time and space complexities are bounded and relatively small due to the fixed size of poker hands. We always choose 0-5 cards to replace, from 47 unknown cards. Unlike Texas Hold'em where multiple runs exist, namely Flop, River, Turn. There is only one round of discard. Searching through all combinations is possible. Strategies like Monte Carlo methods were not necessary for our game.

2. System Functionalities

The functionalities of the game are written in `card.py` `hand_rank.py` and playable by running `FiveCardDraw.py`. `test.py` is a unit test script for essential functions, doc tests were written in some functions serving the role of documentation at the same time. Betting strategy was not included in our project design. Current version of the game requires the player to choose betting for the Bot.
As the implementation of the game itself is not the main purpose of this project. This paper will start from the discard decision making function inside the bot.py.



Cards	High Card	One Pair	Two Pairs	Three of a Kind	Straight	Flush	Full House	Four of a Kind	Straight Flush
()	1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
(♦9,)	0.74468	0.25532	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
(♦10,)	0.74468	0.25532	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
(♦10, ♦9)	0.60037	0.36633	0.02498	0.00833	0.00000	0.00000	0.00000	0.00000	0.00000
(♥6,)	0.65957	0.25532	0.00000	0.00000	0.08511	0.00000	0.00000	0.00000	0.00000
(♥6, ♦9)	0.58927	0.36633	0.02498	0.00833	0.01110	0.00000	0.00000	0.00000	0.00000
(♥6, ♦10)	0.58927	0.36633	0.02498	0.00833	0.01110	0.00000	0.00000	0.00000	0.00000
(♥6, ♦10, ♦9)	0.52242	0.41295	0.04385	0.01733	0.00222	0.00000	0.00111	0.00012	0.00000
(♥Q,)	0.65957	0.25532	0.00000	0.00000	0.08511	0.00000	0.00000	0.00000	0.00000
(♥Q, ♦9)	0.58927	0.36633	0.02498	0.00833	0.01110	0.00000	0.00000	0.00000	0.00000
(♦10, ♥Q)	0.57447	0.36633	0.02498	0.00833	0.02590	0.00000	0.00000	0.00000	0.00000
(♥Q, ♦10, ♦9)	0.51551	0.41295	0.04385	0.01733	0.00913	0.00000	0.00111	0.00012	0.00000
(♥6, ♥Q)	0.56337	0.36633	0.02498	0.00833	0.03700	0.00000	0.00000	0.00000	0.00000
(♥6, ♥Q, ♦9)	0.51724	0.41295	0.04385	0.01733	0.00740	0.00000	0.00111	0.00012	0.00000
(♥6, ♦10, ♥Q)	0.51428	0.41295	0.04385	0.01733	0.01036	0.00000	0.00111	0.00012	0.00000
(♥6, ♥Q, ♦10, ♦9)	0.49040	0.42829	0.04975	0.02300	0.00387	0.00275	0.00161	0.00029	0.00003
(♠8,)	0.74468	0.25532	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
(♠8, ♦9)	0.60037	0.36633	0.02498	0.00833	0.00000	0.00000	0.00000	0.00000	0.00000
(♠8, ♦10)	0.60037	0.36633	0.02498	0.00833	0.00000	0.00000	0.00000	0.00000	0.00000
(♠8, ♦10, ♦9)	0.51446	0.41295	0.04385	0.01733	0.00000	0.01018	0.00111	0.00012	0.00000
(♥6, ♠8)	0.57447	0.36633	0.02498	0.00833	0.02590	0.00000	0.00000	0.00000	0.00000
(♥6, ♠8, ♦9)	0.51551	0.41295	0.04385	0.01733	0.00913	0.00000	0.00111	0.00012	0.00000
(♥6, ♠8, ♦10)	0.51946	0.41295	0.04385	0.01733	0.00518	0.00000	0.00111	0.00012	0.00000
(♥6, ♠8, ♦10, ♦9)	0.49273	0.42829	0.04975	0.02300	0.00247	0.00183	0.00161	0.00029	0.00002
(♠8, ♥Q)	0.58927	0.36633	0.02498	0.00833	0.01110	0.00000	0.00000	0.00000	0.00000
(♠8, ♥Q, ♦9)	0.52242	0.41295	0.04385	0.01733	0.00222	0.00000	0.00111	0.00012	0.00000
(♠8, ♦10, ♥Q)	0.51946	0.41295	0.04385	0.01733	0.00518	0.00000	0.00111	0.00012	0.00000
(♠8, ♥Q, ♦10, ♦9)	0.48987	0.42829	0.04975	0.02300	0.00533	0.00182	0.00161	0.00029	0.00003
(♥6, ♠8, ♥Q)	0.50435	0.41295	0.04385	0.01733	0.01011	0.00993	0.00111	0.00012	0.00025
(♥6, ♠8, ♥Q, ♦9)	0.49131	0.42829	0.04975	0.02300	0.00390	0.00184	0.00161	0.00029	0.00001
(♥6, ♠8, ♦10, ♥Q)	0.49158	0.42829	0.04975	0.02300	0.00363	0.00184	0.00161	0.00029	0.00001
(♦10, ♦9, ♥6, ♠8, ♥Q)	0.50540	0.42068	0.04681	0.02054	0.00301	0.00195	0.00138	0.00022	0.00001

The `bot.choose_discard(isFinal: bool, hand: list[Card])` function is designed to optimise discard strategy through two distinct modes. The first mode, "PatternKeeper," evaluates the current hand and prioritises retaining the highest-formed pattern while discarding all other cards. Effective for recognizing pairs, two pairs, three of a kind, and four of a kind, this algorithm, however, does not identify potential straight and flush candidates as high cards, thus not always ensuring the optimal discard choice.
The second mode, "DecisionTree," meticulously explores every possible combination of discards to ascertain and return the most advantageous solution for each hand, thereby optimising the potential for a winning configuration.
Meanwhile, the `bot.choose_betting(isFinal: bool, hand: list[Card])` function governs betting

actions. However, due to time constraints and the complex, often ambiguous mathematical nature of betting strategies, this function remains incomplete in the current project iteration.

3. Algorithm and Data Structure

The 'decision tree' mode for discard strategy generates a list of all possible hands (from drawing from the draw pile) for all possible discards from the current hand. The possible hands are weighted according to their probability and possible hand rank count, and the discard corresponding to the largest combined rank probability weighting is chosen. In our implementation, the abstract data type is Tree, and we used the Hashtable as our concrete data structure.

Below is the mathematical expression of all possible combinations of hands, given that n represents the number of cards left in the deck and k represents the number of cards to be replaced. Resulting in O(C(n,k)). Since the number of cards left and cards to replace are all constant numbers (47 and 0-5) , our algorithm has a constant time complexity O(1) and the space complexity of O(1) overall.

$$\binom{n}{k} \times \binom{n-k}{k} = \frac{n!}{k!(n-k)!} \times \frac{(n-k)!}{k!((n-k)-k)!}$$

`generate_all_possible_discards(hand)` - $O(2^n)$ (Time-complexity)
This function generates a list of all possible discards that can be made using the current hand.
It begins by initialising a list (`discards`) which contains a single empty list. It iterates across the cards in the hand, each time initialising a second, empty list representing the subset of discards to be added to the total discard list for that loop. For each card in the hand, it creates new `subset` lists to add to the total list, by adding the card to each existing discard list. This is done with a nested loop - the outer loop iterates over each card in the hand, and the inner loop iterates over each subset element currently in the total `discards` list. For each subset in `discards`, a new subset is created by appending the current card to it. Each new subset is added to a temporary list called `subsets_to_add`. After processing all the subsets for the current card, `subsets_to_add` is added to the discard list.

`draw_combinations(deck, r)` - $O(n!)$
The draw combinations function is used to determine the list of possible draws from a `deck` that can be made to fill `r` number of gaps in the hand from discards. The length of the deck is assigned to `n`, and it initialises an empty list, `indices`. It then iterates `r` times, adding the first `r` indices to the list `indices`.
Next, it initialises an empty list, `combinations_list`, to store combinations of drawn cards.
It starts by creating a list of indices, `0` to `r-1`, which represents the first combination that may be drawn from the deck. It finds the cards corresponding to these indices and adds this as a list, `combinations`, to `combinations_list`.
Inside a while (boolean flag) loop, the function checks if the indices of the choices from the deck may be incremented without exceeding the conditions set, i.e. their maximum allowed values. If the indices have met their maximum value conditions, the loop is broken and the combination is added to the combinations list.
The function finds if index `i` that is being used to draw a card has passed its maximum value by iterating over the indexes in reverse order, and for each, checking if the value of `indices[i]` is equal to `i+n-r`, keeping maximum values for the list of indices, in ascending order of index; 48, 49, 50, 51, 52.
The loop stops at the first index (right-most) which has not reached its maximum allowed value.
It increments this index, and sets every index to the right in ascending order from it (eliminating multiple counting of combinations).

`simulate_new_hands(hand, discardSetList)` - $O(2^n)$
In this function, a dictionary is first initialised to store new hands for each discard set.
A new `Deck` object is created. A handSet python set object is created as `set(hand)` (where hand is the input current cards in the hand that may be discarded from).
The hand is iterated over and cards in that list are removed from the `newDeck` (via `remove()` method).
The `discardSetList` is iterated over. Each `discardSet` is converted to a python set object, the `hand_temp` list (`list(handSet - discardSet)`) is created, then, a for loop does the following:
Iterating over values (`new_cards`) generated by finding the combinations of cards in `newDeck.cards`, to fill a position count equal to `len(discardSet)`, a `new_hand` list is created, which is a concatenation of the `hand_temp` list and `list(new_cards)`.
A frozen set is created from the current discard set for use as a key. If the key is not currently in the `newHandsPerDiscardSet` dictionary, the entry is added with the key of the discard set and value `new_hand`.

If the key is already present in the dictionary, `new_hand` is appended to the value.

`discard_rank_count_calculator(handSetDict) - Returns discardRankCounts - O(n²)`
The function returns a dictionary containing the count of hand ranks that can be generated from each discard/draw combination.
Firstly, a dictionary `discardRankcounts` is initialised, empty. A for loop iterates across `handSetDict.items()`. The loop initialises an empty dictionary `discardRankCount[discardSet]`.
If the length of the `discardSet` is 5, then `discardRankCounts[discardSet]` is set equal to a dictionary containing a count for each rank corresponding to a draw of a full 5 cards after discarding 5 of a high card hand. The strategy of discarding 5 cards is unusually disadvantageous, and results are generally similar enough to be pre-calculated.
If not, for each hand in the `handSets`, the rank is found using `hand_rank` function, and incremented as count values in `discardRankCounts[discardSet][handValue]`.

`calculate_relative_values(rankCounts) - O(n²)`
Weights the counts of each type of hand rank generated by the `discard_rank_counts` function.
The function returns a dictionary containing a dictionary of the possible discards, with a weight assigned to them as a value, associated with the value of each rank and the proportion of the total possible generated hand count that is taken by each rank, or the likelihood of better ranks being available after each discard.

`calculate_relative_probabilities(rankCounts) - Returns rankCounts - O(n²)`
The outer for loop iterates across `rankCounts.items`, with key of `discard_set` and the values of the dictionaries `rankCountDiscard` (counts for each rank). A `total_hands` variable is assigned the sum of the `rankCountDiscard.values()` for that discard set (total number of hands that may be generated as a result of that discard). The inner for loop then iterates across `rankCountDiscard.items()`, getting the ratio of each count for each rank to the total count of possibly generated hands for that discard set, and adds them back as dictionary entries `rankCounts[discardSet][rank] = ratio`.

The functions are run in the following order (in **`run_best_discard()`**):

```
1. discards = generate_all_possible_discards(hand)
2. rank_counts = discard_rank_count_calculator(simulate_new_hands(hand, discards))
3. relative_values = calculate_relative_values(rank_counts)
4. optimalDiscard = max(relative_values, key=relative_values.get)
```

4. Algorithm and Data Structure Optimization

	Time Cost of `run_best_discard()` (ms)
Prior to optimization	79839
After optimization	23756
After parallelization	4000

4.1 Pruning

Discarding all five cards in a hand opens up the greatest number of possible combinations, calculated as C(47, 5) = 1,533,939. However, this approach yields results not significantly different from those obtained by drawing a fresh hand from a new deck, which involves C(52, 5) combinations. As such, using this tactic as a strategic play is generally less advantageous. By caching the outcomes of discarding a hand containing only high cards, the computational time can be substantially reduced.
Further optimization might be explored through strategies that utilise the symmetry across the four suits, although this path was not pursued due to the project's time constraints.

4.2 Frozenset

Tuples of cards were used as keys for the dictionaries. A significant amount of time was used in hashing these tuples. Shown in the profiling analysis call graph as `__hash__` (see appendix). By switching to frozenset. An immutable version of a Python set object that stores hash value instead of recalculating hash every time. Significantly increased time efficiency.

4.3 Parallelization

The function of `discard_rank_count_calculator()` consumed more than 90% of the processing time. This function evaluates all possible new hands of 32 discarding strategies. There are

C(5,4)*C(47,4)+C(5,3)*C(47,3)+C(5,2)*C(47,2)+C(5,1)*C(47,1) hands to evaluate after pre-calculating C(47,5). By using the multiprocessing `Pool` provided by python, the program uses processes equal to the amount of `cpu_count`. Parallelization decreased time cost from 23756 ms to 4000 ms. Which is within tolerable range of a game.

4.4 Memoization

The `hand_rank()` function is invoked frequently, with 2,598,960 calls per run, suggesting that memoizing its results could offer significant time savings. To this end, a `@memoize` decorator was applied. However, since each combination is unique during a single evaluation, memoization did not decrease the program's runtime every single run. Additionally, with the transition to multiprocessing, managing shared memory across processes proved overly complex for this project, leading to the decision to disable memoization for `hand_rank()`.

Despite these challenges, the concept of memoization remains appealing. As gameplay progresses, caching and reusing previously computed data could effectively trade extra memory usage for reduced computational time.

4. Appendix

See the detailed runtimes of each algorithm in our implementation before and after optimisation in the folder ‘Visualisation’.