Department of Computer Engineering

BLG 336E
Analysis of Algorithm II
Homework 1
Report

ID	*Name*	*Surname*
150140126	*Emre*	*Reyhanlıoğlu*

PART 1 – PROJECT ARCHITECTURE

There are many architectural design of software development area. In this game project, there are many objects to simulate in the project such as characters, skills of characters and game status. Also there are many operations to perform on these objects to simulate the whole game. For these reasons, I used "Clean Architecture" and "SOLID(Single Responsibility, Open Closed, Liskov's Substitution, Interface Segregation and Dependency Inversion) principles whenever it's possible, because I believe that gaming is top of the software development area, so that it has to have a good architecture to be maintainable and testable.

With these architectural design, my prject has 4 parts: Model, usecase, simulation and user interface.

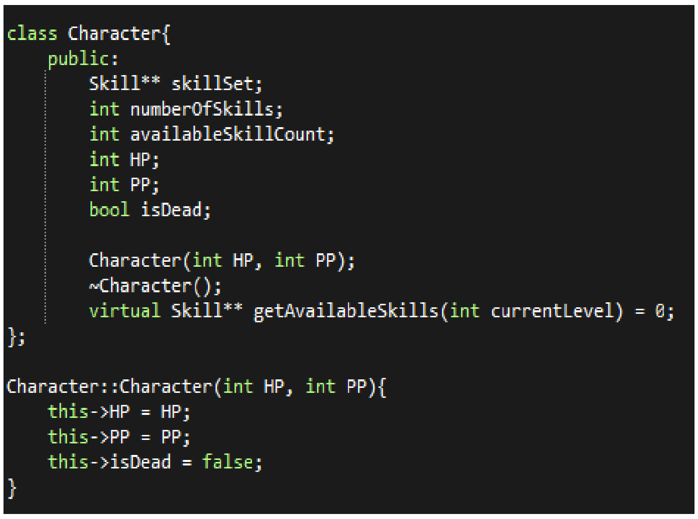
- Model has all of the data classes and necessary methods inside it.
- Usecase has all of the interactions between classes by using models.
- Simulation has all of the initial status of the game and it simulates the game by using usecases.
- User interface runs the proper simulation according to main arguments.

a) <u>Model Classes</u>

Model layer of the project has data classes which hold the necessary informations in the project. There are 6 classes and I will explain them shortly.

1) Character Classes

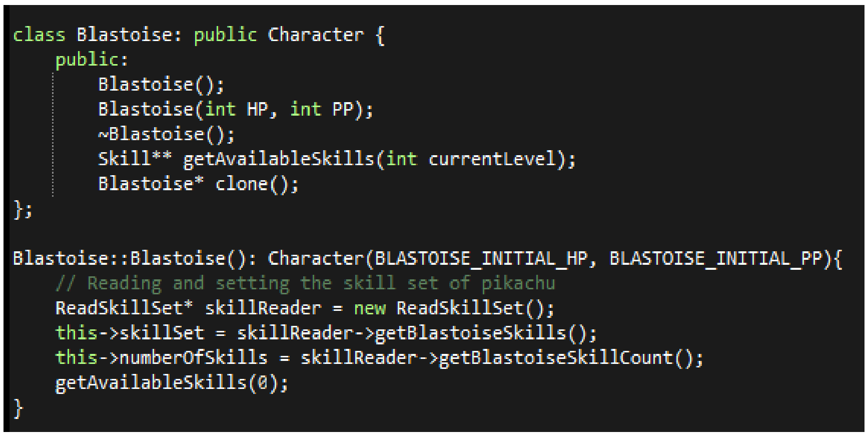
In my project, there is a "Character" class which holds the common attributes of a character and it has one abstract method called "*getAvailableSkills*", becase this method should be implemented by classes which extend "Character" classes. These child classes are "Blastoise" and "Pikachu" classes. You can see the part of "Character" super class below:



```
class Character{
    public:
        Skill** skillSet;
        int numberOfSkills;
        int availableSkillCount;
        int HP;
        int PP;
        bool isDead;

        Character(int HP, int PP);
        ~Character();
        virtual Skill** getAvailableSkills(int currentLevel) = 0;
};

Character::Character(int HP, int PP){
    this->HP = HP;
    this->PP = PP;
    this->isDead = false;
}
```

Figure 1: Part of Character class

Also, there are 2 implementation classes of “Character” class which are “Blastoise” and “Pikachu” classes. You can see the signatures of one of these classes below:



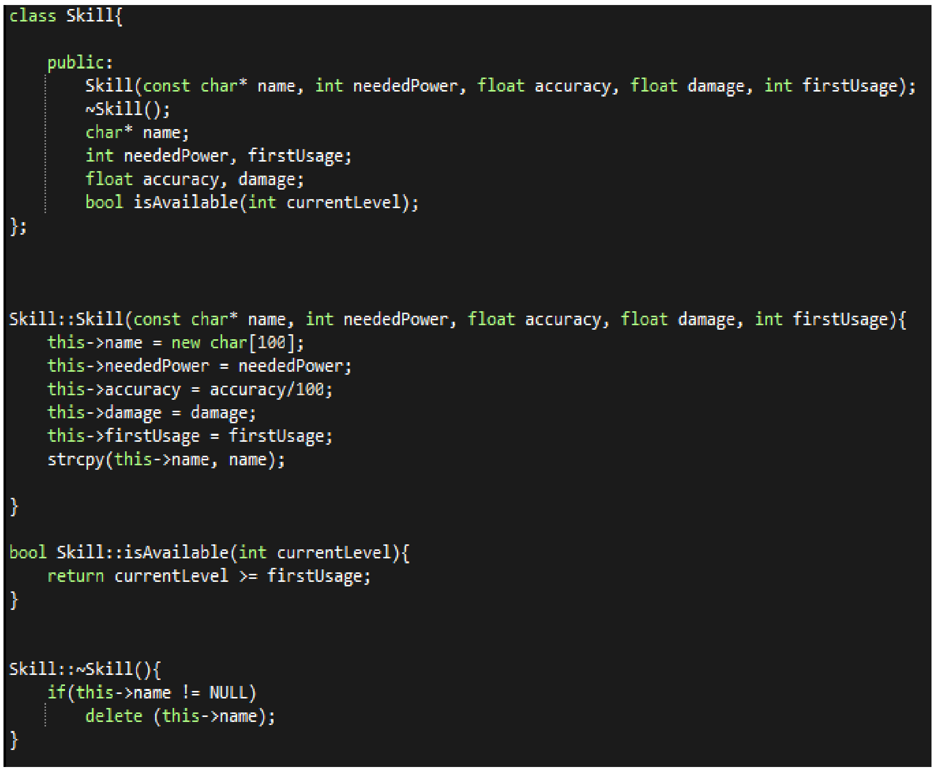
```cpp
class Blastoise: public Character {
    public:
        Blastoise();
        Blastoise(int HP, int PP);
        ~Blastoise();
        Skill** getAvailableSkills(int currentLevel);
        Blastoise* clone();
};

Blastoise::Blastoise(): Character(BLASTOISE_INITIAL_HP, BLASTOISE_INITIAL_PP){
    // Reading and setting the skill set of pikachu
    ReadSkillSet* skillReader = new ReadSkillSet();
    this->skillSet = skillReader->getBlastoiseSkills();
    this->numberOfSkills = skillReader->getBlastoiseSkillCount();
    getAvailableSkills(0);
}
```

Figure 2: Part of Blastoise class signature

Blastoise and Pikachu classes are very similar, but their *clone* and *getAvailableSkills* methods are different. Clone method returns a copy of that class to make the generation of the graph easier and available skills are different for different character types.

2) Skill Class

Skill class holds the informations about skills and it has one method inside it which is “*isAvailable*”. This method takes a parameter called “*currentLevel*” and checks whether this skill can be used at current level of the game by looking its first usage level. You can see the Skill class below:

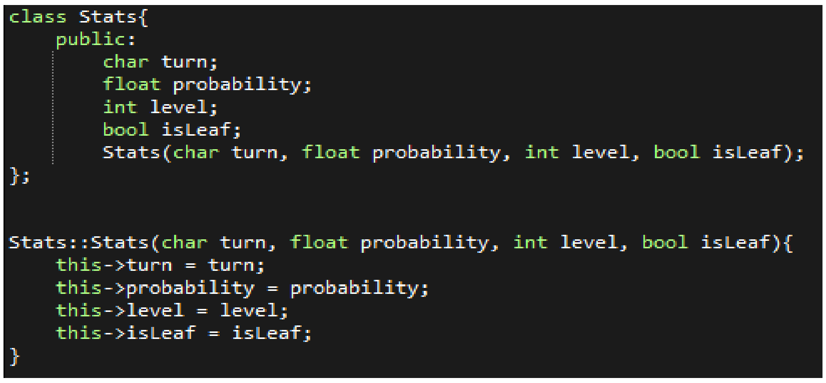


```cpp
class Skill{

    public:
        Skill(const char* name, int neededPower, float accuracy, float damage, int firstUsage);
        ~Skill();
        char* name;
        int neededPower, firstUsage;
        float accuracy, damage;
        bool isAvailable(int currentLevel);
};



Skill::Skill(const char* name, int neededPower, float accuracy, float damage, int firstUsage){
    this->name = new char[100];
    this->neededPower = neededPower;
    this->accuracy = accuracy/100;
    this->damage = damage;
    this->firstUsage = firstUsage;
    strcpy(this->name, name);

}

bool Skill::isAvailable(int currentLevel){
    return currentLevel >= firstUsage;
}


Skill::~Skill(){
    if(this->name != NULL)
        delete (this->name);
}
```

Figure 3: Skill class

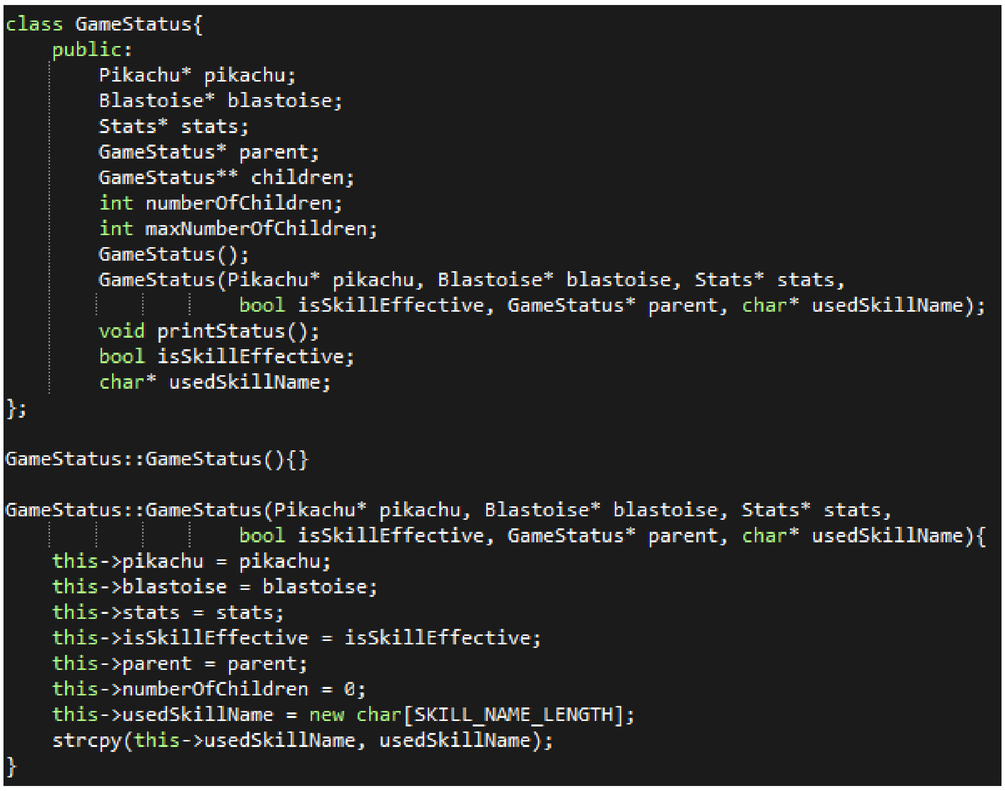
3) Game Status Classes

In the project, I need to hold current status of the game after every action(a character uses one of its skills). Hence, I wrote two classes named “*Stats*” and “GameStatus”. Also “*Stats”* class is part of “*GameStatus*” class. You can see both of these classes below:



```
class Stats{
    public:
        char turn;
        float probability;
        int level;
        bool isLeaf;
        Stats(char turn, float probability, int level, bool isLeaf);
};


Stats::Stats(char turn, float probability, int level, bool isLeaf){
    this->turn = turn;
    this->probability = probability;
    this->level = level;
    this->isLeaf = isLeaf;
}
```

Figure 4: Stats class



```
class GameStatus{
    public:
        Pikachu* pikachu;
        Blastoise* blastoise;
        Stats* stats;
        GameStatus* parent;
        GameStatus** children;
        int numberOfChildren;
        int maxNumberOfChildren;
        GameStatus();
        GameStatus(Pikachu* pikachu, Blastoise* blastoise, Stats* stats,
                    bool isSkillEffective, GameStatus* parent, char* usedSkillName);
        void printStatus();
        bool isSkillEffective;
        char* usedSkillName;
};

GameStatus::GameStatus(){}

GameStatus::GameStatus(Pikachu* pikachu, Blastoise* blastoise, Stats* stats,
                    bool isSkillEffective, GameStatus* parent, char* usedSkillName){
    this->pikachu = pikachu;
    this->blastoise = blastoise;
    this->stats = stats;
    this->isSkillEffective = isSkillEffective;
    this->parent = parent;
    this->numberOfChildren = 0;
    this->usedSkillName = new char[SKILL_NAME_LENGTH];
    strcpy(this->usedSkillName, usedSkillName);
}
```

Figure 5: GameStatus class

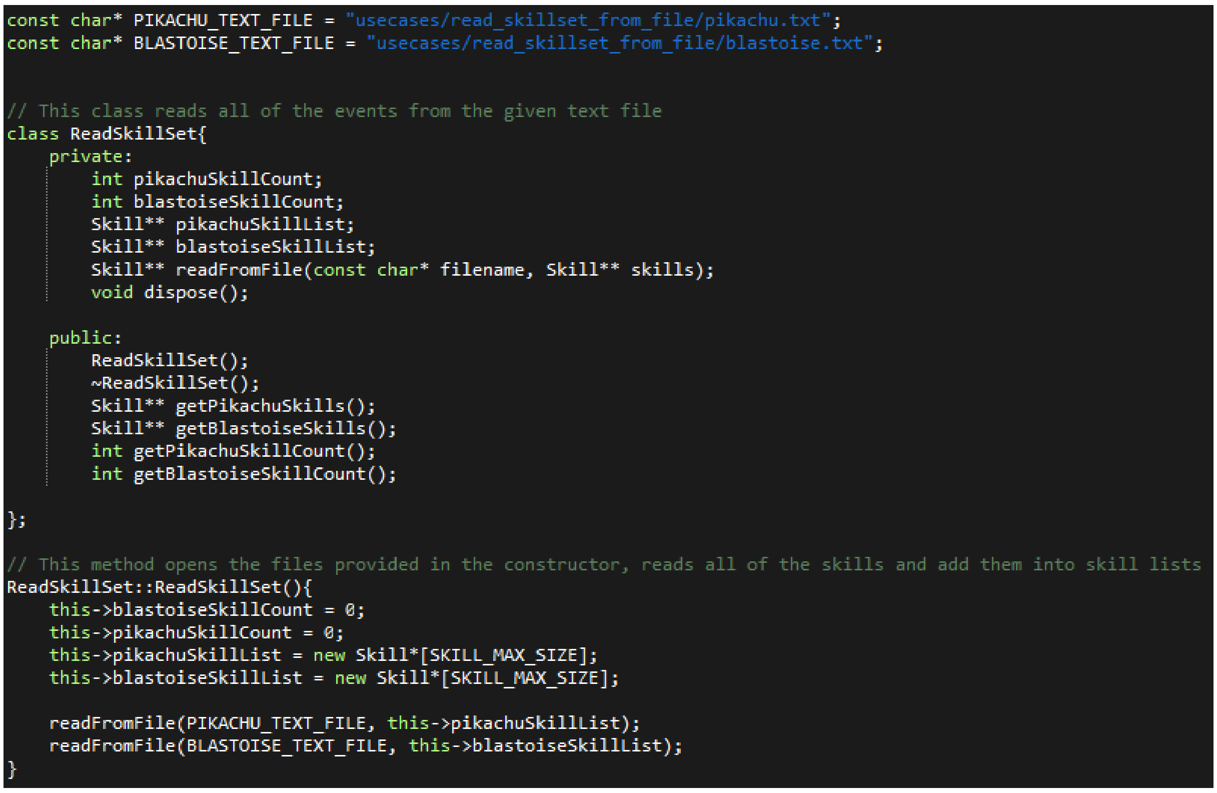
These was my project's model layer which holds all the data in the project. I didn't use getters/setters for these data classes. In a real game project, these may be important; however I made them public for the simplicity. In the next layers (usecases and simulation) these access modifiers are used properly.

b) Usecases

Usecase layer of the project has all of the interactions in the project between different classes. For example, *"Character"* uses a *"Skill"* and UseSkill class executes this operation properly. This layer is also may be called *"Bussiness Logic of the Application"* and its the brain layer of the project. Simulation layer simply uses these classes and this approach is good for *"SoC(Seperation of Concerns) Principle".*

1) ReadSkillSet Class

This class reads *"pikachu.txt"* and *"blastoise.txt"* files in the project and allows other classes to retreive these skill sets by providing getters. You can see the signature of the class below:

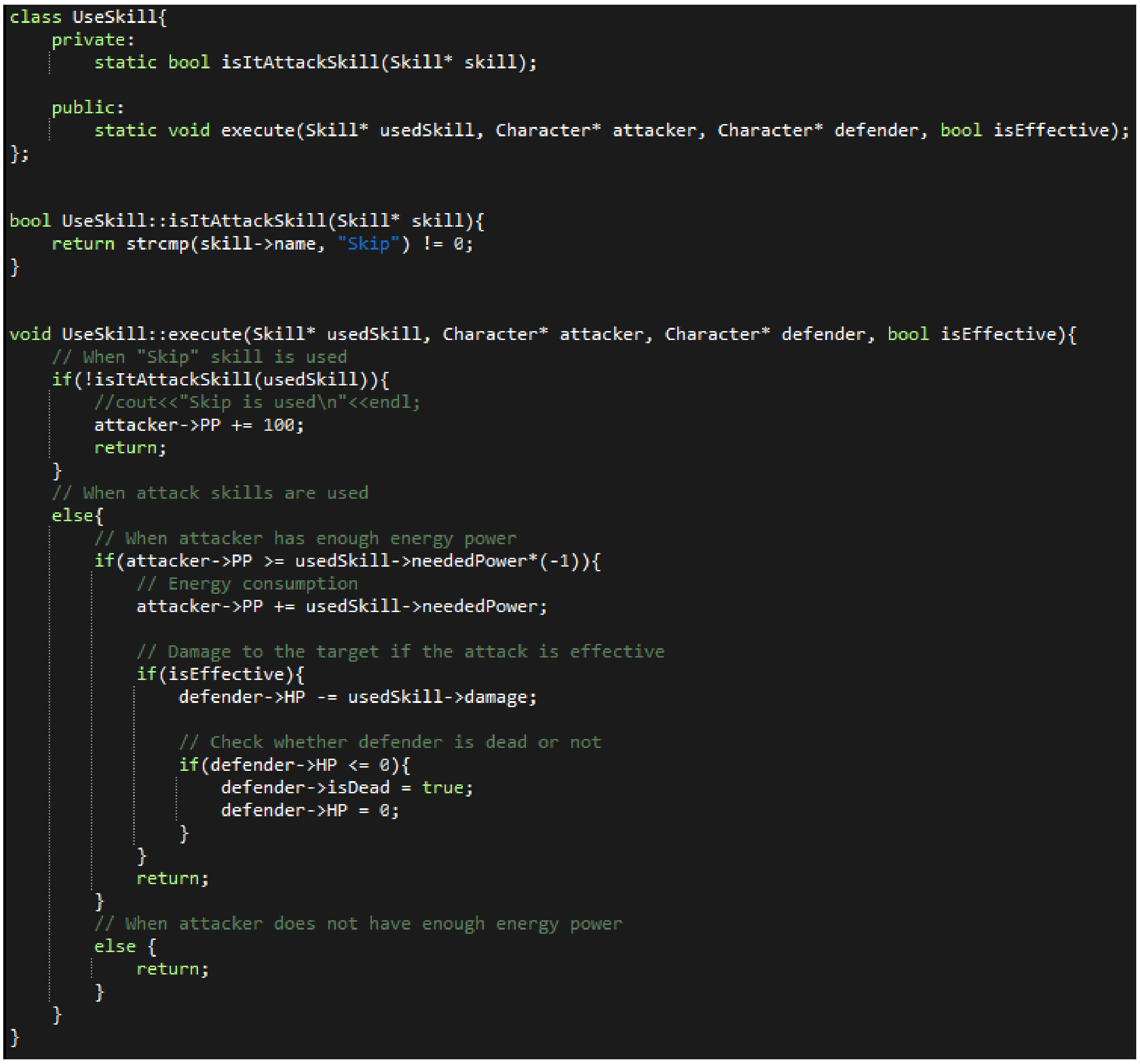


```
const char* PIKACHU_TEXT_FILE = "usecases/read_skillset_from_file/pikachu.txt";
const char* BLASTOISE_TEXT_FILE = "usecases/read_skillset_from_file/blastoise.txt";


// This class reads all of the events from the given text file
class ReadSkillSet{
    private:
        int pikachuSkillCount;
        int blastoiseSkillCount;
        Skill** pikachuSkillList;
        Skill** blastoiseSkillList;
        Skill** readFromFile(const char* filename, Skill** skills);
        void dispose();

    public:
        ReadSkillSet();
        ~ReadSkillSet();
        Skill** getPikachuSkills();
        Skill** getBlastoiseSkills();
        int getPikachuSkillCount();
        int getBlastoiseSkillCount();

};

// This method opens the files provided in the constructor, reads all of the skills and add them into skill lists
ReadSkillSet::ReadSkillSet(){
    this->blastoiseSkillCount = 0;
    this->pikachuSkillCount = 0;
    this->pikachuSkillList = new Skill*[SKILL_MAX_SIZE];
    this->blastoiseSkillList = new Skill*[SKILL_MAX_SIZE];

    readFromFile(PIKACHU_TEXT_FILE, this->pikachuSkillList);
    readFromFile(BLASTOISE_TEXT_FILE, this->blastoiseSkillList);
}
```

Figure 6: ReadSkillSet Class signature

2) UseSkill Class

This class is used to execute the effects of using a skill. Constructor parameters are usedSkill with “*Skill*” type, attacker and defender with “Character” type and isEffective with “*bool*” type. Using “Character” super class in the model layer allow me to design this class, because I don't care the attacker is from “Blastoise” class or “Pikachu” class. This “Character” interface allow me to do operations for both of these classes properly. You can see the class with all the details below:

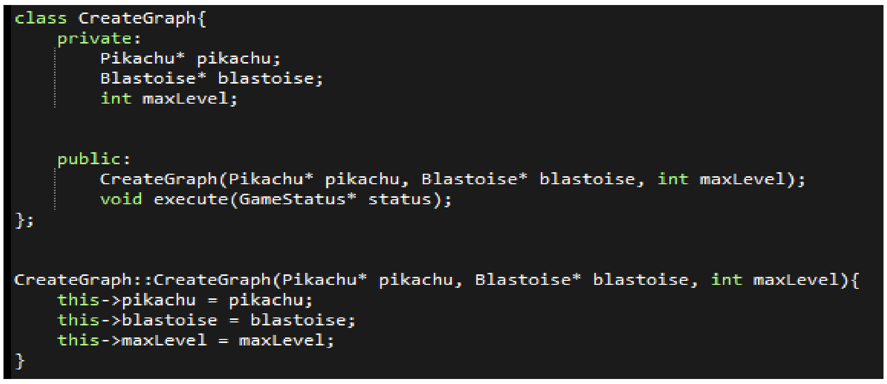


```
class UseSkill{
    private:
        static bool isItAttackSkill(Skill* skill);

    public:
        static void execute(Skill* usedSkill, Character* attacker, Character* defender, bool isEffective);
};


bool UseSkill::isItAttackSkill(Skill* skill){
    return strcmp(skill->name, "Skip") != 0;
}


void UseSkill::execute(Skill* usedSkill, Character* attacker, Character* defender, bool isEffective){
    // When "Skip" skill is used
    if(!isItAttackSkill(usedSkill)){
        //cout<<"Skip is used\n"<<endl;
        attacker->PP += 100;
        return;
    }
    // When attack skills are used
    else{
        // When attacker has enough energy power
        if(attacker->PP >= usedSkill->neededPower*(-1)){
            // Energy consumption
            attacker->PP += usedSkill->neededPower;

            // Damage to the target if the attack is effective
            if(isEffective){
                defender->HP -= usedSkill->damage;

                // Check whether defender is dead or not
                if(defender->HP <= 0){
                    defender->isDead = true;
                    defender->HP = 0;
                }
            }
            return;
        }
        // When attacker does not have enough energy power
        else {
            return;
        }
    }
}
```

Figure 7: UseSkill Class

3) CreateGraph Class

This class creates the action graph of the game according to initial players' status and max level parameter. You can see the signature of the class below:



```
class CreateGraph{
    private:
        Pikachu* pikachu;
        Blastoise* blastoise;
        int maxLevel;


    public:
        CreateGraph(Pikachu* pikachu, Blastoise* blastoise, int maxLevel);
        void execute(GameStatus* status);
};


CreateGraph::CreateGraph(Pikachu* pikachu, Blastoise* blastoise, int maxLevel){
    this->pikachu = pikachu;
    this->blastoise = blastoise;
    this->maxLevel = maxLevel;
}
```

Figure 8: CreateGraph Class signature

Execute method of this class creates the graph properly. This method's codes are long enough, so that I will explain what the code does step by step. Also, you can read the comments in the code if you want, it's clear and easy to read.

Steps of execute method;

1) Check the current level of the input status of the graph, if it is lower than the max level then continue.

2) Check whether both of the players are alive or not, if they alive are then continue.

3) Check the turn variable of status to act according to player(Pikachu or Blastoise). (After this point there are two branches which are very similar to each other)

4) Get the current player's skill set and number of available skill at that level

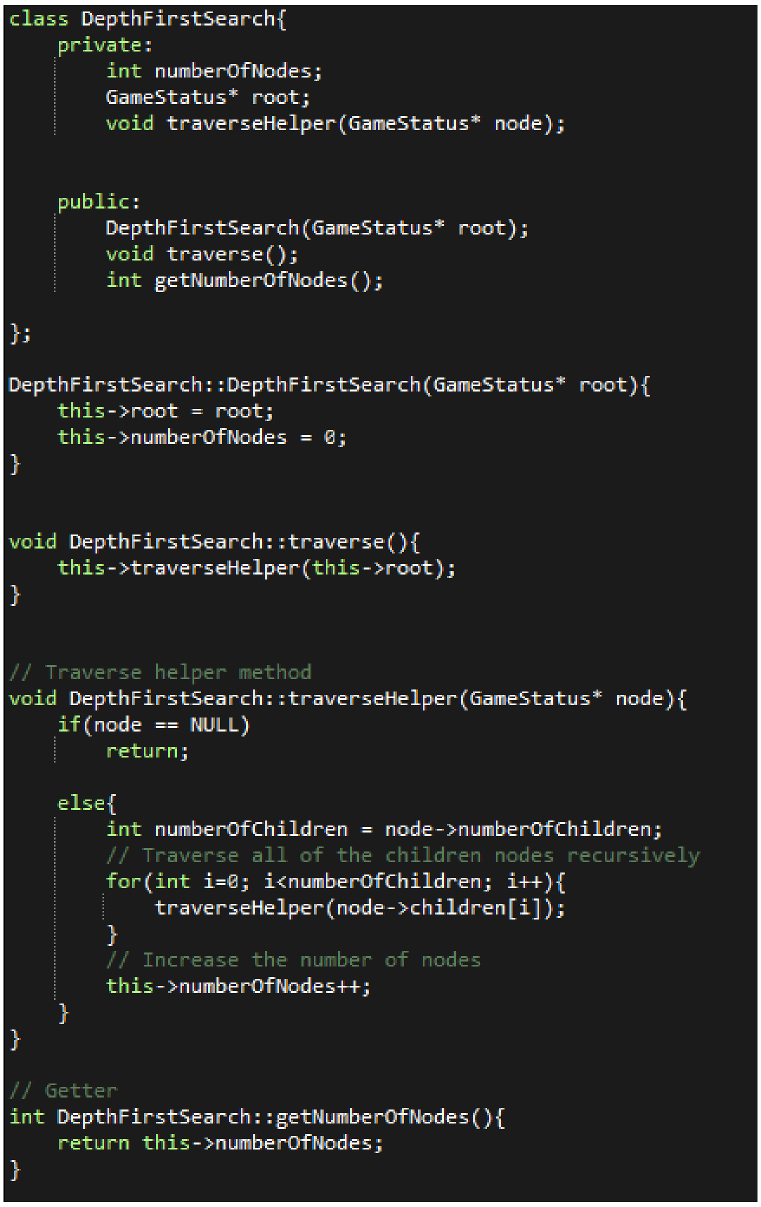
5) Start a for loop to add new child nodes to current node based on number of skills and skills' success probabilities.

6) If success probability of a skill is greater than zero, then create a new node by cloning the current state and executing the skill. After that, call this execute method with recently created child node recursively if both of the players are still alive.

7) Do the same operations in 6 for a missing skills if its missing probability is also greater than zero.

4) DepthFirstSearch Class

DepthFirstSearch class traverses the graph and counts the total number of nodes in the graph. It takes a root node in the constructor and provides a getter for total number of nodes to the other classes. Its traverse method has a very simple. It calls a helper method named *"traverseHelper"* to traverse the tree in recursive way. This helper method calls itself and increments the total number of nodes for every child node of that class. You can see this class below:



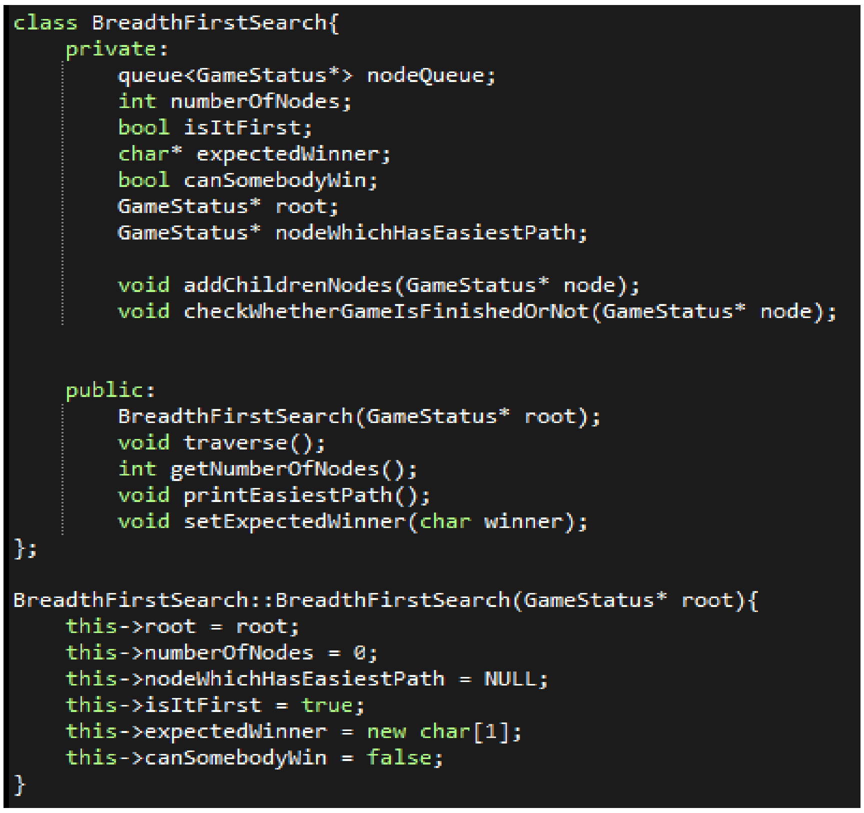
```
class DepthFirstSearch{
    private:
        int numberOfNodes;
        GameStatus* root;
        void traverseHelper(GameStatus* node);


    public:
        DepthFirstSearch(GameStatus* root);
        void traverse();
        int getNumberOfNodes();

};

DepthFirstSearch::DepthFirstSearch(GameStatus* root){
    this->root = root;
    this->numberOfNodes = 0;
}


void DepthFirstSearch::traverse(){
    this->traverseHelper(this->root);
}


// Traverse helper method
void DepthFirstSearch::traverseHelper(GameStatus* node){
    if(node == NULL)
        return;

    else{
        int numberOfChildren = node->numberOfChildren;
        // Traverse all of the children nodes recursively
        for(int i=0; i<numberOfChildren; i++){
            traverseHelper(node->children[i]);
        }
        // Increase the number of nodes
        this->numberOfNodes++;
    }
}

// Getter
int DepthFirstSearch::getNumberOfNodes(){
    return this->numberOfNodes;
}
```

Figure 9: DepthFirstSearch Class

5) BreadthFirstSearch Class

This class also traverses the tree like the previous DepthFirstSearch class; however this class has additional parameters and methods because of the last part of the homework which is finding easiest path. I have used BFS algorithm to find the easiest path, because easiest path should be close to the root node, so that searching level by level makes sense.

Moreover, I used a queue in my BFS algorithm to traverse the tree in level order. In this algorithm, root is added to the queue firstly. Then I start a for loop which runs until queue is empty. Inside the loop, I get the first node in the queue and add its children if there is any then pop that node in the queue. In this way, all of the nodes will be visited in a right order. Also, I inspired from a video about this BFS algorithm. If want to take a look at, the link of the video is https://www.youtube.com/watch?v=0u78hx-66Xk



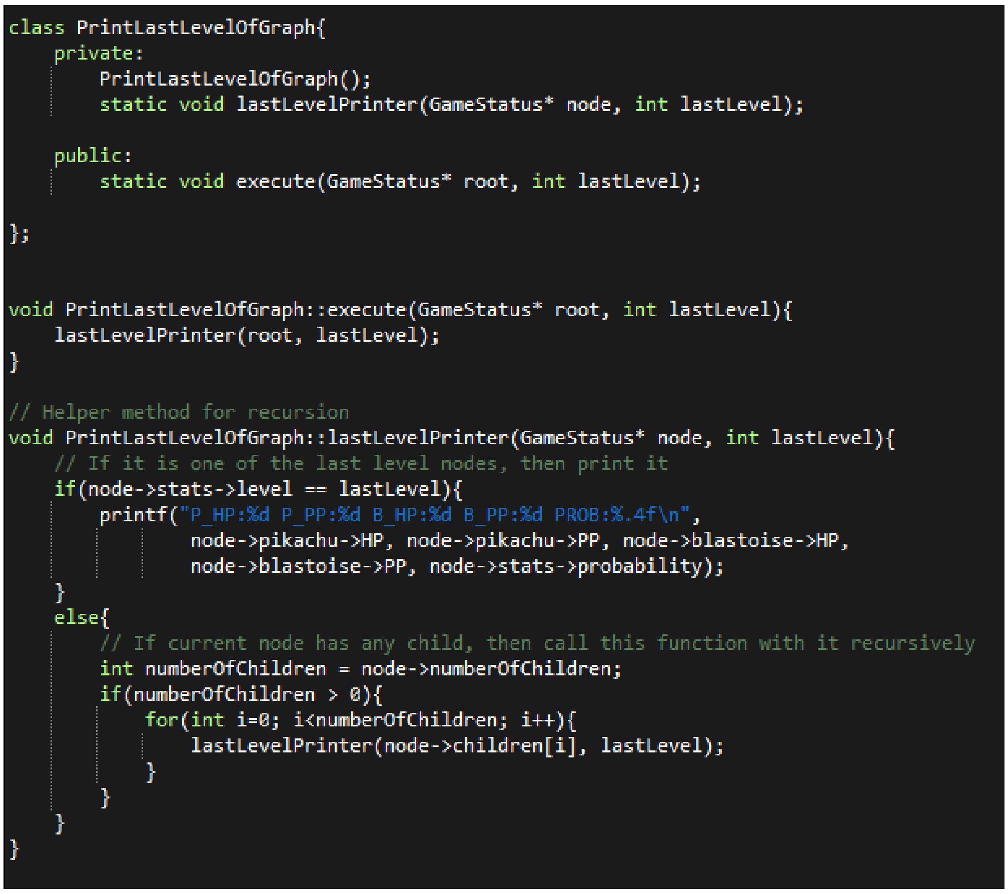
```cpp
class BreadthFirstSearch{
    private:
        queue<GameStatus*> nodeQueue;
        int numberOfNodes;
        bool isItFirst;
        char* expectedWinner;
        bool canSomebodyWin;
        GameStatus* root;
        GameStatus* nodeWhichHasEasiestPath;

        void addChildrenNodes(GameStatus* node);
        void checkWhetherGameIsFinishedOrNot(GameStatus* node);


    public:
        BreadthFirstSearch(GameStatus* root);
        void traverse();
        int getNumberOfNodes();
        void printEasiestPath();
        void setExpectedWinner(char winner);
};

BreadthFirstSearch::BreadthFirstSearch(GameStatus* root){
    this->root = root;
    this->numberOfNodes = 0;
    this->nodeWhichHasEasiestPath = NULL;
    this->isItFirst = true;
    this->expectedWinner = new char[1];
    this->canSomebodyWin = false;
}
```

Figure 10: BreadthFirstSearch Class signature

For the implementation of these methods, please take a look at the code. You will find detailed comments about what these methods do.

6) PrintLastLevelOfGraph Class

This class prints the last level of a given graph. It takes root node of the graph and maximum level of it in the constructor. Also, it has a helper method for recursion. It checks the node's level and if it is equal to the max level then prints it, else it calls itself with its children nodes. You can see the class below:



```cpp
class PrintLastLevelOfGraph{
    private:
        PrintLastLevelOfGraph();
        static void lastLevelPrinter(GameStatus* node, int lastLevel);

    public:
        static void execute(GameStatus* root, int lastLevel);

};


void PrintLastLevelOfGraph::execute(GameStatus* root, int lastLevel){
    lastLevelPrinter(root, lastLevel);
}

// Helper method for recursion
void PrintLastLevelOfGraph::lastLevelPrinter(GameStatus* node, int lastLevel){
    // If it is one of the last level nodes, then print it
    if(node->stats->level == lastLevel){
        printf("P_HP:%d P_PP:%d B_HP:%d B_PP:%d PROB:%.4f\n",
                node->pikachu->HP, node->pikachu->PP, node->blastoise->HP,
                node->blastoise->PP, node->stats->probability);
    }
    else{
        // If current node has any child, then call this function with it recursively
        int numberOfChildren = node->numberOfChildren;
        if(numberOfChildren > 0){
            for(int i=0; i<numberOfChildren; i++){
                lastLevelPrinter(node->children[i], lastLevel);
            }
        }
    }
}
```

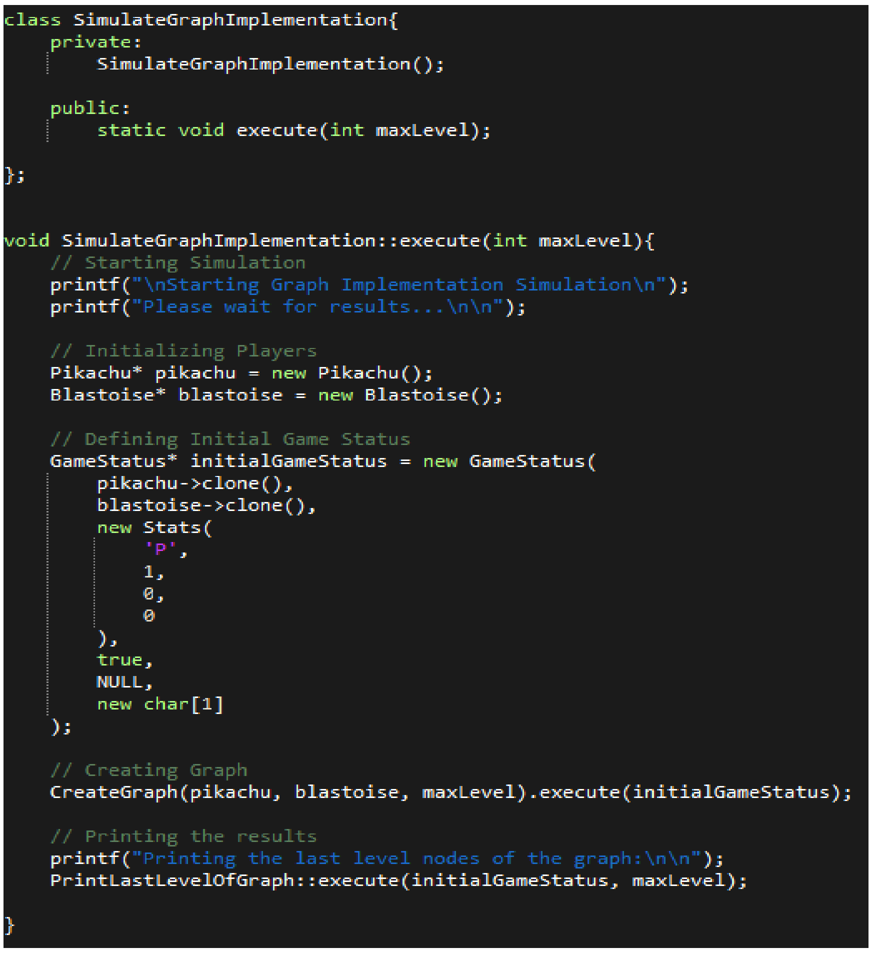
Figure 11: PrintLastLevelOfGraph Class

c) Simulation

Simulation layer has all of the necessary classes which setup the initial status of the scenario and simulate the game with that scenario. These simulation classes use the usecase classes.

1) SimulateGraphImplementation Class

This class simulates the first part of the homework. Execute method is a static method and takes the max level of the graph as a parameter, and it prints the last layer nodes of the graph.

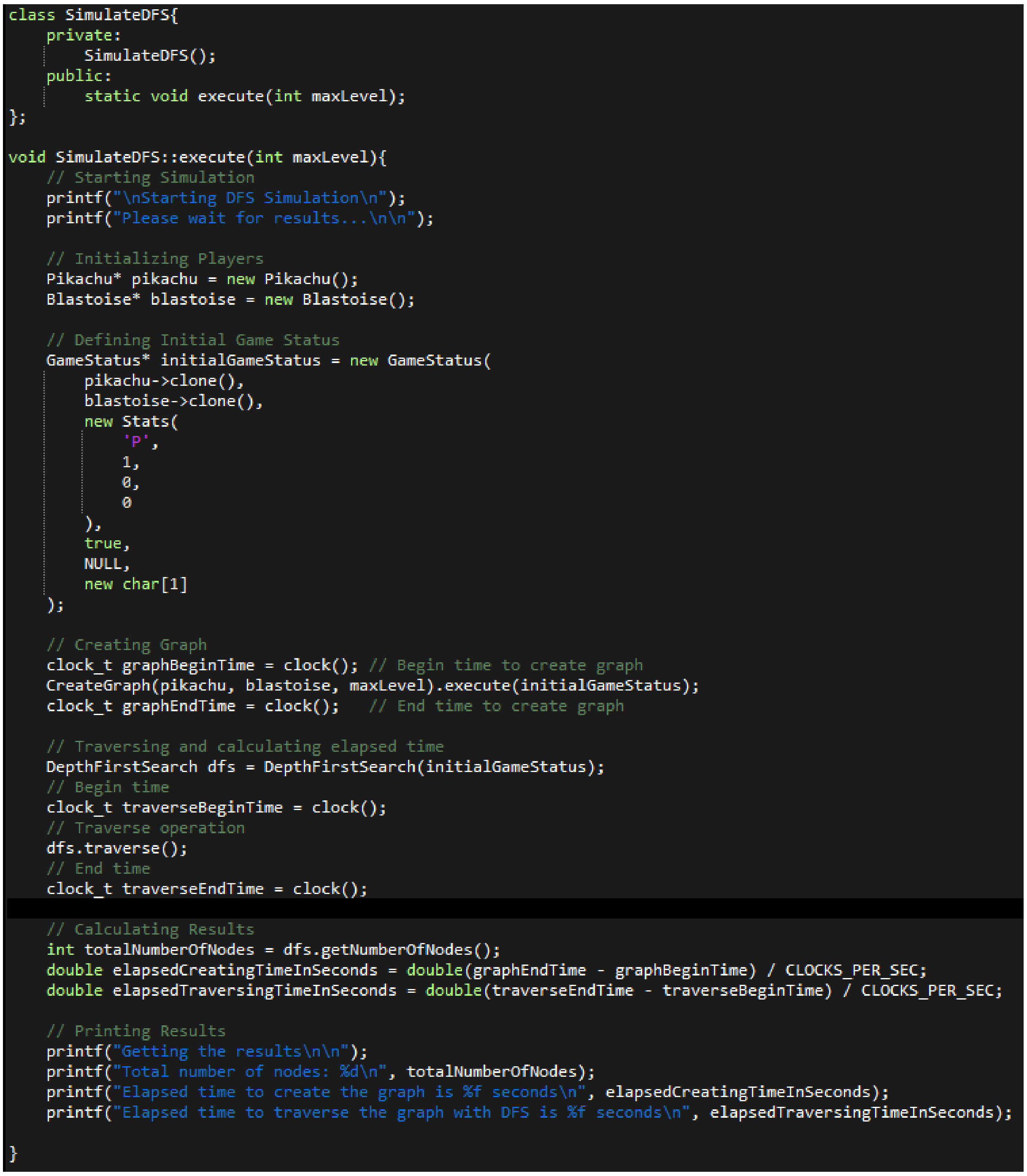


```
class SimulateGraphImplementation{
    private:
        SimulateGraphImplementation();

    public:
        static void execute(int maxLevel);

};


void SimulateGraphImplementation::execute(int maxLevel){
    // Starting Simulation
    printf("\nStarting Graph Implementation Simulation\n");
    printf("Please wait for results...\n\n");

    // Initializing Players
    Pikachu* pikachu = new Pikachu();
    Blastoise* blastoise = new Blastoise();

    // Defining Initial Game Status
    GameStatus* initialGameStatus = new GameStatus(
        pikachu->clone(),
        blastoise->clone(),
        new Stats(
            'P',
            1,
            0,
            0
        ),
        true,
        NULL,
        new char[1]
    );

    // Creating Graph
    CreateGraph(pikachu, blastoise, maxLevel).execute(initialGameStatus);

    // Printing the results
    printf("Printing the last level nodes of the graph:\n\n");
    PrintLastLevelOfGraph::execute(initialGameStatus, maxLevel);

}
```

Figure 12: SimulateGraphImplementation Class

2.a) SimulateDFS Class

This class simulates the second part of the homework for the DFS case. In the execute method of the class, pikachu and blastoise characters are spawned and initial game status is initialized properly. After initialization, graph is created and traverse method is called, then elaped time informations is printed to the console. You can see this class below:

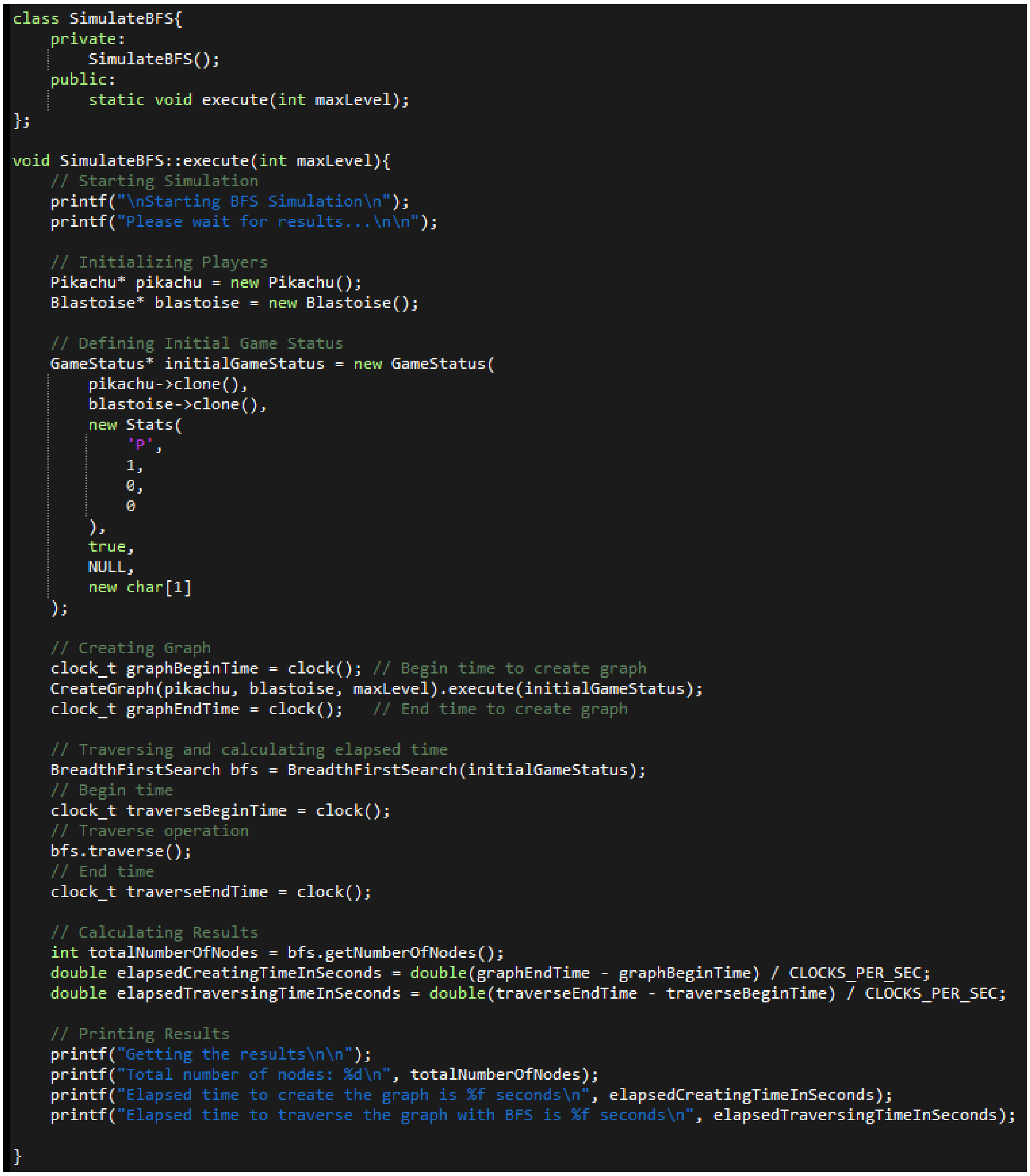


```
class SimulateDFS{
    private:
        SimulateDFS();
    public:
        static void execute(int maxLevel);
};

void SimulateDFS::execute(int maxLevel){
    // Starting Simulation
    printf("\nStarting DFS Simulation\n");
    printf("Please wait for results...\n\n");

    // Initializing Players
    Pikachu* pikachu = new Pikachu();
    Blastoise* blastoise = new Blastoise();

    // Defining Initial Game Status
    GameStatus* initialGameStatus = new GameStatus(
        pikachu->clone(),
        blastoise->clone(),
        new Stats(
            'P',
            1,
            0,
            0
        ),
        true,
        NULL,
        new char[1]
    );

    // Creating Graph
    clock_t graphBeginTime = clock(); // Begin time to create graph
    CreateGraph(pikachu, blastoise, maxLevel).execute(initialGameStatus);
    clock_t graphEndTime = clock();   // End time to create graph

    // Traversing and calculating elapsed time
    DepthFirstSearch dfs = DepthFirstSearch(initialGameStatus);
    // Begin time
    clock_t traverseBeginTime = clock();
    // Traverse operation
    dfs.traverse();
    // End time
    clock_t traverseEndTime = clock();

    // Calculating Results
    int totalNumberOfNodes = dfs.getNumberOfNodes();
    double elapsedCreatingTimeInSeconds = double(graphEndTime - graphBeginTime) / CLOCKS_PER_SEC;
    double elapsedTraversingTimeInSeconds = double(traverseEndTime - traverseBeginTime) / CLOCKS_PER_SEC;

    // Printing Results
    printf("Getting the results\n\n");
    printf("Total number of nodes: %d\n", totalNumberOfNodes);
    printf("Elapsed time to create the graph is %f seconds\n", elapsedCreatingTimeInSeconds);
    printf("Elapsed time to traverse the graph with DFS is %f seconds\n", elapsedTraversingTimeInSeconds);

}
```

Figure 13: SimulateDFS Class

2.b) SimulateBFS Class

This class is very similar to the previous simulation class. It simulates the second part of the homework for the BFS case. In the execute method of the class, pikachu and blastoise characters are spawned and initial game status is initialized properly. After initialization, graph is created and traverse method is called, then elaped time informations is printed to the console. You can see this class below:

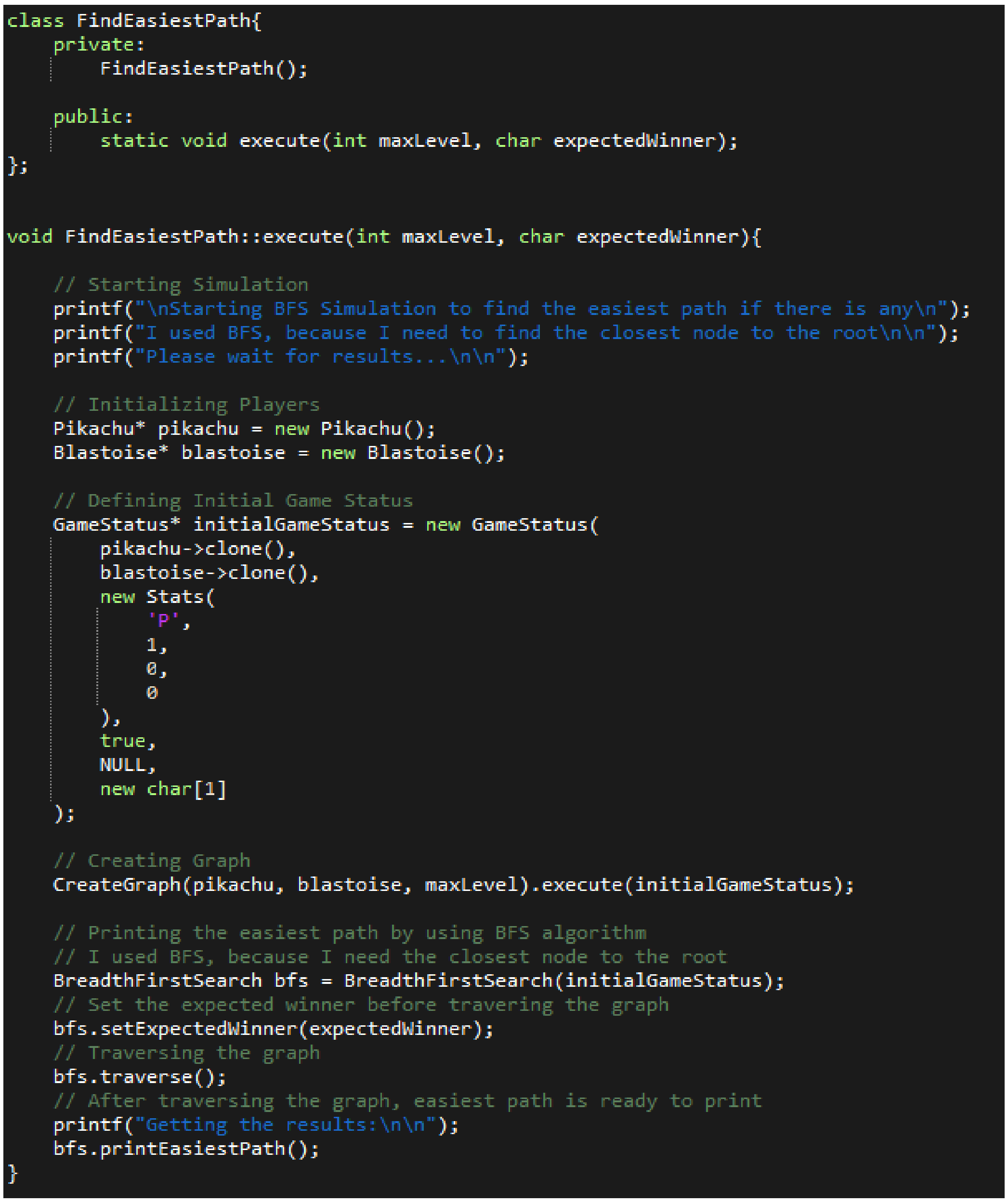


```cpp
class SimulateBFS{
    private:
        SimulateBFS();
    public:
        static void execute(int maxLevel);
};

void SimulateBFS::execute(int maxLevel){
    // Starting Simulation
    printf("\nStarting BFS Simulation\n");
    printf("Please wait for results...\n\n");

    // Initializing Players
    Pikachu* pikachu = new Pikachu();
    Blastoise* blastoise = new Blastoise();

    // Defining Initial Game Status
    GameStatus* initialGameStatus = new GameStatus(
        pikachu->clone(),
        blastoise->clone(),
        new Stats(
            'P',
            1,
            0,
            0
        ),
        true,
        NULL,
        new char[1]
    );

    // Creating Graph
    clock_t graphBeginTime = clock(); // Begin time to create graph
    CreateGraph(pikachu, blastoise, maxLevel).execute(initialGameStatus);
    clock_t graphEndTime = clock();   // End time to create graph

    // Traversing and calculating elapsed time
    BreadthFirstSearch bfs = BreadthFirstSearch(initialGameStatus);
    // Begin time
    clock_t traverseBeginTime = clock();
    // Traverse operation
    bfs.traverse();
    // End time
    clock_t traverseEndTime = clock();

    // Calculating Results
    int totalNumberOfNodes = bfs.getNumberOfNodes();
    double elapsedCreatingTimeInSeconds = double(graphEndTime - graphBeginTime) / CLOCKS_PER_SEC;
    double elapsedTraversingTimeInSeconds = double(traverseEndTime - traverseBeginTime) / CLOCKS_PER_SEC;

    // Printing Results
    printf("Getting the results\n\n");
    printf("Total number of nodes: %d\n", totalNumberOfNodes);
    printf("Elapsed time to create the graph is %f seconds\n", elapsedCreatingTimeInSeconds);
    printf("Elapsed time to traverse the graph with BFS is %f seconds\n", elapsedTraversingTimeInSeconds);

}
```

Figure 14: SimulateBFS Class

3) FindEasiestPath Class

This class simulates the last part of the homework. It finds the easiest path which is the shortest action sequence of the termination game for a selected winner. In the execute method of the class, pikachu and blastoise characters are spawned and initial game status is initialized properly. After the initialization, graph is created and BFS is applied to the graph to detect the node which has easiest path according to the expected winner. Finally, printEasiestPath method prints the full path from root to node and other necessary informations. You can see this class below:

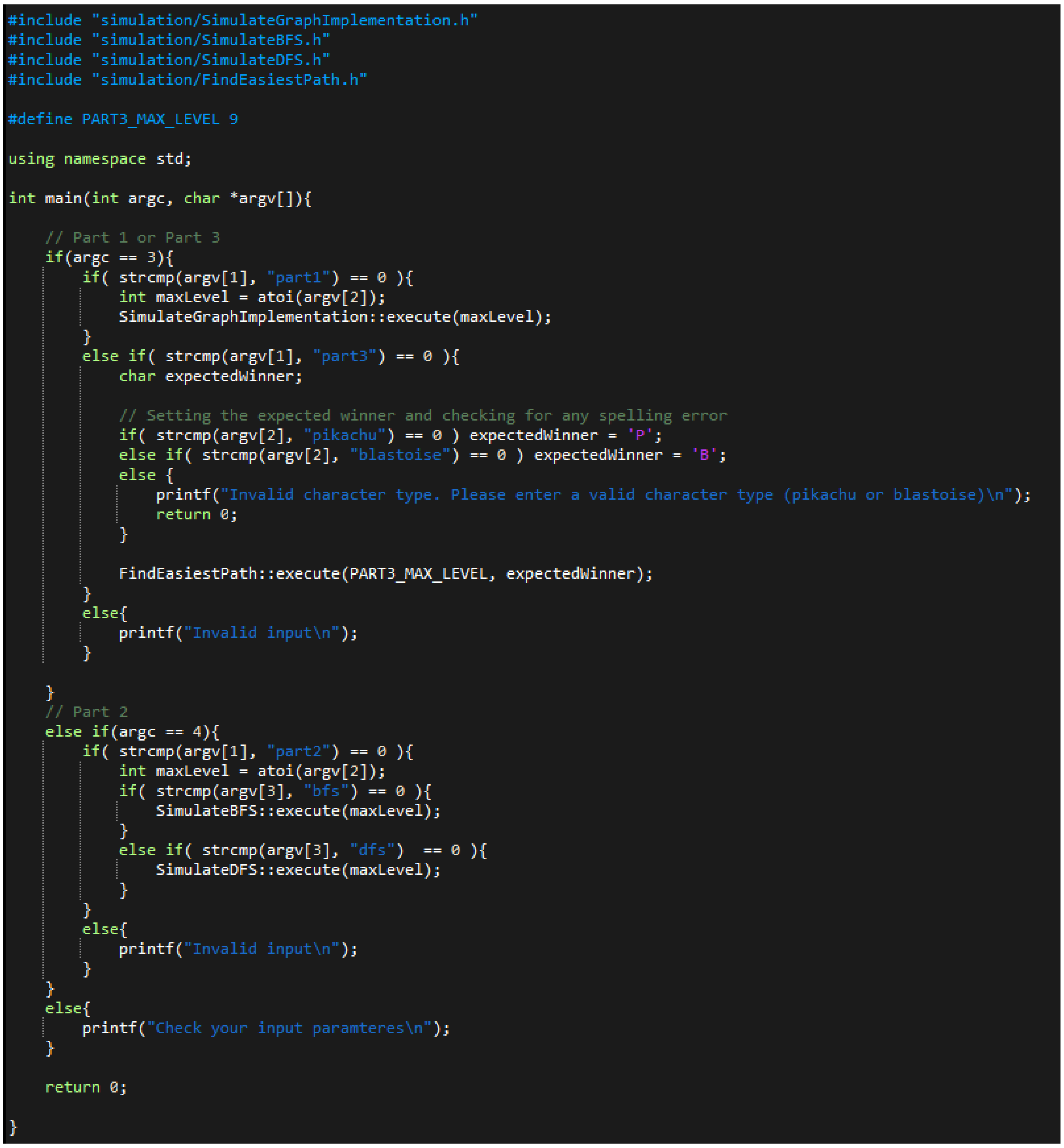


```cpp
class FindEasiestPath{
    private:
        FindEasiestPath();

    public:
        static void execute(int maxLevel, char expectedWinner);
};


void FindEasiestPath::execute(int maxLevel, char expectedWinner){

    // Starting Simulation
    printf("\nStarting BFS Simulation to find the easiest path if there is any\n");
    printf("I used BFS, because I need to find the closest node to the root\n\n");
    printf("Please wait for results...\n\n");

    // Initializing Players
    Pikachu* pikachu = new Pikachu();
    Blastoise* blastoise = new Blastoise();

    // Defining Initial Game Status
    GameStatus* initialGameStatus = new GameStatus(
        pikachu->clone(),
        blastoise->clone(),
        new Stats(
            'P',
            1,
            0,
            0
        ),
        true,
        NULL,
        new char[1]
    );

    // Creating Graph
    CreateGraph(pikachu, blastoise, maxLevel).execute(initialGameStatus);

    // Printing the easiest path by using BFS algorithm
    // I used BFS, because I need the closest node to the root
    BreadthFirstSearch bfs = BreadthFirstSearch(initialGameStatus);
    // Set the expected winner before travering the graph
    bfs.setExpectedWinner(expectedWinner);
    // Traversing the graph
    bfs.traverse();
    // After traversing the graph, easiest path is ready to print
    printf("Getting the results:\n\n");
    bfs.printEasiestPath();
}
```

Figure 15: FindEasiestPath Class

d) User Interface

This layer contains “main.cpp” file which runs the selected simulation according to the main arguments which are provided by the user and it checks for the errors and prints the error messages. You can see the code below:

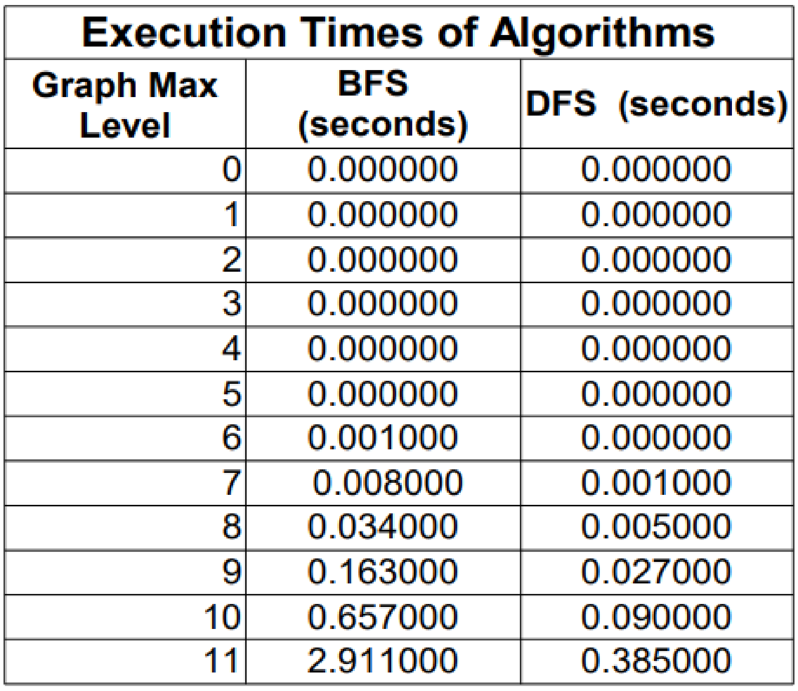


```
#include "simulation/SimulateGraphImplementation.h"
#include "simulation/SimulateBFS.h"
#include "simulation/SimulateDFS.h"
#include "simulation/FindEasiestPath.h"

#define PART3_MAX_LEVEL 9

using namespace std;

int main(int argc, char *argv[]){

    // Part 1 or Part 3
    if(argc == 3){
        if( strcmp(argv[1], "part1") == 0 ){
            int maxLevel = atoi(argv[2]);
            SimulateGraphImplementation::execute(maxLevel);
        }
        else if( strcmp(argv[1], "part3") == 0 ){
            char expectedWinner;

            // Setting the expected winner and checking for any spelling error
            if( strcmp(argv[2], "pikachu") == 0 ) expectedWinner = 'P';
            else if( strcmp(argv[2], "blastoise") == 0 ) expectedWinner = 'B';
            else {
                printf("Invalid character type. Please enter a valid character type (pikachu or blastoise)\n");
                return 0;
            }

            FindEasiestPath::execute(PART3_MAX_LEVEL, expectedWinner);
        }
        else{
            printf("Invalid input\n");
        }

    }
    // Part 2
    else if(argc == 4){
        if( strcmp(argv[1], "part2") == 0 ){
            int maxLevel = atoi(argv[2]);
            if( strcmp(argv[3], "bfs") == 0 ){
                SimulateBFS::execute(maxLevel);
            }
            else if( strcmp(argv[3], "dfs")  == 0 ){
                SimulateDFS::execute(maxLevel);
            }
        }
        else{
            printf("Invalid input\n");
        }
    }
    else{
        printf("Check your input paramteres\n");
    }

    return 0;

}
```

Figure 16: “main.cpp”

PART 2 – SIMULATION RESULTS

These are the simulation results for both BFS and DFS algorithms in the first 12 level graphs.



Execution Times of Algorithms		
Graph Max Level	BFS (seconds)	DFS (seconds)
0	0.000000	0.000000
1	0.000000	0.000000
2	0.000000	0.000000
3	0.000000	0.000000
4	0.000000	0.000000
5	0.000000	0.000000
6	0.001000	0.000000
7	0.008000	0.001000
8	0.034000	0.005000
9	0.163000	0.027000
10	0.657000	0.090000
11	2.911000	0.385000

Figure 17: Simulation results

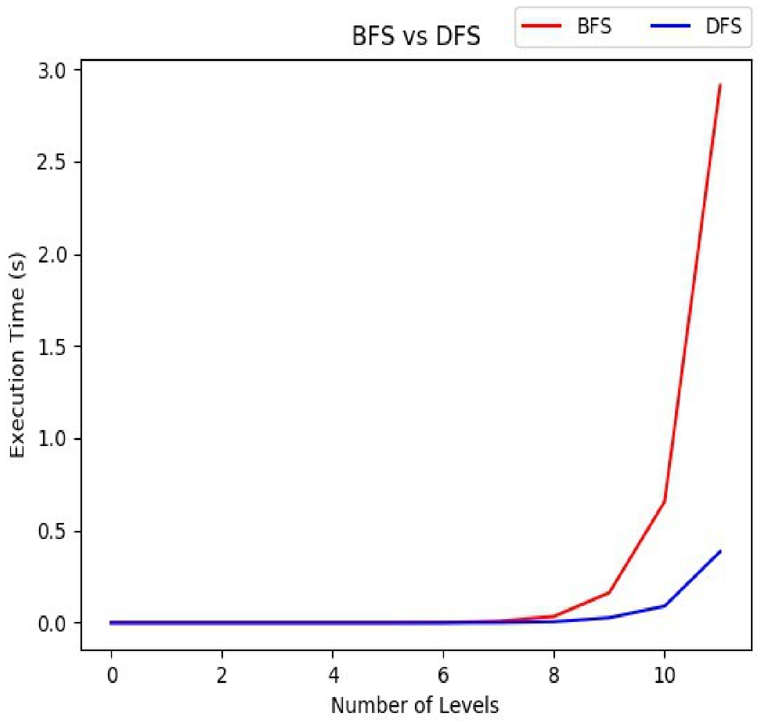


Figure 18: Graphical results of these simulations