Perfect-Information Blackjack

By

Aakarsh 3122215002001

Aasish 3122215002002

Abarna 3122215002003

Adharsh 3122215002004

Aditi 3122215002005

Ahamed 3122215002006

CLIENT: **Dr.R Srinivasan**

PROBLEM STATEMENT

- The problem statement is to develop a **web-based** blackjack game.
- The normal rules of blackjack must be implemented.
- Enhancing user interaction by **incorporating input functionality** to the game. This includes choosing the number of rounds to play, and what decisions to make when a round begins [Hit or Stand].

PROBLEM STATEMENT

- The implementation should provide users with the ability to utilise limited hints to assist them in making strategic moves during gameplay
- The result of each round must be displayed along with the updated scores.
- At the end of the game, the final winner must be declared along with the tally of scores.

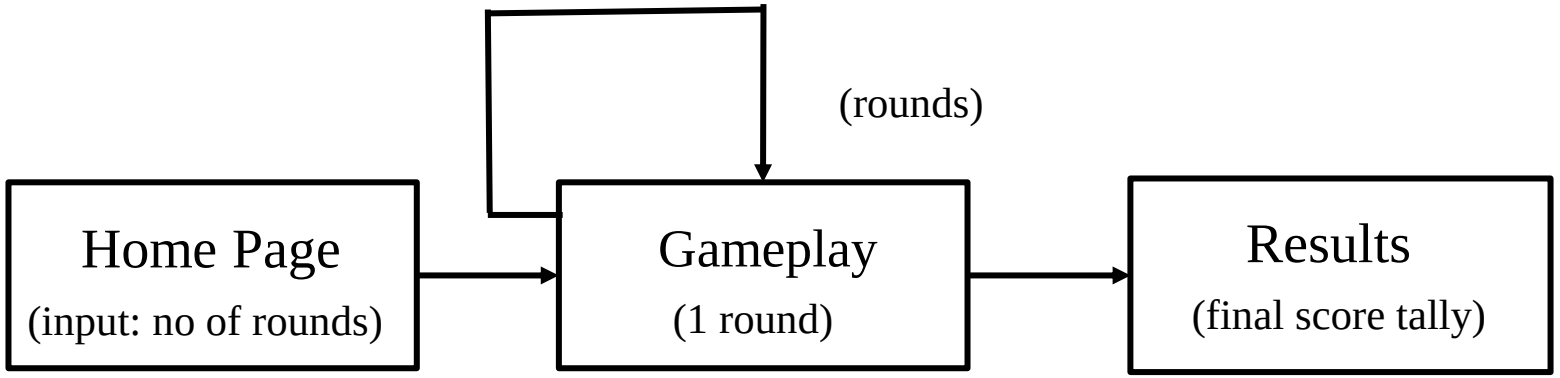
MOTIVATION

- **Algorithmic Strategies:** Optimizing code efficiency with Dynamic Programming and Greedy Method.

- **Interactive:** Making the web application interactive and easy to use for an enhanced gaming experience.

- **Visually Appealing:** Enhancing game aesthetics for an immersive experience.

SYSTEM DESIGN



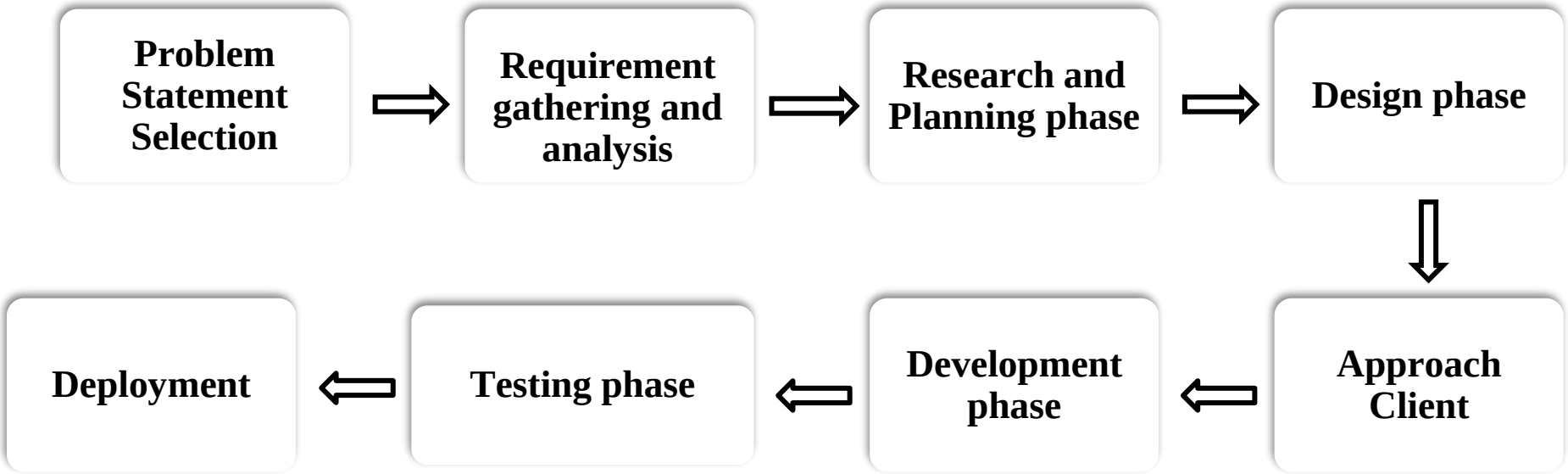
System design

DEVELOPMENT TOOLS

- **FRONTEND:**
  - **Flask Microframework:** To develop a web application using python.
  - **HTML:** To deploy items in the web page and integrate different components.
  - **CSS:** For Styling.

- **BACKEND:**
  - **Javascript:** Used to integrate HTML widgets with their functionality
  - **Python:** To implement the blackjack game abstractly using algorithms and data structures.The algorithms are
    - **Dynamic Programming:** Used to provide limited hints to the user as assistance during gameplay
    - **Greedy method:** For implementing the dealer functionality

PROCESS MANAGEMENT TIMELINE



RISK MANAGEMENT

Risk	Risk Description	Impact	Mitigation Plan
Disturbance in the deck of cards after each round	After each round, the order of the cards are to be preserved properly	It makes the array for best possible outcomes useless and disturbs the hint generation.	Using a single global variable with one shallow copy.
Un-optimal Hints	The hints provided should be optimal for the entire gameplay.	Improper hints will lead to degrading scores on using those hints.	Usage of copy element in the deck class

RISK MANAGEMENT

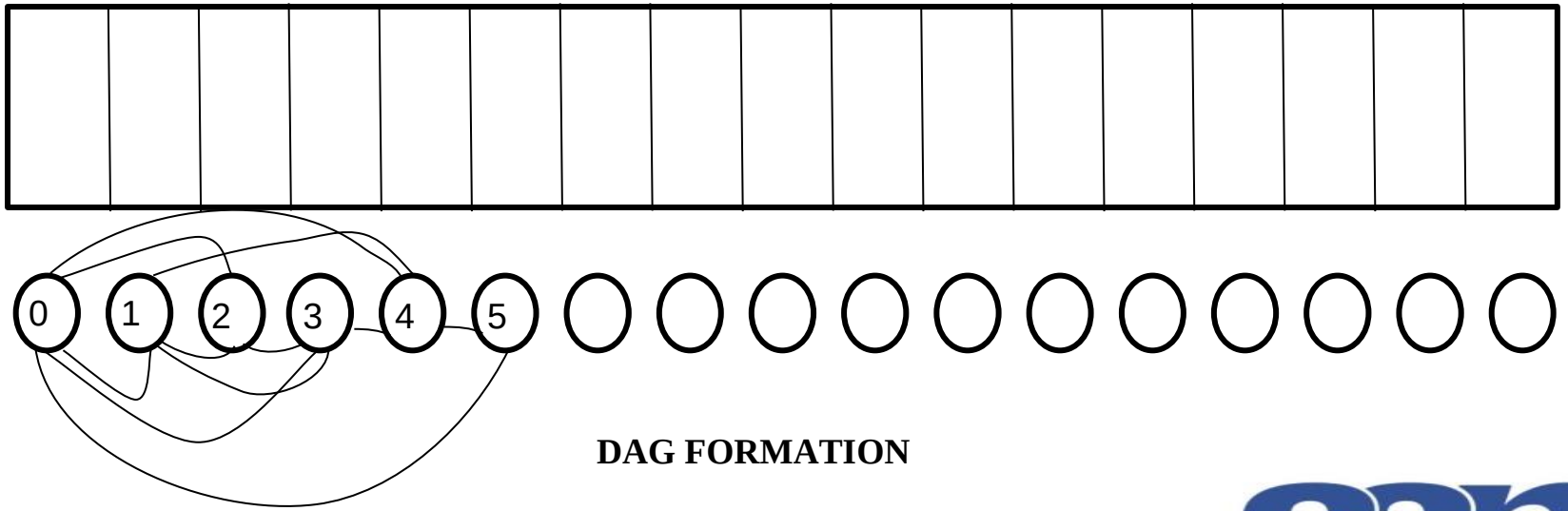
Risk	Risk description	Impact	Mitigation Plan
Separating current and user inputs	The overwriting of values between existing values in computation and user input values	Makes the application unstable, produces inconsistent results	Usage of separate post and get method in the flask routing methods
Error in log data	Data about player, rounds played, hints used, and deck of cards after each round have to be carried forward correctly.	Affects the smooth gameplay between each round, results in a bad user experience.	Usage of session dictionary in flask that creates a JSON serialized file during the runtime of the application

IMPLEMENTATION

- The player has an **x ray vision** of all the cards present in the deck.
- This gives the player a future vision on the **maximum winnings** he/she can make from the game and the **number of hits in total** to win the game.
- The only decision the player has to make is to hit or stand (depending on the total number of hits left)
- This can be done in two ways:
  - Topological sort using DFS
  - Dynamic Programming

Topological Sort DFS

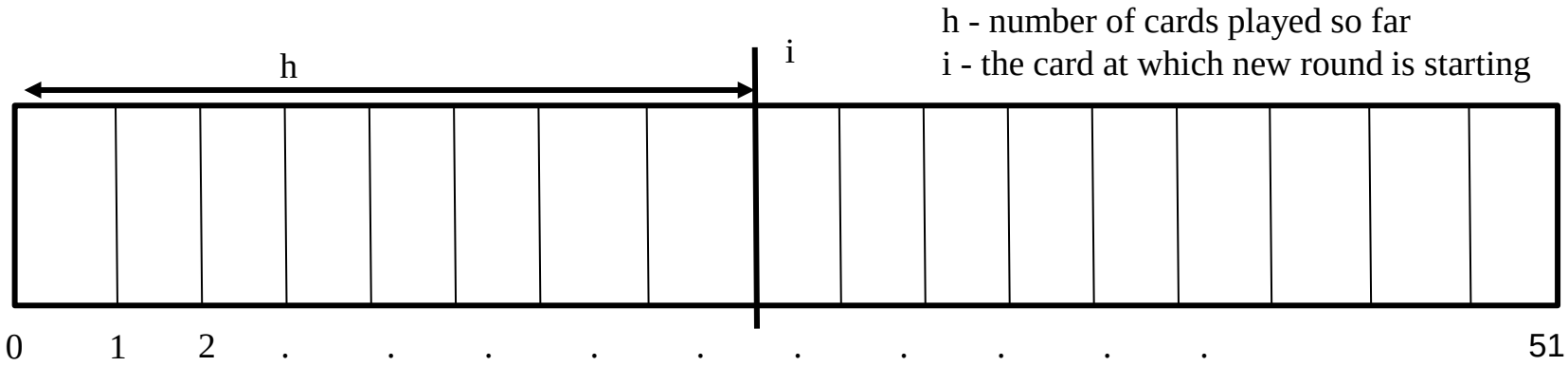
- The number of hits left and the score is represented in the form of a graph (DAG). This is then traversed to find the solution using Topological Sort DFS
- This has a higher time complexity and a more complex code.



DAG FORMATION

Dynamic Programming

- All the different player states are denoted by subproblems and all the cards in the deck are represented in the form of an array
- Code implementation is much easier and the time complexity is also reduced.



- Time complexity of **Topological Sort DFS**: O(V+E)
- Time complexity of **Dynamic Programming**: O(N)

❑ Data Structure Used: List

Class: Deck
-suits: list - ranks: list - number_of_decks: int - deck: list
+__init__(number_of_decks:int) + deal(num: int) → Union[tuple, list] + deal_copy(num: int) → Union[tuple, list] + __str__() → str + reset() +reset_copy()

Game.py

Functions	Time complexity
convert_card_names(cards)	O(N)
get_hand_value(hand)	O(N)
get_verdict(player_hand, house_hand)	O(N)
house_plays(deck, player_hand, house_hand)	O(N)
pick_winner(scores, r)	O(1)

dp.py

Functions	Time complexity
Calculate_max_score(deck,start,scores)	O(N)
get_score(starting,deck)	O(N^2)
Simulate(start,w,h)	O(N*W)

BACKEND IMPLEMENTATION

```
def get_score(starti,deck,seed=None):

    scores = [0 for _ in range(52)]  #keeps in track of all the scores the player can achieve
    #with the given starting index
    f= {}
    for start in range(48, -1, -1):
        #backward iteration is used as it follows a bottom up dynamic programming approach
        f[start]=(calculate_max_score(deck, start,scores))

    return scores[starti]
```

Get_scores takes a starting index and starts to **iterate backwards** from 49(52-4+1) till the first card which will be used to create a list that stores the most optimal scores at each position.
The calculate_max_score uses these starting index to find the optimal number of hits,and stores the result of each hits along with the end position which in turn will have an optimal path till the end. Then the most optimal solution i.e., the one with most overall winning rather than the one with more recent win is selected and returned and simulated

Simulation of the most optimal play results at a given index

```
====== RESTART: C:\Users\Lenovo\Desktop\ads project\blackjack-master\dp.
==
player's CARDS:
CARD:  10
CARD:  3
SUM:  13
dealer's CARDS:
CARD:  1
CARD:  6
CARD:  2
CARD:  5
CARD:  10
SUM:  24
Win

player's CARDS:
CARD:  1
CARD:  7
CARD:  6
CARD:  10
SUM:  24
dealer's CARDS:
CARD:  10
CARD:  4
CARD:  7
SUM:  21
Lost

player's CARDS:
CARD:  7
CARD:  9
CARD:  1
SUM:  17
dealer's CARDS:
CARD:  3
CARD:  6
CARD:  10
SUM:  19
Lost

player's CARDS:
CARD:  2
CARD:  10
SUM:  12
dealer's CARDS:
CARD:  10
CARD:  5
CARD:  10
SUM:  25
Win

player's CARDS:
CARD:  9
CARD:  8
CARD:  4
SUM:  21
dealer's CARDS:
CARD:  10
CARD:  6
CARD:  10
SUM:  26
Win
```

Calculate_max_score function:

```
def calculate_max_score(deck, start,scores):
    '''function for calculating the score '''
    player = Player(deck.deal(start,1))
    dealer = Player(deck.deal(start + 1, 1))
    player.deal(deck.deal(start+2,1))
    dealer.deal(deck.deal(start+3,1))
    results = []  #used for storing all possible scores of the player from the given deck of cards
    action_player = []
    for i in range(49 - start):  #iterating through all the remaining undealt cards
        count = start + 4 #gives the amount of cards that have already been dealt
        if count>52 or player.total>21:
            break
        for hits in range(i):
            player.deal(deck.deal(count, 1))  #equivalent to the hit function
            count += 1
            if count>52 or player.total>21:
                break
            if dealer.total < 17:
                dealer.deal(deck.deal(count,1))
                count+=1
        if player.total > 21:
                '''player gets busted and since this is undesirable, backtracking occures and the
                solution is broken'''
                results.append((-1, count,i))
        while dealer.total < 17 and count < 52:
                dealer.deal(deck.deal(count, 1))
                count += 1
        if dealer.total > 21:
                results.append((1, count,i))
                action_player.append(i)
        else:
            if player.total > dealer.total:
                results.append((1,count,i))  #1st element - decides if the player won or lost
                    #2nd element - gives the card index at which the game reached the end
                action_player.append(i)
            elif player.total < dealer.total:
                results.append((-1,count,i))
            else:
                results.append((0,count,i))
    options = [] #it stores all the scores possible for the player to get
    for score, next_start,hits in results:
        #score gives the cost of the winnings in the current round and the score that the player gets in the next game this is assuming that for each round won - +1
        options.append((score + scores[next_start][0] if next_start <= 48 else score,hits))
        #if the card index is 49 or 50, it indicates the end of the game, the score is simply the result of the current game the best score is considered
    scores[start] = options[0]
    for i in options:
        if scores[start][0]<i[0]:
            scores[start] = i
```

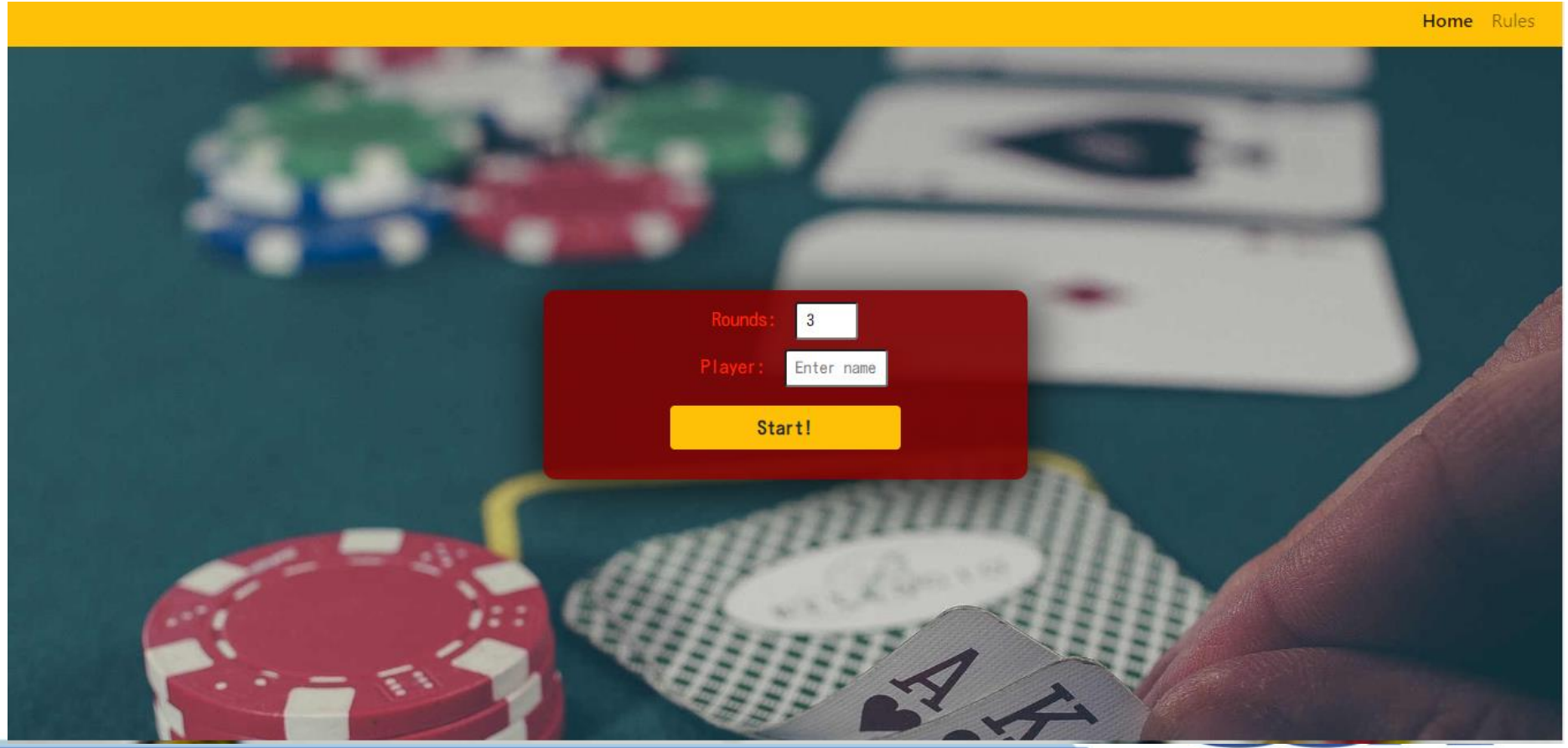
BACKEND IMPLEMENTATION

```
def hint(player_hand,house_hand,deck):


    p2 = copy.copy(player_hand)


    c1,s1 = get_hand_value(player_hand)
    c2,s2 = get_hand_value(p2)

    i = 0
    while c2<21:
        if i:
            for j in range(i):
                p2.append(deck.deal_copy())
        c2,s2 = get_hand_value(p2)
        print('\n',p2,'\n')
        if c2>21:
            deck.reset_copy()
            return "Can't win this round"
        house_hand = deck.deal_copy(2)
        cp,sp = get_hand_value(house_hand)
        while cp<17:
            house_hand.append(deck.deal_copy())
            cp,sp = get_hand_value(house_hand)
        print(house_hand)
        if cp>21 or c2>cp:
            deck.reset_copy()
            return i
        deck.reset_copy()
        p2 = copy.copy(player_hand)
        i+=1
    return "Cannot win"
```

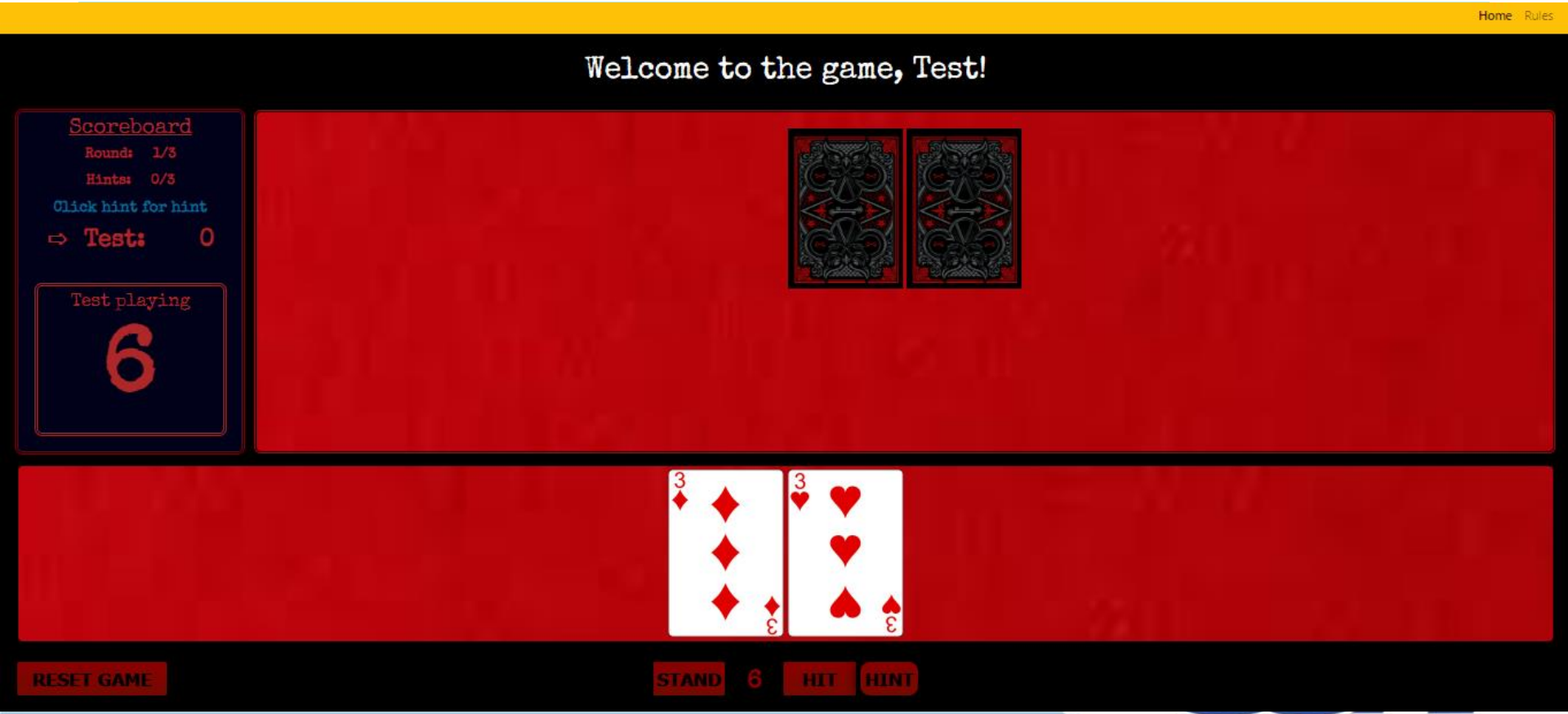
The hint function utilizing the information of the number of hits from the dynamic programming simulation and verifying through the state space for optimal solution and returning the exact number of hits that round.

```
========== RESTART: C:\Users\Lenovo\Downloads\hdd.py ==========
[('clubs', 'J'), ('diamonds', '4')] []

 [('clubs', 'J'), ('diamonds', '4')]

[('spades', '4'), ('hearts', 'K'), ('hearts', '3')]

 [('clubs', 'J'), ('diamonds', '4'), ('spades', '4')]

[('hearts', 'K'), ('hearts', '3'), ('spades', 'J')]
1
[('clubs', 'J'), ('diamonds', '4')] []
```

FRONTEND IMPLEMENTATION



FRONTEND IMPLEMENTATION

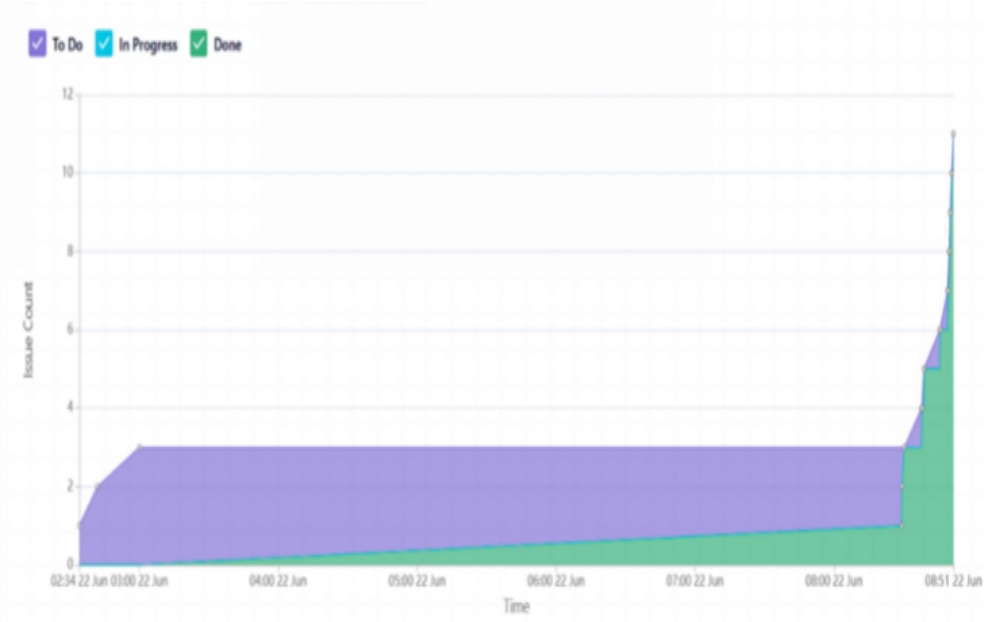


CASE 1: DEALER WINS

CASE 2: PLAYER WINS

PROCESS MANAGEMENT (JIRA)

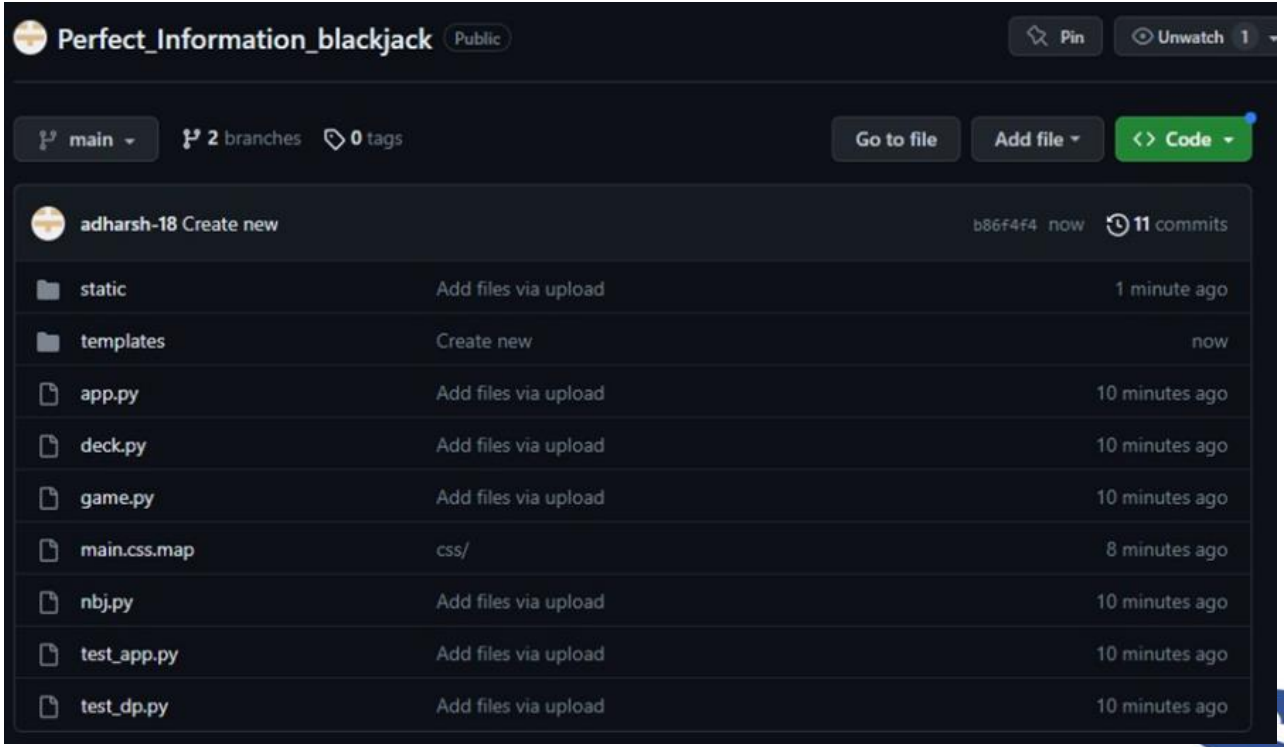
CUMULATIVE FLOW DIAGRAM



ISSUES

- A mechanism to transition to the final page declaring the winner and the scores, whenever the dealer or user loses
  PIB-10 · Perfect Information Blackjack
- A mechanism to allow the user to hit or stand
  PIB-9 · Perfect Information Blackjack
- A method that allows distribution of cards from the deck to the dealer and the user, as well as calculating the total sum of the player
  PIB-8 · Perfect Information Blackjack
- Assigning values to each card in the deck, as well as assigning it a symbol and the suit it belongs to
  PIB-7 · Perfect Information Blackjack
- Creating an interface displaying the main gameplay.
  PIB-6 · Perfect Information Blackjack
- Creating a page explaining/illustrating the rules of the game
  PIB-5 · Perfect Information Blackjack
- Creating an interface to display the main screen of the web application which collects the player's name and no of rounds
  PIB-4 · Perfect Information Blackjack
- Approach Client
  PIB-3 · Perfect Information Blackjack
- Collecting requirements and evaluate risks
  PIB-2 · Perfect Information Blackjack
- RESEARCH on Blackjack and perfect information blackjack
  PIB-1 · Perfect Information Blackjack

PROJECT MANAGEMENT (GITHUB)



SCOPE AND LIMITATIONS

- **SCOPE:**
  - **Visually appealing:** Developing an intuitive and visually appealing user interface for playing blackjack.
  - **Interactibility:** The project aims on deriving an optimized algorithm that can be used to maximize winnings in blackjack.
  - **Game Hints:** Limited game hints are offered to the player, to increase the interactibility as well as help the player make strategic moves.

- **LIMITATIONS:**
  - **Slow response time:** When a hint is utilised, since all possible solutions are explored (state space search), this will involve a higher time complexity to search for the optimal solution, thus resulting in a slower response time.
  - **Single-player constraint:** Only one user can play the game at a time.
  - **Lack of database:** Since there is no database connected to the web application, the user cannot save their game progress at any point in time, nor view their game history.

FUTURE SCOPE

- The project can be built further using javascript for a more interactive interface.
- A database can be added to the existing application for the user to save their game progress.
- Multiplayer functionality can also be added to enhance the gaming experience.

REFERENCES

- **MIT:**
  https://courses.csail.mit.edu/6.006/fall11/rec/rec20.pdf
  https://www.youtube.com/watch?v=jZbkToeNK2g

- **Washington Post:**
  https://games.washingtonpost.com/games/blackjack

- **Byrne, Donal.** "Learning To Win Blackjack with Dynamic Programming Methods." Medium, Towards Algorithms, 7 Nov. 2018.