Project 2

UNO Card Game With Graphs and Trees

CIS-17C

47065

June 6, 2023

David Guadian

1 Introduction

The card game UNO has been around since 1992 it was been the heart of many parties and family game nights. UNO is a game that can be enjoyed no matter what age and is why I decided to program the game of UNO. Also UNO being one of the games I most have experience playing is the reason I choose this project to work on.

Number of Classes	6
Number of comments	66
Number of Lines	564

This project took around 40 hours with an estimated 5 hours planning out the project and the rest programming and working through issues within the project.

Project Locations: https://github.com/dguad54/CIS-17C_Project-2

2 Concepts

Recursion: This program uses recursion within the playTurn method from the UNOGame class. This method uses recursion by calling playTurn after a valid move have been made by the current play which then plays the turn of the following player. This recursve call continues until the cards on the current play runs out and that player is named the winner.

Merge sort: The mergeSort function takes a reference to a vector of Card objects (cards).

It checks the base case: if the size of the cards vector is less than or equal to 1, the vector is already sorted, so the function returns without performing any further operations.

If the base case is not met, the vector is divided into two halves: left and right.

The midpoint mid is calculated as the size of cards divided by 2.

The elements from the beginning of cards up to mid are copied to the left vector.

The elements from mid to the end of cards are copied to the right vector.

The mergeSort function is recursively called for both the left and right vectors. This step ensures that each sub-vector is sorted.

After the recursive calls to mergeSort complete, the left and right vectors are sorted individually.

The sorted halves are then merged back into the original cards vector.

Three index variables are used: i for the left half, j for the right half, and k for the merged vector.

Starting with i and j at 0, the function compares the values of the elements at left[i] and right[j].

The smaller value is copied to cards[k].

The corresponding index (i or j) is incremented, and k is incremented.

This process is repeated until either the left or right vector is fully copied into cards.

If there are any remaining elements in the left or right vector, they are copied to the cards vector.

If there are elements remaining in left, they are copied starting from left[i].

If there are elements remaining in right, they are copied starting from right[j].

In both cases, the indices i, j, and k are incremented accordingly.

The sorting is complete, and the sorted cards vector is returned.

In the dealCards function, the mergeSort function is called for each player's cards to sort them after being dealt from the deck.

The merge sort algorithm provides an efficient way to sort the player's cards, ensuring that they are in ascending order based on their values.

The mergeSort function takes a reference to a vector of Card objects (cards).

It checks the base case: if the size of the cards vector is less than or equal to 1, the vector is already sorted, so the function returns without performing any further operations.

If the base case is not met, the vector is divided into two halves: left and right.

The midpoint mid is calculated as the size of cards divided by 2.

The elements from the beginning of cards up to mid are copied to the left vector.

The elements from mid to the end of cards are copied to the right vector.

The mergeSort function is recursively called for both the left and right vectors. This step ensures that each sub-vector is sorted.

After the recursive calls to mergeSort complete, the left and right vectors are sorted individually.

The sorted halves are then merged back into the original cards vector.

Three index variables are used: i for the left half, j for the right half, and k for the merged vector.

Starting with i and j at 0, the function compares the values of the elements at left[i] and right[j].

The smaller value is copied to cards[k].

The corresponding index (i or j) is incremented, and k is incremented.

This process is repeated until either the left or right vector is fully copied into cards.

If there are any remaining elements in the left or right vector, they are copied to the cards vector.

If there are elements remaining in left, they are copied starting from left[i].

If there are elements remaining in right, they are copied starting from right[j].

In both cases, the indices i, j, and k are incremented accordingly.

The sorting is complete, and the sorted cards vector is returned.

In the dealCards function, the mergeSort function is called for each player's cards to sort them after being dealt from the deck.

The merge sort algorithm provides an efficient way to sort the player's cards, ensuring that they are in ascending order based on their values.

Hashing:

The hashFunction takes a reference to a const string parameter named key.

It initializes a variable hash of type size_t to 0.

The function iterates over each character c in the key string.

For each character, the ASCII value of c is added to the hash variable.

Finally, the function returns the calculated hash value.

hashCard:

This function takes a reference to a const Card parameter named card.

Inside the function, a cardString is constructed by concatenating the color and value fields of the card object.

The cardString is then passed to the hashFunction to calculate the hash value.

The calculated hash value is returned.

The purpose of these hash functions is to generate a unique hash value for each card based on its color and value. This can be useful for various purposes, such as indexing or storing cards in a hash table or other data structures where efficient lookup is required. By using the hashFunction and hashCard functions, cards can be uniquely identified and manipulated based on their hash values.

Trees:

The tree is created and initialized in the UNOGame constructor:

The currentNode member variable is assigned the root node of the tree using the getRoot() function of the gameTree object.

The tree is utilized in the playTurn() function:

Inside the playTurn() function, after a player has made a valid move, the code creates a new node for the next player using the createNode() function of the gameTree object.

The newly created node is added as a child to the currentNode using the push_back() function.

The currentNode is then updated to be the newly created node for the next player's turn.

The tree structure allows for the creation of a game tree, where each node represents a player's turn in the game. The currentNode keeps track of the current node in the tree during the game, and new nodes are created and added as children to represent subsequent player turns. This allows for tracking and branching of the game's progress and enables the ability to navigate and analyze different paths and outcomes in the game tree.

Graphs:

The graph is constructed in the main() function:

The Graph object is instantiated as unoGraph.

UNO cards are added to the graph using the addEdge() function, where each vertex represents a color and each edge represents a card within that color.

The graph is then displayed using the displayGraph() function.

Graph Traversal: BFS and DFS

The graph class provides two traversal algorithms: Breadth-First Search (BFS) and Depth-First Search (DFS).

BFS is implemented in the BFS() function, and DFS is implemented in the DFS() function.

Both algorithms take a start vertex and a target card as parameters.

The algorithms use a queue (for BFS) or a stack (for DFS) to keep track of the vertices to visit.

The graph is traversed by exploring adjacent cards of the current vertex.

If the target card is found during the traversal, a message is printed, and the function returns true.

If the target card is not found after traversing the entire graph, a corresponding message is printed, and the function returns false.

In the main() function, the user has the option to enable or disable graph traversal by entering 1 or 0, respectively. If enabled, the program creates the graph, displays it, and prompts the user to enter a target card. Then, it performs a BFS and DFS to find the target card within the graph.

After the graph traversal section, the code proceeds to initialize an UNOGame object (unoGame) and calls the playGame() function to start playing the UNO game.

In summary, the Graph class in this code represents a simple undirected graph, and the provided code demonstrates the usage of graph traversal algorithms (BFS and DFS) to search for a target card within the graph.

3 Classes

Graph.cpp:

The Graph class contains the following member functions:

addEdge(const string& vertex, const vector<UNOCard>& cards): This function adds an edge to the graph by associating a vertex (represented by a string) with a vector of UNO cards. It updates the adjacency list adjList with the provided vertex and cards.

displayGraph(): This function displays the graph by iterating over each vertex and its associated cards in the adjList. It prints the vertex followed by the UNO cards associated with it.

BFS(const string& startVertex, const UNOCard& targetCard): This function performs a Breadth-First Search (BFS) traversal of the graph starting from the startVertex. It searches for a target card within the graph by comparing the color and number of each card with the target card. If the target card is found, it prints a message and returns true. If the target card is not found, it prints a corresponding message and returns false.

DFS(const string& startVertex, const UNOCard& targetCard): This function performs a Depth-First Search (DFS) traversal of the graph starting from the startVertex. It searches for a target card within the graph by comparing the color and number of each card with the target card. If the target card is found, it prints a message and returns true. If the target card is not found, it prints a corresponding message and returns false.

Tree.cpp:

The Tree class contains the following member functions:

Tree(): This is the constructor of the Tree class. It initializes the root of the tree to nullptr.

TreeNode* createNode(const string& winnerName): This function creates a new tree node and assigns the provided winnerName to it. It returns the newly created node.

void insert(TreeNode* parent, TreeNode* child): This function inserts a child node into the children vector of a given parent node.

void setRoot(TreeNode* node): This function sets the root of the tree to the provided node.

TreeNode* getRoot(): This function returns the root of the tree.

void displayTree(TreeNode* node, int level): This function recursively displays the tree starting from the provided node. It prints each node's winner value, indented based on its level in the tree, and uses a vertical line symbol (|_) to denote the hierarchy of the tree. It traverses each child node of the current node and recursively calls the displayTree function to display the subtree rooted at that child.

UNOGame.cpp:

The UNOGame class contains the following member functions:

UNOGame(): This is the constructor of the UNOGame class. It initializes various member variables, such as currentPlayer, currentNode, reverse, and calls the initializeDeck() and initializePlayers() functions.

size_t hashFunction(const string& key): This function calculates the hash value for a given key using a simple hash function that sums up the ASCII values of its characters. It returns the calculated hash value.

size_t hashCard(const Card& card): This function generates a hash value for a given card by concatenating its color and value properties and then passing it to the hashFunction function.

void initializeDeck(): This function initializes the deck vector with UNO cards. It creates cards with different colors and values, including special cards such as "Wild" and "Wild Draw Four".

void mergeSort(vector<Card>& cards): This function implements the merge sort algorithm to sort the given cards vector in ascending order based on their value property.

void initializePlayers(): This function prompts the user to enter the number of players and their names. It initializes the players map with the player names and assigns each player a score of 0.

void shuffleDeck(): This function shuffles the deck vector using the Fisher-Yates algorithm, ensuring a random order of cards.

void dealCards(): This function distributes cards from the deck to each player in a round-robin fashion. Each player receives 7 cards, which are sorted using the mergeSort function.

void displayHand(const string& playerName): This function displays the hand of a given player by iterating through their cards and printing their color and value.

bool hasValidMove(const string& playerName): This function checks if a player has a valid move in their hand. It iterates through the player's cards and checks if any card can be played based on the current top card in the discardPile.

bool isValidMove(const Card& card): This function checks if a given card can be played based on the current top card in the discardPile. It compares the colors and values of the cards to determine if the move is valid.

void updateCurrentPlayer(): This function updates the currentPlayer variable to determine the next player's turn. It considers the direction of play (reverse) and adjusts the player index accordingly.

void playTurn(): This function represents a player's turn in the game. It displays the current player's hand, prompts them to select a card to play, validates the selected card, updates the game state accordingly, and recursively calls playTurn() for the next player's turn.

void updateScore(const string& playerName, const Card& card): This function updates a player's score based on the card they played. It checks the color and value of the card and increments the score accordingly.

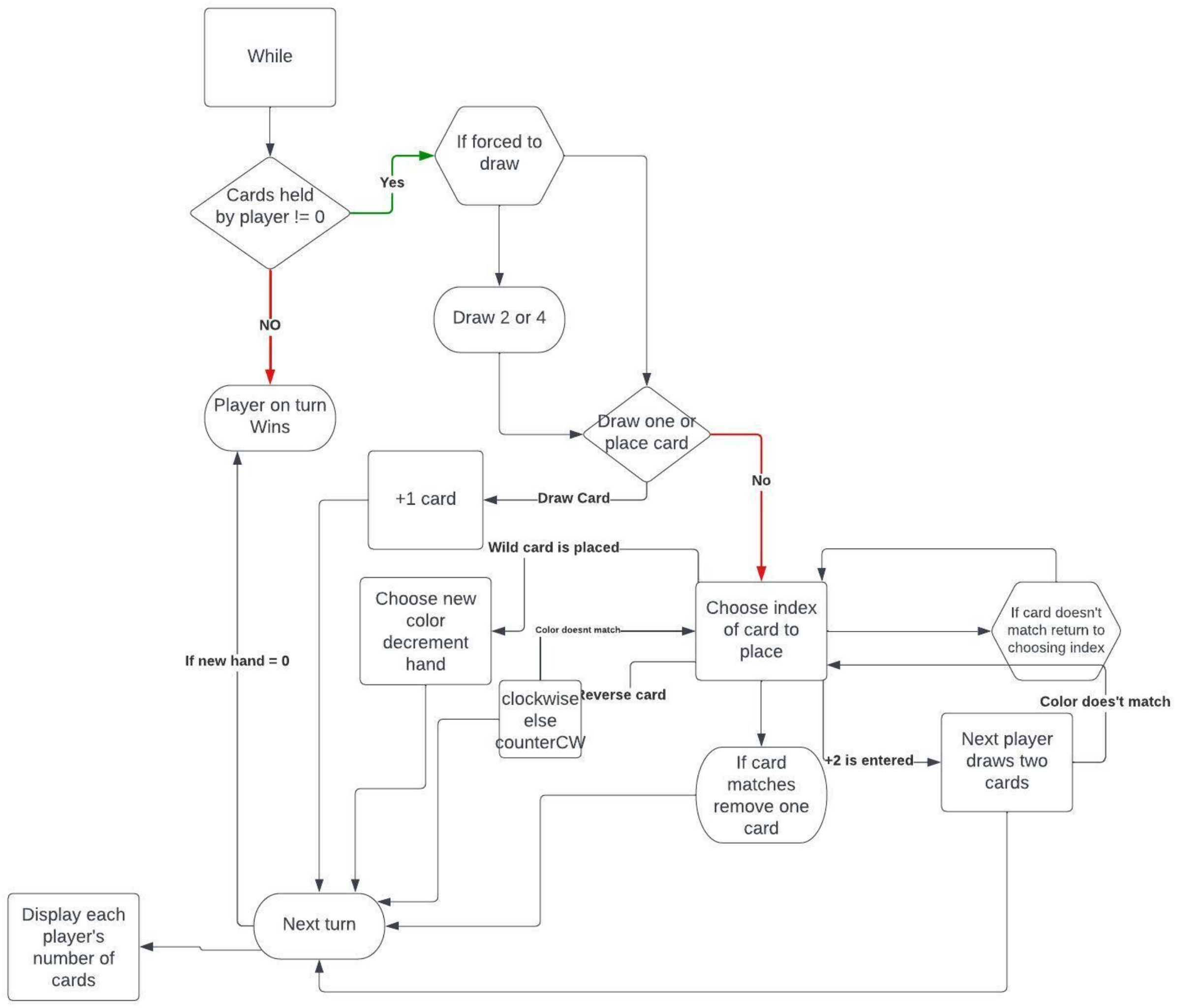
void drawCards(const string& playerName, int numCards): This function allows a player to draw a specified number of cards from the deck and adds them to their hand.

string getNextPlayer(): This function determines the next player's name based on the current player's index and the direction of play (reverse).

string getPlayerName(int playerIndex): This function retrieves the name of a player based on their index in the players map.

void playGame(): This function starts the UNO game by shuffling the deck, dealing cards to players, and calling playTurn() for the first player's turn. After the game finishes, it displays the scores of all players.

4 Game Logic


While
Cards held by player != 0
Yes
If forced to draw
Draw 2 or 4
NO
Player on turn Wins
Draw one or place card
No
Draw Card
+1 card
Wild card is placed
Choose new color decrement hand
Color doesnt match
Choose index of card to place
If card doesn't match return to choosing index
If new hand = 0
clockwise else counterCW
Reverse card
Color does't match
If card matches remove one card
+2 is entered
Next player draws two cards
Display each player's number of cards
Next turn

5 Pseudo-Code

UNOCard:

// Declare that this header file should only be included once during compilation

// Include the necessary library for string manipulation

#include <string>

```
// Use the standard namespace
using namespace std;

// Define a structure for representing UNO cards
struct UNOCard {
    // Declare the properties of a UNO card
    string color;
    int number;

    // Define the constructor for initializing a UNO card
    UNOCard(string c, int n) {
        color = c;
        number = n;
    }
}
```

Graph class:

```
// Include the necessary header file for the graph
#include "Graph.h"

// Include the necessary library for input/output operations
#include <iostream>

// Use the standard namespace
using namespace std;

// Define the function to add an edge to the graph
void Graph::addEdge(const string& vertex, const vector<UNOCard>& cards) {
    // Assign the vector of cards to the vertex in the adjacency list
    adjList[vertex] = cards;
}

// Define the function to display the graph
void Graph::displayGraph() {
    // Iterate over each vertex and its associated cards in the adjacency list
    for (const auto& pair : adjList) {
        // Get the vertex and cards from the pair
        const string& vertex = pair.first;
        const vector<UNOCard>& cards = pair.second;

        // Print the vertex and its associated cards
        cout << vertex << ": ";
        for (const auto& card : cards) {
            cout << "[" << card.color << " " << card.number << "] ";
        }
        cout << endl;
    }
}

// Define the breadth-first search (BFS) algorithm
bool Graph::BFS(const string& startVertex, const UNOCard& targetCard) {
    // Create a map to track visited vertices
    unordered_map<string, bool> visited;
    // Create a queue to perform BFS traversal
    queue<string> q;
    // Push the start vertex to the queue and mark it as visited
    q.push(startVertex);
    visited[startVertex] = true;
    // Perform BFS until the queue is empty
    while (!q.empty()) {
        // Get the current vertex from the front of the queue
        string currentVertex = q.front();
        q.pop();

        // Iterate over each card associated with the current vertex
        for (const auto& card : adjList[currentVertex]) {
            // Check if the card matches the target card
            if (card.color == targetCard.color && card.number == targetCard.number) {
                cout << "Found card [" << card.color << " " << card.number << "] using BFS!" << endl;
                return true;
            }

            // If the color of the card has not been visited, push it to the queue and mark it as visited
            if (!visited[card.color]) {
                q.push(card.color);
                visited[card.color] = true;
            }
        }
    }

    cout << "Card not found using BFS!" << endl;
    return false;
}

// Define the depth-first search (DFS) algorithm
bool Graph::DFS(const string& startVertex, const UNOCard& targetCard) {
```

```
// Create a map to track visited vertices
unordered_map<string, bool> visited;
// Create a stack to perform DFS traversal
stack<string> s;
// Push the start vertex to the stack and mark it as visited
s.push(startVertex);
visited[startVertex] = true;

// Perform DFS until the stack is empty
while (!s.empty()) {
  // Get the current vertex from the top of the stack
  string currentVertex = s.top();
  s.pop();

  // Iterate over each card associated with the current vertex
  for (const auto& card : adjList[currentVertex]) {
    // Check if the card matches the target card
    if (card.color == targetCard.color && card.number == targetCard.number) {
      cout << "Found card [" << card.color << " " << card.number << "] using DFS!" << endl;
      return true;
    }

    // If the color of the card has not been visited, push it to the stack and mark it as visited
    if (!visited[card.color]) {
      s.push(card.color);
      visited[card.color] = true;
    }
  }
}
```

```
    cout << "Card not found using DFS!" << endl;
    return false;
}
```

Tree Class:

```
// Include the necessary header file for the tree
#include "Tree.h"

// Include the necessary library for input/output operations
#include <iostream>

// Use the standard namespace
using namespace std;

// Define the default constructor for the Tree class
Tree::Tree() {
    root = nullptr;
}

// Define the function to create a new tree node
TreeNode* Tree::createNode(const string& winnerName) {
    // Allocate memory for a new tree node
    TreeNode* newNode = new TreeNode;
    // Set the winner name for the node
    newNode->winner = winnerName;
    return newNode;
}
```

```
// Define the function to insert a child node under a parent node
void Tree::insert(TreeNode* parent, TreeNode* child) {
    // Add the child node to the list of children of the parent node
    parent->children.push_back(child);
}

// Define the function to set the root of the tree
void Tree::setRoot(TreeNode* node) {
    // Set the given node as the root of the tree
    root = node;
}

// Define the function to get the root of the tree
TreeNode* Tree::getRoot() {
    // Return the root of the tree
    return root;
}

// Define the function to display the tree in a hierarchical format
void Tree::displayTree(TreeNode* node, int level) {
    // Check if the current node is nullptr (base case)
    if (node == nullptr) {
        return;
    }

    // Print indentation based on the level in the tree
    for (int i = 0; i < level; i++) {
        cout << "  ";
    }
    // Print the winner name of the current node
    cout << "|_" << node->winner << endl;

    // Recursively display the children of the current node
    for (TreeNode* child : node->children) {
        displayTree(child, level + 1);
    }
}
```

UNOGame class:

```
// Include the necessary header file for the UNOGame
#include "UNOGame.h"

// Include the necessary libraries
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cstdlib>
#include <random>

// Use the standard namespace
using namespace std;

// Define the default constructor for the UNOGame class
UNOGame::UNOGame() {
    currentPlayer = 0;
    currentNode = gameTree.getRoot();
    reverse = false;
    initializeDeck();
    initializePlayers();
}

// Define the hash function for a key
size_t UNOGame::hashFunction(const string& key) {
    size_t hash = 0;
    for (char c : key) {
        hash += c;
    }
    return hash;
}

// Define the hash function for a card
size_t UNOGame::hashCard(const Card& card) {
    string cardString = card.color + card.value;
    return hashFunction(cardString);
}

// Define the function to initialize the deck with UNO cards
void UNOGame::initializeDeck() {
    vector<string> colors = { "Red", "Blue", "Green", "Yellow" };
    vector<string> values = { "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "Skip", "Reverse", "Draw Two" };

    // Create cards for each color and value combination
    for (const string& color : colors) {
        for (const string& value : values) {
            Card card;
            card.color = color;
            card.value = value;

            deck.push_back(card);

        }

    }


    // Add wild cards and wild draw four cards

    for (int i = 0; i < 4; i++) {

        Card wildCard;

        wildCard.color = "Wild";

        wildCard.value = "Wild";

        deck.push_back(wildCard);


        Card wildDrawFourCard;

        wildDrawFourCard.color = "Wild";

        wildDrawFourCard.value = "Wild Draw Four";

        deck.push_back(wildDrawFourCard);

    }

}


// Define the merge sort algorithm to sort the cards

void UNOGame::mergeSort(vector<Card>& cards) {

    if (cards.size() <= 1) {

        return;

    }


    // Divide the vector into two halves

    int mid = cards.size() / 2;

    vector<Card> left(cards.begin(), cards.begin() + mid);

    vector<Card> right(cards.begin() + mid, cards.end());
```

```
// Recursively sort the two halves
mergeSort(left);
mergeSort(right);

// Merge the sorted halves
int i = 0; // Index for the left half
int j = 0; // Index for the right half
int k = 0; // Index for the merged vector

while (i < left.size() && j < right.size()) {
    if (left[i].value <= right[j].value) {
        cards[k] = left[i];
        i++;
    }
    else {
        cards[k] = right[j];
        j++;
    }
    k++;
}

// Copy the remaining elements from the left half
while (i < left.size()) {
    cards[k] = left[i];
    i++;
    k++;
}
```

```
    // Copy the remaining elements from the right half
    while (j < right.size()) {
        cards[k] = right[j];
        j++;
        k++;
    }
}

// Define the function to initialize players and their names
void UNOGame::initializePlayers() {
    int numPlayers;
    cout << "Enter the number of players: ";
    cin >> numPlayers;

    // Get the names of each player
    for (int i = 1; i <= numPlayers; i++) {
        string playerName;
        cout << "Enter player " << i << " name:" << endl;
        cin >> playerName;
        players[playerName] = 0;
    }
}

// Define the function to shuffle the deck of cards
void UNOGame::shuffleDeck() {
    random_device rd;
    mt19937 g(rd());
    shuffle(deck.begin(), deck.end(), g);
}
```

```
// Define the function to deal cards to each player
void UNOGame::dealCards() {
   for (auto& player : players) {
      for (int i = 0; i < 7; i++) {
         playerCards[player.first].push_back(deck.back());
         deck.pop_back();
      }

      mergeSort(playerCards[player.first]);  // Sort the player's cards
   }
}

// Define the function to display a player's hand
void UNOGame::displayHand(const string& playerName) {
   cout << playerName << "'s Hand: ";
   for (const Card& card : playerCards[playerName]) {
      cout << "[" << card.color << " " << card.value << "] ";
   }
   cout << endl;
}

// Define the function to check if a player has a valid move
bool UNOGame::hasValidMove(const string& playerName) {
   for (const Card& card : playerCards[playerName]) {
      if (isValidMove(card)) {
         return true;
      }
   }
```

```
    return false;
}

// Define the function to check if a move is valid
bool UNOGame::isValidMove(const Card& card) {
    const Card& topCard = discardPile.back();
    if (card.color == "Wild" || card.color == topCard.color || card.value == topCard.value) {
        return true;
    }
    return false;
}

// Define the function to update the current player after a turn
void UNOGame::updateCurrentPlayer() {
    int numPlayers = players.size();
    if (reverse) {
        currentPlayer--;
        if (currentPlayer < 0) {
            currentPlayer = numPlayers - 1;
        }
    }
    else {
        currentPlayer++;
        if (currentPlayer >= numPlayers) {
            currentPlayer = 0;
        }
    }
}
```

```
// Define the function to play a turn
void UNOGame::playTurn() {
    const string& currentPlayerName = getPlayerName(currentPlayer);
    cout << "Current Player: " << currentPlayerName << endl;
    displayHand(currentPlayerName);

    bool validMove = false;
    while (!validMove) {
        int cardIndex;
        cout << "Enter the index of the card to play (0-" << playerCards[currentPlayerName].size() - 1 << "):
";
        cin >> cardIndex;

        if (cardIndex >= 0 && cardIndex < playerCards[currentPlayerName].size()) {
            const Card& selectedCard = playerCards[currentPlayerName][cardIndex];
            if (isValidMove(selectedCard)) {
                discardPile.push_back(selectedCard);
                playerCards[currentPlayerName].erase(playerCards[currentPlayerName].begin() + cardIndex);
                validMove = true;
                updateScore(currentPlayerName, selectedCard);
                cout << currentPlayerName << " played [" << selectedCard.color << " " << selectedCard.value <<
"]" << endl;

                if (playerCards[currentPlayerName].empty()) {
                    cout << currentPlayerName << " has no cards left. They win!" << endl;
                    currentNode->children.push_back(gameTree.createNode(currentPlayerName));
                    return;
                }

                if (selectedCard.value == "Reverse") {
```

```
            reverse = !reverse;
        }
        else if (selectedCard.value == "Skip") {
            updateCurrentPlayer();
        }
        else if (selectedCard.value == "Draw Two") {
            string nextPlayer = getNextPlayer();
            drawCards(nextPlayer, 2);
        }
        else if (selectedCard.value == "Wild Draw Four") {
            string nextPlayer = getNextPlayer();
            drawCards(nextPlayer, 4);
        }
        else if (selectedCard.color == "Wild") {
            string newColor;
            cout << "Enter the new color (Red, Blue, Green, Yellow): ";
            cin >> newColor;
            selectedCard.color = newColor;
        }

        string nextPlayer = getNextPlayer();
        currentNode->children.push_back(gameTree.createNode(nextPlayer));
        currentNode = currentNode->children.back();
        playTurn();  // Recursively call playTurn for the next player's turn
        return;
    }
    else {
        cout << "Invalid move. Please select a valid card." << endl;
    }
```

```
        }
        else {
            cout << "Invalid index. Please select a valid card." << endl;
        }
    }
}

// Define the function to update a player's score based on the played card
void UNOGame::updateScore(const string& playerName, const Card& card) {
    int& score = players[playerName];
    if (card.color == "Wild" || card.color == "Wild Draw Four") {
        score += 50;
    }
    else if (card.value == "Draw Two" || card.value == "Reverse" || card.value == "Skip") {
        score += 20;
    }
    else {
        score += stoi(card.value);
    }
}

// Define the function to draw cards for a player
void UNOGame::drawCards(const string& playerName, int numCards) {
    for (int i = 0; i < numCards; i++) {
        playerCards[playerName].push_back(deck.back());
        deck.pop_back();
    }
}
```

```
// Define the function to determine the next player
string UNOGame::getNextPlayer() {
    int numPlayers = players.size();
    int nextPlayer;
    if (reverse) {
        nextPlayer = currentPlayer - 1;
        if (nextPlayer < 0) {
            nextPlayer = numPlayers - 1;
        }
    }
    else {
        nextPlayer = currentPlayer + 1;
        if (nextPlayer >= numPlayers) {
            nextPlayer = 0;
        }
    }
    return getPlayerName(nextPlayer);
}

// Define the function to get a player's name based on their index
string UNOGame::getPlayerName(int playerIndex) {
    for (const auto& player : players) {
        if (player.second == playerIndex) {
            return player.first;
        }
    }
    return "";
}
```

```
// Define the function to play the UNO game
void UNOGame::playGame() {
    shuffleDeck();
    dealCards();

    std::cout << "Game Started!" << std::endl;

    playTurn();  // Start the game by calling playTurn for the first player's turn

    cout << "------------------------" << endl;
    cout << "Scores:" << endl;
    for (const auto& player : players) {
        cout << player.first << ": " << player.second << endl;
    }
    cout << "------------------------" << endl;
}
```

Main.cpp:

```
int main() {
    int input;

    cout << "Welcome to UNO Game!" << endl;
    cout << "Do you wish to view graph traversal? (1 for yes, 0 for no)" << endl;
    cin >> input;
    if (input == 1) {
        Graph unoGraph;

        // Adding UNO cards to the graph using a loop
        vector<string> colors = { "Red", "Green", "Blue" };
        for (const auto& color : colors) {
```

```
        vector<UNOCard> cards;
        for (int i = 0; i <= 9; i++) {
            cards.push_back(UNOCard(color, i));
        }
        unoGraph.addEdge(color, cards);
    }

    // Displaying the graph
    cout << "UNO Card Graph:" << endl;
    unoGraph.displayGraph();
    cout << endl;

    // Getting user input for the target card
    string targetColor;
    int targetNumber;

    cout << "Enter the color of the target card: ";
    cin >> targetColor;

    cout << "Enter the number of the target card: ";
    cin >> targetNumber;

    UNOCard targetCard(targetColor, targetNumber);

    // Searching for the user-specified target card using BFS and DFS
    unoGraph.BFS("Red", targetCard);
    unoGraph.DFS("Red", targetCard);
}
else {
        cout << "You chose not to view graph traversal." << endl;
    }

    UNOGame unoGame;
    unoGame.playGame();

    return 0;
}
```

7 CheckOff List

1. ~~**Trees**~~
2. ~~**Graphs**~~
3. ~~**Recursion**~~
4. ~~**Recursive sorts**~~
5. ~~**Hashing**~~