ISTANBUL TECHNICAL UNIVERSITY
FACULTY OF COMPUTER AND INFORMATICS ENGINEERING

BLG336E – Analysis of Algorithms II
Spring 2022

Lecturer: Assoc. Prof. Dr. İlkay Öksüz
T.As: Res. Asst. Uğur Ayvaz,
Alperen Kantarcı,
Caner Özer,
Muhammed Raşit Erol

Project 1 – Battleship Game

Mustafa İzzet Muştu
504211564

1. Introduction

In this project, we are wanted to implement a battleship game with 2 players by using Breadth First Search (BFS) and Depth First Search (DFS) algorithms. Players start in a cell on game board and explore other cells from starting neighbor cells. Players have ships on some cells on game board. Exploring algorithms are done with BFS or DFS algorithm. Player 1 starts the game and player 2 continues. They make moves one by one with this order. If a player encounters a ship of the other player in a cell, this part of the ship will sink. Once one of the players sinks all of the ships of the other player, he/she wins.

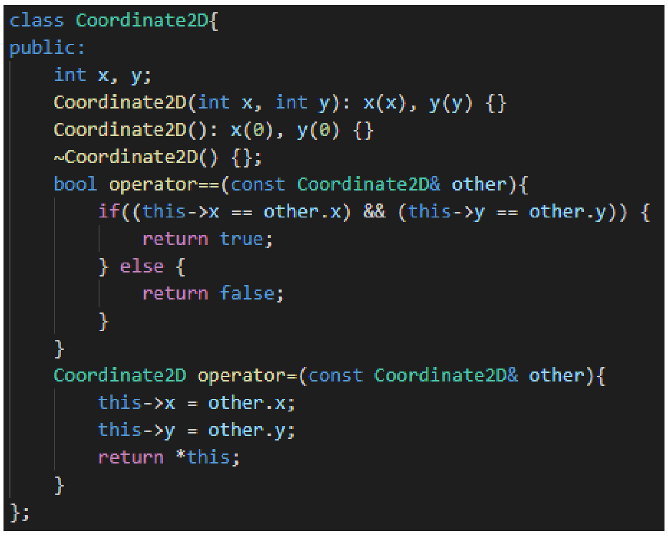
2. Development and Environment

The game is coded on Microsoft Visual Studio Code in a Microsoft Windows 10 system. It is compiled by using g++ both on local computer and ITU SSH server without any warning or error.

3. Classes:

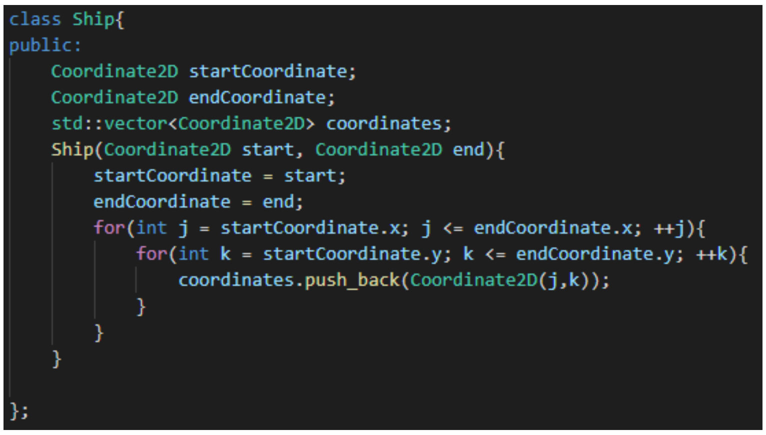
There are 5 classes in the code. These are Coordiante2D, Ship, Player, Node and GameBoard.

3.1 Coordinate2D: This class just represents the x and y coordinates. In addition, equal and is equal operators are overloaded.



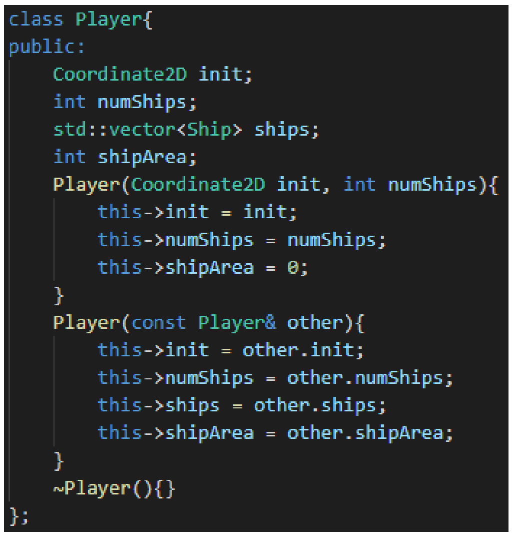
```cpp
class Coordinate2D{
public:
    int x, y;
    Coordinate2D(int x, int y): x(x), y(y) {}
    Coordinate2D(): x(0), y(0) {}
    ~Coordinate2D() {};
    bool operator==(const Coordinate2D& other){
        if((this->x == other.x) && (this->y == other.y)) {
            return true;
        } else {
            return false;
        }
    }
    Coordinate2D operator=(const Coordinate2D& other){
        this->x = other.x;
        this->y = other.y;
        return *this;
    }
};
```

3.2 Ship: This class represents ship in the game and holds the ship coordinates on a 2D coordinate system.



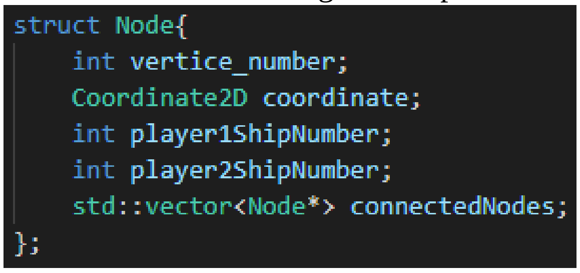
```cpp
class Ship{
public:
    Coordinate2D startCoordinate;
    Coordinate2D endCoordinate;
    std::vector<Coordinate2D> coordinates;
    Ship(Coordinate2D start, Coordinate2D end){
        startCoordinate = start;
        endCoordinate = end;
        for(int j = startCoordinate.x; j <= endCoordinate.x; ++j){
            for(int k = startCoordinate.y; k <= endCoordinate.y; ++k){
                coordinates.push_back(Coordinate2D(j,k));
            }
        }
    }

};
```

3.3 Player: This class represents player in the game and holds his/her ships with total number of ship coordinates on a 2D coordinate system. Also, players start in a position on the game board.



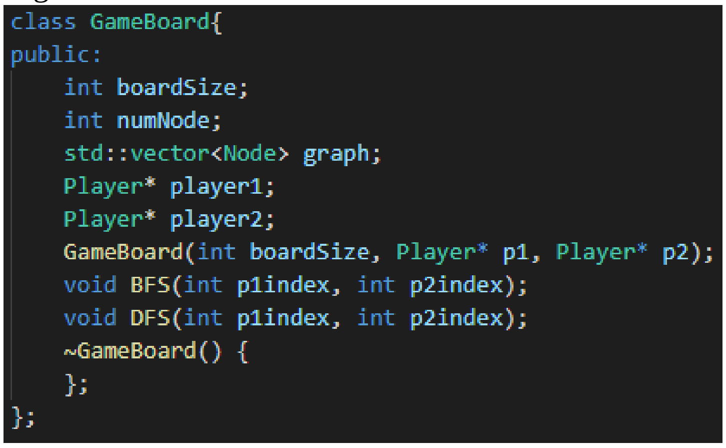
```cpp
class Player{
public:
    Coordinate2D init;
    int numShips;
    std::vector<Ship> ships;
    int shipArea;
    Player(Coordinate2D init, int numShips){
        this->init = init;
        this->numShips = numShips;
        this->shipArea = 0;
    }
    Player(const Player& other){
        this->init = other.init;
        this->numShips = other.numShips;
        this->ships = other.ships;
        this->shipArea = other.shipArea;
    }
    ~Player(){}
};
```

3.4 Node: This class represents nodes in the graph and cells in the game board. Every instance of it has an index which is equal to the cell number in the game board. Nodes have coordinates on the game board. Every node holds the number of ships of both players. Saying number of ships, I mean number of ship parts on a cell. Since every player can have more than one ship in a cell, we can't hold the information with Booleans. In addition to these, nodes hold the nodes to which it is connected. Edges are represented in this way in the graph.



```cpp
struct Node{
    int vertice_number;
    Coordinate2D coordinate;
    int player1ShipNumber;
    int player2ShipNumber;
    std::vector<Node*> connectedNodes;
};
```

3.5 GameBoard: This class represents the game board, holds the graph, has pointers to the players and has the algorithm functions.



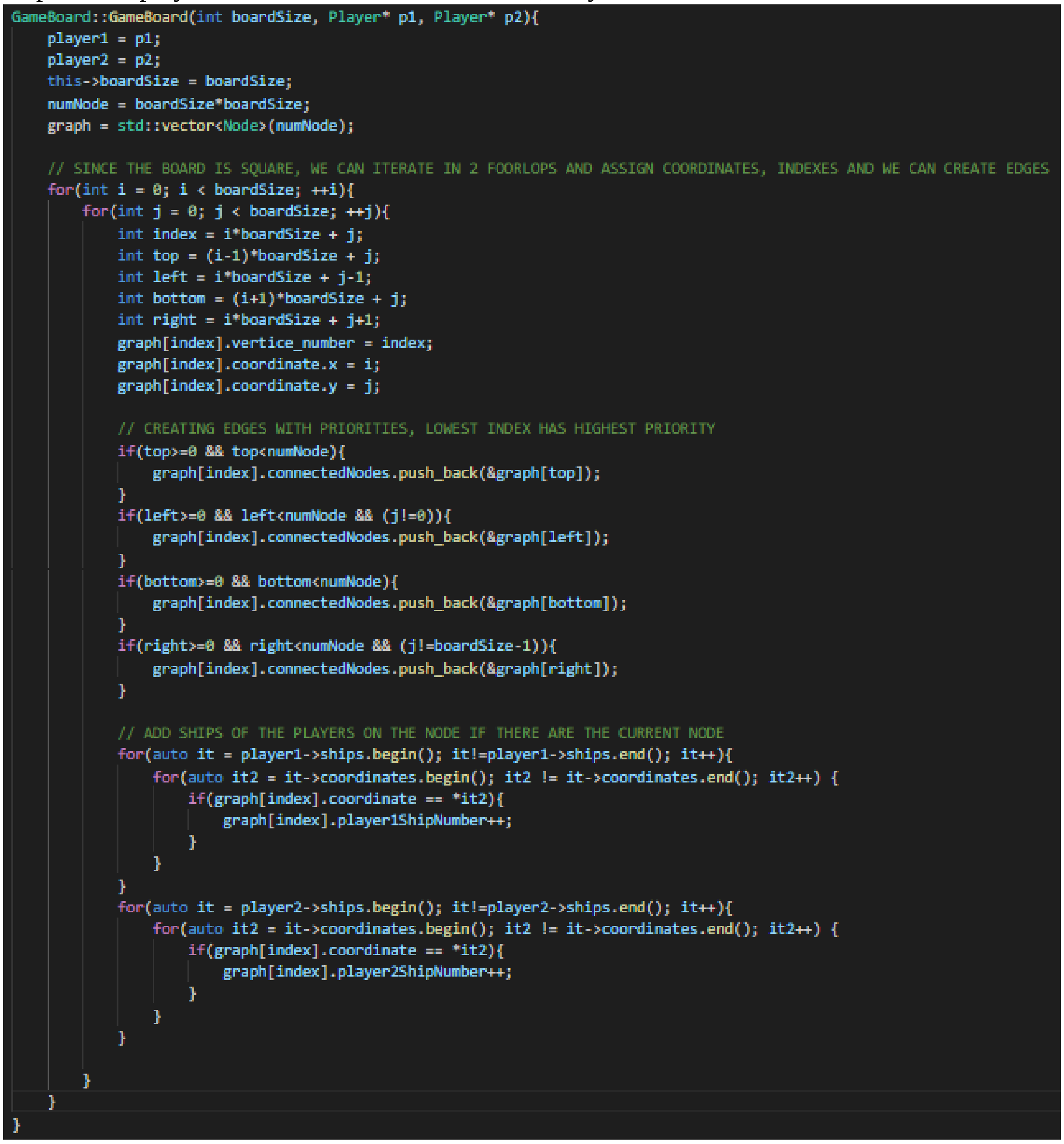
```cpp
class GameBoard{
public:
    int boardSize;
    int numNode;
    std::vector<Node> graph;
    Player* player1;
    Player* player2;
    GameBoard(int boardSize, Player* p1, Player* p2);
    void BFS(int p1index, int p2index);
    void DFS(int p1index, int p2index);
    ~GameBoard() {
    };
};
```

4. Functions:

4.1 int main(int argc, char** argv)

When the program is executed, it takes 2 input arguments from command line. These arguments are the input files which hold information about chosen algorithm, board size and players. These input files are read, players and game board objects are created and the algorithm which will be implemented are called.

4.2 gameBoard::GameBoard(int boardSize, Player* p1, Player* p2): Since graph creation happens here, I also wanted to explain this constructor. There are N^2 number of nodes in the graph are hold in a vector with size of N^2. Inside 2 nested for loop; coordinates, node indexes, edges and ships are implemented. Each node has edges with priorities. If indexes of top, left, bottom and right nodes are valid numbers, these edges are created. After that, if any ships of the players exist in the same coordinate, they will be added.



```cpp
GameBoard::GameBoard(int boardSize, Player* p1, Player* p2){
    player1 = p1;
    player2 = p2;
    this->boardSize = boardSize;
    numNode = boardSize*boardSize;
    graph = std::vector<Node>(numNode);

    // SINCE THE BOARD IS SQUARE, WE CAN ITERATE IN 2 FOORLOPS AND ASSIGN COORDINATES, INDEXES AND WE CAN CREATE EDGES
    for(int i = 0; i < boardSize; ++i){
        for(int j = 0; j < boardSize; ++j){
            int index = i*boardSize + j;
            int top = (i-1)*boardSize + j;
            int left = i*boardSize + j-1;
            int bottom = (i+1)*boardSize + j;
            int right = i*boardSize + j+1;
            graph[index].vertice_number = index;
            graph[index].coordinate.x = i;
            graph[index].coordinate.y = j;

            // CREATING EDGES WITH PRIORITIES, LOWEST INDEX HAS HIGHEST PRIORITY
            if(top>=0 && top<numNode){
                graph[index].connectedNodes.push_back(&graph[top]);
            }
            if(left>=0 && left<numNode && (j!=0)){
                graph[index].connectedNodes.push_back(&graph[left]);
            }
            if(bottom>=0 && bottom<numNode){
                graph[index].connectedNodes.push_back(&graph[bottom]);
            }
            if(right>=0 && right<numNode && (j!=boardSize-1)){
                graph[index].connectedNodes.push_back(&graph[right]);
            }

            // ADD SHIPS OF THE PLAYERS ON THE NODE IF THERE ARE THE CURRENT NODE
            for(auto it = player1->ships.begin(); it!=player1->ships.end(); it++){
                for(auto it2 = it->coordinates.begin(); it2 != it->coordinates.end(); it2++) {
                    if(graph[index].coordinate == *it2){
                        graph[index].player1ShipNumber++;
                    }
                }
            }
            for(auto it = player2->ships.begin(); it!=player2->ships.end(); it++){
                for(auto it2 = it->coordinates.begin(); it2 != it->coordinates.end(); it2++) {
                    if(graph[index].coordinate == *it2){
                        graph[index].player2ShipNumber++;
                    }
                }
            }

        }
    }
}
```

4.3 void gameBoard::BFS(int p1index, int p2index)

Let's say number of nodes in the graph is N and number of edges in the graph is M. Inside the while loop, we iterate all the nodes and in the for loop inside it we iterate all the edges in the graph for the worst case. So, time complexity of the algorithm is O(N+M).

```
PSEUDO CODE:
create queues for both player
create player1VisitedList, player2VisitedList, player1QueuedList, player2QueuedList
set number of visited nodes and number of nodes kept in memory 0 for both players
set player1win and player2win false
push player 1 node index to player 1 queue, player 2 node index to player 2 queue
increase number of nodes kept in memory for both players
create u for current node index for player 1 and v for player 2
create winner
while(queues are not empty and there are no winner){
    set u=front of the player 1 queue, v=front of the player 2 queue
    increase number of visited nodes for both players
    set player1VisitedList[u]=true, player2VisitedList[v]=true
    for(playerX in player1, player2){
        if(any opponent ships on the current node index for playerX){
            sink opponent ships, decrease opponent ship number
            if(there are no opponent ship on the table) {
                set playerXwin=true
                set winner=playerX
                break
            }
        }
    }
    pop node indexes from both player queues
    for(playerX in player1, player2){
        for(y in adjacents of the current node of the playerX){
            if(playerXVisitedList[y]=false and playerXQueuedList[y]=false){
                push y in playerX queue
                increase number of nodes kept in memory for playerX
                set playerXQueuedList[y]=true
            }
        }
    }
}
print number of visited nodes for both players
print number of nodes kept in memory for both players
print winner
```

4.4 void gameBoard::DFS(int p1index, int p2index)

Let's say number of nodes in the graph is N and number of edges in the graph is M. Inside the while loop, we iterate all the nodes and in the for loop inside it we iterate all the edges in the graph for the worst case. So, time complexity of the algorithm is O(N+M).

PSEUDO CODE:

```
create stacks for both player
create player1VisitedList, player2VisitedList
set number of visited nodes and number of nodes kept in memory 0 for both players
set player1win and player2win false
push player 1 node index to player 1 stack, player 2 node index to player 2 stack
increase number of nodes kept in memory for both players
create u for current node index for player 1 and v for player 2
create winner
while(stacks are not empty and there are no winner){
    set u=front of the player 1 queue, v=front of the player 2 queue
```

```
        increase number of visited nodes for both players
        set player1VisitedList[u]=true, player2VisitedList[v]=true
        for(playerX in player1, player2){
                if(any opponent ships on the current node index for playerX){
                        sink opponent ships, decrease opponent ship number
                        if(there are no opponent ship on the table) {
                                set playerXwin=true
                                set winner=playerX
                                break
                        }
                }
        }
        pop node indexes from both player queues
        for(playerX in player1, player2){
                for(y in adjacents of the current node of the playerX){
                        if(playerXVisitedList[y]=false){
                                push y in playerX stack
                                increase number of nodes kept in memory for playerX
                        }
                }
        }
}
print number of visited nodes for both players
print number of nodes kept in memory for both players
print winner
```

5. Questions:

5.1 Analyze and compare the algorithm results for "game2 and game3". They have the exact locations for ships but with different algorithms. Hence, you can compare BFS with DFS through game2's players vs. game3's players in terms of the number of visited nodes, the number of nodes kept in the memory and the running time.

	Game 0	**Game 1**	**Game 2**	**Game 3**	**Game 4**	**Game 5**
Algorithm	DFS	DFS	DFS	BFS	DFS	BFS
Player 1 Visited Node	2	31	42	48	16	16
Player 2 Visited Node	2	30	41	47	15	15
Player 1 Kept Memory Node	3	58	78	49	25	16
Player 2 Kept Memory Node	3	58	76	49	25	16
Running Time	0.6ms	0.8ms	0.82ms	0.85ms	0.75ms	0.73ms
Winner	Player2	Player1	Player1	Player1	Player1	Player1

Firstly, running times are obtained using chrono but in the submitted file time library is used because chrono is not compiled in ITU SSH servers. From the table, we can make an inference that running time is proportional to number of visited nodes. In BFS, we need to go to deeper levels in the BFS tree as ships get closer to corners of the game board. So, number of visited nodes may be lower than DFS for same game options. I used another list to keep queued nodes in BFS algorithm so number of nodes kept in memory can not exceed total number of nodes. I did this because I first add adjacent nodes in the queue and visit them one by one. If any two nodes in the same depth are connected to same non-visited node, we should not add this non-visited node to the queue. However, since we use stack in DFS, we can not implement the same list. As a result, number of nodes kept in memory can exceed total number of nodes. If we change the algorithm, if we use recursive function rather than stack, visit while exploring, number of nodes kept in memory will be equal to the number of visited nodes.

5.2 Why should we maintain a list of discovered nodes? How does this affect the outcome of the algorithms?

If we do not maintain a list of discovered, players go to same cell over and over again because they never know if they were there before. As a result, this creates a cycle. Since we check the number of sunk ships in the algorithms, game will end at some point for BFS. However, DFS algorithm may not end because we use stack to visit next nodes and we have priorities. Due to chosen priority, players will first go to top left corner first, then they will go to bottom left corner. After that the cycle will occur between top left and bottom left on the most left cells.

5.3 How does increasing board size while keeping the same number of ships on the board affect the memory complexity of the algorithms?

From the view of memory complexity, since we use adjacency list representation of graphs, it is O(n). If we double the number of nodes, n->2n, it will be O(2n)=O(n)