

# BLG336E – Homework I

## Report

- The program can be compiled by using this command line:  
`$ g++ -std=c++11 -o main`
- In order to run the program, you can use these command lines:  
`$ ./main part1 2`  
`$ ./main part2 5 bfs`    or    `$ ./main part2 5 dfs`  
`$ ./main part3 pikachu`    or    `$ ./main part3 blastoise`

### 1) Code Explanation

#### 1.a) Part 1

```
void Part1(Graph* nodeGraph, int maxLevel, bool print) {
    int currentLevel = 0;

    vector<Attack*> pikachuAttacks;
    vector<Attack*> blastoiseAttacks;

    ReadFile("pikachu.txt", &pikachuAttacks);
    ReadFile("blastoise.txt", &blastoiseAttacks);

    InitializeGraph(nodeGraph);

    while (currentLevel != maxLevel) {
        vector<Node*> allNodesInCurrLevel = nodeGraph->
        getAllNodesInLastLevel(currentLevel);
        char turn = (allNodesInCurrLevel[0]->getStats())->
        getTurn();           // getting the current turn
        if (turn == 'P')      // Pikachu is attacking
            TurnHandler(nodeGraph, 'P', currentLevel, allNodesInCurrLevel, pikachuAttacks);
        else                  // Blastoise is attacking
            TurnHandler(nodeGraph, 'B', currentLevel, allNodesInCurrLevel, blastoiseAttacks);
        currentLevel++;
    }

    if (print)
        nodeGraph->printLeaves();
}
```

```
for(vector<Attack*>::size_type i = 0; i != pikachuAttacks.size(); i++)  
    delete pikachuAttacks[i];           // deleting memory  
  
for(vector<Attack*>::size_type i = 0; i != blastoiseAttacks.size(); i++)  
    delete blastoiseAttacks[i];         // deleting memory  
}
```

The main function for the part1 is above. Firstly, the function reads the attacks for pikachu and blastoise from the text files and stores them in vectors separately. Then, the graph initialization is done.

```
void InitializeGraph(Graph* nodeGraph) {  
    Pokemon* Pikachu = new Pokemon(200, 100);  
    Pokemon* Blastoise = new Pokemon(200, 100);  
  
    Stats* stats = new Stats('P', 1, 0, true);  
  
    Node* root = new Node(Pikachu, Blastoise, stats, NULL, "", false);  
  
    nodeGraph->setRoot(root);  
    nodeGraph->increaseCounter(1);  
}
```

In InitializeGraph() function, the first node is created and set this node as the root of the graph. After the creation of the first node, the other levels are created in a while loop. In this while loop, the function first gets all the nodes in the current level in order to create child nodes for them. Each level has a turn value so that only that pokemon is able to attack its opponent. After getting the turn value, the function calls the TurnHandler() for the pokemon.

```
void TurnHandler(Graph* nodeGraph, char turn, int currentLevel, vector<Node*> all  
NodesInCurrLevel, vector<Attack*> allAttacks) {  
    for (vector<Node*>::size_type i = 0; i != allNodesInCurrLevel.size(); i++) {  
        Node *currNode = allNodesInCurrLevel[i];  
        int attackerHP = (turn == 'P') ? currNode->getPikachu()-  
>getHP() : currNode->getBlastoise()->getHP();  
        int attackerPP = (turn == 'P') ? currNode->getPikachu()-  
>getPP() : currNode->getBlastoise()->getPP();  
        if (attackerHP > 0) {           // if the attacker isn't dead, pokemon can at  
tack  
            for (vector<Attack*>::size_type j = 0; j != allAttacks.size(); j++) {  
                Attack* attack = allAttacks[j];  
                if (attack->getFirstUsage() <= currentLevel && (attackerPP >= attack->getPP() || attack->  
getName().compare("Skip") == 0)) {           // checking if the attack is usable  
                    if (attack->getAccuracy() == 100) {           // one possible node
```

```

        AttackOpponent(turn, currNode, attack, allAttacks, currentLevel, true);
        nodeGraph->increaseCounter(1); // a new node
    }
    else { // two possibilities
        AttackOpponent(turn, currNode, attack, allAttacks, currentLevel, true); // effective case

        AttackOpponent(turn, currNode, attack, allAttacks, currentLevel, false); // ineffective case
        nodeGraph->increaseCounter(2); // 2 new nodes
    }
}
else { // attack is not usable, skip it
    continue;
}
}
}
else if (nodeGraph->getWinPath(0) == 0 || nodeGraph->getWinPath(1) == 0) {
    if (turn == 'B' && nodeGraph->getWinPath(0) == 0)
        nodeGraph->setWinPath(0, currNode->getId());
    else if (turn == 'P' && nodeGraph->getWinPath(1) == 0)
        nodeGraph->setWinPath(1, currNode->getId());
}
}
}
}

```

The TurnHandler() function tries to create new children for the nodes in the current level. For each node, each attack is checked if it is applicable. If the attack is applicable AttackOpponent() function is called.

```

void AttackOpponent(char turn, Node* currNode, Attack* attack, vector<Attack*> allAttacks, int currentLevel, bool isEffective) {
    int blastoiseNewHP = 0, blastoiseNewPP = 0, pikachuNewHP = 0, pikachuNewPP = 0;
    char nextTurn = (turn == 'P') ? 'B' : 'P';

    if (turn == 'P') {
        blastoiseNewHP = (isEffective) ? currNode->getBlastoise()->getHP() - attack->getDamage() : currNode->getBlastoise()->getHP();
        blastoiseNewPP = currNode->getBlastoise()->getPP();

        pikachuNewHP = currNode->getPikachu()->getHP();
        pikachuNewPP = currNode->getPikachu()->getPP() + attack->getPP();
    }
}

```

```
else if (turn == 'B') {
    blastoiseNewHP = currNode->getBlastoise()->getHP();
    blastoiseNewPP = currNode->getBlastoise()->getPP() + attack->getPP();

    pikachuNewHP = (isEffective) ? currNode->getPikachu()->getHP() - attack-
>getDamage() : currNode->getPikachu()->getHP();
    pikachuNewPP = currNode->getPikachu()->getPP();
}

Pokemon* Pikachu = new Pokemon(pikachuNewHP, pikachuNewPP);
Pokemon* Blastoise = new Pokemon(blastoiseNewHP, blastoiseNewPP);

float prob = (isEffective) ? CalculateProb(allAttacks, currentLevel, attack-
>getAccuracy(), currNode->getStats()->getProb())
              : CalculateProb(allAttacks, currentLevel, 100 - a
ttack->getAccuracy(), currNode->getStats()->getProb());

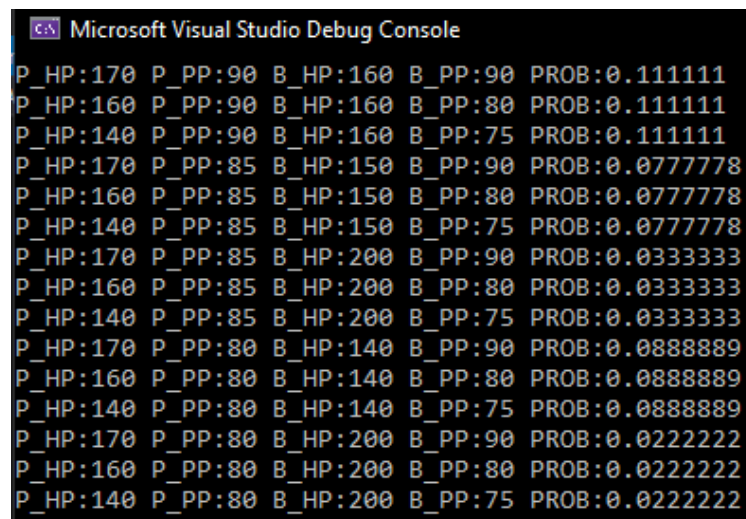
Stats* stats = new Stats(nextTurn, prob, currentLevel + 1, true);

Node* newNode = new Node(Pikachu, Blastoise, stats, currNode, attack-
>getName(), isEffective);
currNode->addChild(newNode);
currNode->getStats()->setIsLeaf(false);
}
```

The AttackOpponent() function takes the necessary parameters and creates a new child for the node. All the assignments for the new node are done here. That is all for the graph creation. After that, all leaves of the graph are printed on the console.

Example output:

\$ ./main part1 2



```
Microsoft Visual Studio Debug Console
P_HP:170 P_PP:90 B_HP:160 B_PP:90 PROB:0.111111
P_HP:160 P_PP:90 B_HP:160 B_PP:80 PROB:0.111111
P_HP:140 P_PP:90 B_HP:160 B_PP:75 PROB:0.111111
P_HP:170 P_PP:85 B_HP:150 B_PP:90 PROB:0.0777778
P_HP:160 P_PP:85 B_HP:150 B_PP:80 PROB:0.0777778
P_HP:140 P_PP:85 B_HP:150 B_PP:75 PROB:0.0777778
P_HP:170 P_PP:85 B_HP:200 B_PP:90 PROB:0.0333333
P_HP:160 P_PP:85 B_HP:200 B_PP:80 PROB:0.0333333
P_HP:140 P_PP:85 B_HP:200 B_PP:75 PROB:0.0333333
P_HP:170 P_PP:80 B_HP:140 B_PP:90 PROB:0.0888889
P_HP:160 P_PP:80 B_HP:140 B_PP:80 PROB:0.0888889
P_HP:140 P_PP:80 B_HP:140 B_PP:75 PROB:0.0888889
P_HP:170 P_PP:80 B_HP:200 B_PP:90 PROB:0.0222222
P_HP:160 P_PP:80 B_HP:200 B_PP:80 PROB:0.0222222
P_HP:140 P_PP:80 B_HP:200 B_PP:75 PROB:0.0222222
```

## 1.b) Part 2

```
void Part2(Graph* nodeGraph, int maxLevel, char* algorithm) {
    auto t1 = chrono::high_resolution_clock::now();
    if (strcmp(algorithm, "bfs") == 0) {
        nodeGraph->BFS();
    }
    else if (strcmp(algorithm, "dfs") == 0) {
        nodeGraph->DFS(0, 0);          // 0 mode: classic traverse
    }
    auto t2 = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(t2 - t1).count();
    cout << "Time taken by " << algorithm << " function is : " << fixed << (double)duration / 1000000 << " sec" << endl;
    cout << "Node count: " << nodeGraph->getCounter() << endl;
}
```

The main function for the part 2 is above. It just calls the function that is passed as a parameter and prints the time and node count.

```
void Graph::BFS() const {
    bool *visited = new bool[counter];
    for(int i = 0; i < counter; i++)
        visited[i] = false;

    list<Node*> queue;
    Node* node = root;
    int nodeId = node->getId();

    visited[nodeId] = true;
    queue.push_back(node);

    while(!queue.empty()) {
        node = queue.front();
        nodeId = node->getId();
        queue.pop_front();

        if (node->getChildrenCount() > 0){
            vector<Node*> children = node->getChildren();
            for (vector<Node*>::size_type i = 0; i != children.size(); i++) {
                int childId = children[i]->getId();
                if (!visited[childId]) {
                    visited[childId] = true;
                    queue.push_back(children[i]);
                }
            }
        }
    }
}
```

```
    }  
}  
delete[] visited;  
}
```

The BFS() function for the graph is above. In this function, in order to check the visited nodes and not to visit them more than once, I used a marking method and created a visited bool array. Then, starting from the root, I stored the nodes in a stack. For each node that is visited, I also pushed its children to the stack to visit them later. By doing so, the function traverses the graph level by level.

```
vector<Node*> Graph::DFS(int searchMode, int searchValue) {  
    vector<Node*> nodes;  
    bool found = false;  
    bool *visited = new bool[counter];    // marking all the vertices as not vi  
sited  
    for (int i = 0; i < counter; i++)  
        visited[i] = false;  
  
    DFSUtil(root, visited, &nodes, searchValue, &found, searchMode);  
    delete[] visited;    // deleting memory  
    return nodes;  
}
```

The DFS() function for the graph is above. The same method for marking the nodes that are visited is also used here. For the DFS(), the recursion method is used. For this purpose, I wrote a DFSUtil() recursive function.

```
void Graph::DFSUtil(Node* node, bool visited[], vector<Node*>* nodes, int searchV  
alue, bool* found, int searchMode) {  
    if (searchMode == 1 && *found)  
        return;  
  
    int nodeId = node->getId();  
    visited[nodeId] = true;  
  
    if (searchMode == 1) {  
        nodes->push_back(node);  
        if (nodeId == searchValue) {  
            *found = true;  
            return;  
        }  
    }  
    else if (searchMode == 2 && node->getStats()->getIsLeaf())  
        nodes->push_back(node);  
    else if (searchMode == 3 && node->getStats()->getLevel() == searchValue)
```

```
nodes->push_back(node);

if (node->getChildrenCount() > 0) {
    vector<Node*> children = node->getChildren();

    for (vector<Node*>::size_type i = 0; i != children.size(); i++)
        if (!visited[children[i]->getId()]) {
            DFSUtil(children[i], visited, nodes, searchValue, found, searchMo
de);

            if (searchMode == 1) {
                if (*found) return;
                else nodes->pop_back();
            }
        }
    }
}
```

The DFSUtil() function does not do only traverse operation. It also makes various search operations. For this purpose, there are multiple search modes.

|                |   |   |
|----------------|---|---|
| SearchMode = 0 | : | classic graph traverse                          |
| SearchMode = 1 | : | finds a leaf and returns its path from the root |
| SearchMode = 2 | : | finds all leaves and returns them               |
| SearchMode = 3 | : | finds all the nodes in given level              |

Let's examine how DFSUtil() function works. For each node, its children are found and for the first child, the DFSUtil() function is called again. By doing so, the function goes directly to the leaf and continues from there. As the function's name indicates, the focus of the algorithm is to go deeper until a node without children is found.

Example outputs:

\$ ./main part2 8 bfs

```
Microsoft Visual Studio Debug Console
Time taken by bfs function is : 0.483017 sec
Node count: 215796
```

\$ ./main part2 8 dfs

```
Microsoft Visual Studio Debug Console
Time taken by dfs function is : 0.143237 sec
Node count: 215796
```

### 1.c) Part 3

```
void Part3(Graph* nodeGraph, char* pokemon) {
    if (strcmp(pokemon, "pikachu") == 0) {
        vector<Node*> pikachuWinPath = nodeGraph->DFS(1, nodeGraph-
>getWinPath(0));    // 0 for pikachu
        cout << "Pikachu's win:" << endl;
        if (pikachuWinPath.size() > 1)
            PrintWinPath(pikachuWinPath);
        else
            cout << "Could not win in given maxLevel." << endl;
    }
    else if (strcmp(pokemon, "blastoise") == 0) {
        vector<Node*> blastoiseWinPath = nodeGraph->DFS(1, nodeGraph-
>getWinPath(1));    // 1 for blastoise
        cout << "Blastoise's win:" << endl;
        if (blastoiseWinPath.size() > 1)
            PrintWinPath(blastoiseWinPath);
        else
            cout << "Could not win in given maxLevel." << endl;
    }
    else
        cout << "Wrong pokemon name. Try again." << endl;
}
```

The function that handles part 3 is above. In part 1, while creating the graph, the function also checks the nodes if they have a pokemon with 0 HP value and stores the first nodes that are found in the graph. By doing so, the program finds the shortest path for both pokemons. After they are found, in part3, we can get their path using DFS function with searchMode=1 and print all the nodes on the console.

Example outputs:

\$ ./main part3 pikachu

```
Microsoft Visual Studio Debug Console
Pikachu's win:
Pokemon used Thundershock. It's effective. (P:200, B:160)
Blastoise used Tackle. It's effective. (P:170, B:160)
Pokemon used Thundershock. It's effective. (P:170, B:120)
Blastoise used Tackle. It's effective. (P:140, B:120)
Pokemon used Slam. It's effective. (P:140, B:60)
Blastoise used Tackle. It's effective. (P:110, B:60)
Pokemon used Slam. It's effective. (P:110, B:0)
Level count: 7
Probability: 9.25926e-05
```



\$ ./main part3 blastoise

```
Microsoft Visual Studio Debug Console
Blastoise's win:
Pokemon used Thundershock. It's effective. (P:200, B:160)
Blastoise used Tackle. It's effective. (P:170, B:160)
Pokemon used Thundershock. It's effective. (P:170, B:120)
Blastoise used Bite. It's effective. (P:110, B:120)
Pokemon used Thundershock. It's effective. (P:110, B:80)
Blastoise used Bite. It's effective. (P:50, B:80)
Pokemon used Thundershock. It's effective. (P:50, B:40)
Blastoise used Bite. It's effective. (P:-10, B:40)
Level count: 8
Probability: 3.6169e-05
```

## 2) Code Analysis

First of all, let's look at the differences that BFS and DFS algorithms have.

The BFS algorithm traverses all the nodes level by level and uses a queue for it. On the other hand, the DFS algorithm traverses the graph by focusing to go deeper in the graph and uses stack for it. Thus, if the node that we want to reach is quite away from the root, using the DFS is more sensible. It most likely will find it before DFS does. However, for the short distances from the root, the DFS can be the way to go.

In this homework, we mainly worked on the leaves of the graph since we are trying to create nodes for the leaves until the required level is reached. As the level gets higher, the distance between the root and the leaves increases. For this reason, I chose to use DFS for some operations like finding all leaves or finding all the nodes in the last level. If we run the part 2, we can easily see the difference.

```
Microsoft Visual Studio Debug Console
Time taken by bfs function is : 0.111200 sec
Node count: 52596
```

```
Microsoft Visual Studio Debug Console
Time taken by dfs function is : 0.030103 sec
Node count: 52596
```

The DFS performs almost 4 times faster than the BFS for a graph with 7 levels.

Now, let's analyze the functions line by line.

|   | Statement                          | Steps/<br>execution | frequency |          | Total steps |          |
|---|------------------------------------|---------------------|-----------|----------|-------------|----------|
|   |                                    |                     | If-true   | If-false | If-true     | If-false |
| 1 | void Graph::BFS() const            |                     |           |          |             |          |
| 2 | {                                  | 0                   | -         | -        | 0           | 0        |
| 3 | bool *visited = new bool[counter]; | 1                   | 1         | 1        | 1           | 1        |
| 4 | for(int i = 0; i < counter; i++)   | 1                   | n+1       | n+1      | n+1         | n+1      |
| 5 | visited[i] = false;                | 1                   | n         | n        | n           | n+1      |

|        |  |   |     |     |          |      |
|--------|--|---|-----|-----|----------|------|
| 6      | list<Node*> queue;<br>Node* node = root;<br>int nodeId = node->getId();<br>visited[nodeId] = true;<br>queue.push_back(node); | 5 | 1   | 1   | 5        | 5    |
| 7      | while(!queue.empty()) {  | 1 | n   | 0   | n        | 0    |
| 8      | node = queue.front();<br>nodeId = node->getId();<br>queue.pop_front();   | 3 | n   | 0   | 3*n      | 0    |
| 9      | if (node->getChildrenCount() > 0) {  | 1 | n   | 0   | n        | 0    |
| 10     | vector<Node*> children = node->getChildren();  | 1 | n   | 0   | n        | 0    |
| 11     | for (size_type i = 0; i != children.size(); i++) {   | 1 | x+1 | x+1 | n        | 0    |
| 12     | int childId = children[i]->getId();  | 1 | n*x | 0   | n*x      | 0    |
| 13     | if (!visited[childId]) {   | 1 | n*x | 0   | n*x      | 0    |
| 14     | visited[childId] = true;<br>queue.push_back(children[i]);  | 2 | n*x | 0   | 2*n*x    | 0    |
| 15     | }  |   |     |     |          |      |
| 16     | }  |   |     |     |          |      |
| 17     | }  |   |     |     |          |      |
| 18     | }  |   |     |     |          |      |
| 19     | delete[] visited;  | 1 | 1   | 1   | 1        | 1    |
| 20     | }  |   |     |     |          |      |
| Total: |  |   |     |     | 4nx+9n+8 | 2n+9 |

n = number of nodes,                      x = number of attacks

For the BFS() function:

$$O(n) = 4*n*x + 9n + 8$$

$$O(n) = n*x + n$$

The number of nodes times number of attacks gives us the number of edges in the graph. Let's call this value e, number of edges. Hence,

$$O(n, e) = n + e$$

The BSP function's running time depends on the number of edges and nodes.

|   | Statement  | Steps/<br>execution | frequency |          | Total steps |          |
|---|--|---------------------|-----------|----------|-------------|----------|
|   |  |                     | If-true   | If-false | If-true     | If-false |
| 1 | void Graph::DFSUtil(Node* node, bool visited[]) {      |                     |           |          |             |          |
| 2 | int nodeId = node->getId();<br>visited[nodeId] = true; | 2                   | 1         | 1        | 2           | 2        |
| 3 | if (node->getChildrenCount() > 0) {                    | 1                   | 1         | 1        | 1           | 1        |

|        |  |   |     |   |        |   |
|--------|--|---|-----|---|--------|---|
| 4      | vector<Node*> children = node->getChildren();    | 1 | 1   | 0 | 1      | 0 |
| 5      | for (size_type i = 0; i != children.size(); i++) | 1 | a+1 | 0 | a+1    | 0 |
| 6      | if (!visited[children[i]->getId()])              | 1 | a   | 0 | a      | 0 |
| 7      | DFSUtil(children[i], visited);                   | x | a   | 0 | a*x    | 0 |
| 8      | }  |   |     |   |        |   |
| Total: |  |   |     |   | ax+a+5 | 3 |

a = number of attacks

For the DFSUtil() function, each node will be visited once since we mark the visited nodes. For each visited node, we call the same DFSUtil() function for its children if it has any. That means we check also every edge of the node. In the end:

$$O(n) = n + e, \quad n = \# \text{ of nodes}, e = \# \text{ of edges}$$

|        | Statement                          | Steps/<br>execution | frequency |          | Total steps |          |
|--------|------------------------------------|---------------------|-----------|----------|-------------|----------|
|        |                                    |                     | If-true   | If-false | If-true     | If-false |
| 1      | void Graph::DFS() {                |                     |           |          |             |          |
| 2      | bool *visited = new bool[counter]; | 1                   | 1         | 1        | 1           | 1        |
| 3      | for (int i = 0; i < counter; i++)  | 1                   | n+1       | n+1      | n+1         | n+1      |
| 4      | visited[i] = false;                | 1                   | n         | n        | n           | n        |
| 5      | DFSUtil(root, visited);            | 1                   | n+e       | n+e      | n+e         | n+e      |
| 6      | delete[] visited;                  | 1                   | 1         | 1        | 1           | 1        |
| 8      | }                                  |                     |           |          |             |          |
| Total: |                                    |                     |           |          | 3n+e+3      | 3n+e+3   |

Finally, for the DFS() function:

$$O(n) = n + e$$

Thus, the running time of DFS() function depends on the number of nodes and edges again.