Probabilistic GameState Strategy Engine

A Pokémon Battle Simulation
BLG 223E - Data Structures
Istanbul Technical University
2024

Note: Personal and professor-specific details have been removed to protect privacy. This project demonstrates a strategic simulation using a custom graph structure.

Introduction

I was supposed to simulate a Pokemon battle where a Pikachu is fighting against a Blastoise. For that problem I meant to simplify the Pokemon fighting procedures according to the game rules, values and limitations given in the assignments. In the simplified procedure, there are two attributes of each Pokemon: HP and PP. HP (Health Points): Total health of a Pokemon. May be decreased after attack of the opponent. No defense mechanism is included. I set The HPs of Pikachu and Blastoise are both 200 HP in case of low RAM space while creating the graph. Total ability to do an attack. Each Pokemon starts with a total of 100 PP and it decreases according to each attack's PP values. In struct Pokemon there are pp, hp and name attributes need to be updated for all game states.

```
// Initialize with 100 PP and 200 HP
pokemon* pikachu = new pokemon("Pikachu", 100, 200);
pokemon* blastoise = new pokemon("Blastoise", 100, 200);
```

Attacks:

Also for an attack there are four different attributes:
PP: To do an attack, PP value of the selected attack should be decreased from Pokemon's own PP. If this value is greater than the Pokemon's PP, attack cannot be used.
Accuracy: Accuracy of an attack. If it is not an attack with 100% accuracy, multiple nodes should be created in the graph.
Damage: Damage of the attack.
FirstUsage: The first level of the graph where the attack may be used.

```
    pikachu->attacks.addBack(new attack("Thundershock", -10, 100, 40, 0));
    pikachu->attacks.addBack(new attack("Skull Bash", -15, 70, 50, 0));
    pikachu->attacks.addBack(new attack("Slam", -20, 80, 60, 0));
    pikachu->attacks.addBack(new attack("Skip", 100, 100, 0, 3)); // Skip
attack restores PP

    blastoise->attacks.addBack(new attack("Tackle", -10, 100, 30, 0));
    blastoise->attacks.addBack(new attack("Water Gun", -20, 100, 40, 0));
    blastoise->attacks.addBack(new attack("Bite", -25, 100, 60, 0));
    blastoise->attacks.addBack(new attack("Skip", 100, 100, 0, 3)); // Skip
attack restores PP
```

In abstract, as initialization, Pikachu and Blastoise were initialized with their respective HP and PP values. Defined a set of attacks for both Pokémon with attributes such as PP cost,

accuracy, damage, and first usage level. In Graph Expansion, Implemented the expandGraph function to create new game states based on the current state's possible actions. Calculated the probabilities of each action being effective or not based on the attack's accuracy. Created child nodes for each possible action and updated the Pokémon's HP and PP accordingly. Pathfinding: Developed the findShortestWinningPath function to identify the shortest path to a winning state using a Breadth-First Search (BFS) algorithm. Printed the sequence of actions and their effects on the Pokémon's HP and PP, validating the simulation's accuracy. The DoublyList structure is used as a queue, consistent with BFS. Nodes are added to the back of the list and processed from the front. Each node represents a game state, and child nodes represent possible future states based on available actions. By processing nodes in a BFS manner, the simulation ensures that all nodes at the current depth level are fully expanded before moving to the next level, providing a complete and level-wise exploration of possible game states.

Addition to Node struct

Node Constructor And New Ame State Creation For Next Game State

I defined the constructor in the node struct as below with its attributes to the initial default template of skeleton file.

```
    node(pokemon* p, pokemon* b, char s = 'P', int lvl = 0, bool leaf =
false, double probability = 1.0, node* par = nullptr)
        : pikachu(p), blastoise(b), status(s), level(lvl), isleaf(leaf),
prob(probability), parent(par) {
        ifef = "";
        curattack = "";
        num = 0;
    }
```

Next Action Creation and probability calculation

In the simulation, it is taught that Pokemons are not using a decision mechanism to select an attack. Thus, for a list of attacks, it is equally likely to select one of them.

The types of attacks Pikachu can use according to the game rules until level 3 are Slam, Bash, and Thundershock. When Slam is used, there are two possible outcomes: it can be either effective or not effective. Similarly, when Bash is used, it can also be either effective or not effective. However, if Thundershock is used, it is always effective. Each of these attacks reduces Pikachu's PP and, when effective, reduces Blastoise's HP.

When it is Blastoise's turn, it can use Tackle, Watergun, and Bite until level 3, and these are always effective because their accuracy attributes are 100%. These attacks can be used from the first level since their first usage attribute is set to 0. However, the Skip attack has a first usage attribute of 3, meaning it can only be used from level 3 onwards by both Pokémon.

Thus, when it is a spesific Pokémon's turn, it has specific attacks it can use at that level. After each attack, a separate node should be created to represent the next game state and its probability should be calculated.

For example, at level 0, Pikachu can use three attacks. One of them is Thundershock, which has an accuracy of 100%, so only one node is created with an effective attribute and its probability is 1/3, approximately 0.33. If Pikachu uses Slam, it creates two nodes because its accuracy is not 100%, but 80%. The probabilities for these nodes are different. The node with the effective attribute is calculated as (1/3 * 80/100) and the non-effective node as (1/3 * 20/100). The other attack that can be used at level 0 by Pikachu is Skullbash, and since its accuracy is not 100%, it also creates two possible nodes, calculated as (1/3 * 70/100) or (1/3 * 30/100). From the initial node at level 0, there are 5 possible moves, resulting in 5 nodes.

These 5 nodes are added to the doubly linked list, where they are checked to see if they are at the maximum level or a leaf node. If checked nodeis not leaf_node or not maximum depth level achieved, then all the next possible game states are expanded in the graph as that node’s children.

The probability calculation is not solely dependent on the accuracy attribute. The probability of a node being formed from a previous node so is as showed below:

[The previous node's probability] * [1 / the number of usable attacks for the Pokémon on that turn) * (The probability of being ineffective or effective based on the accuracy attribute). (For attacks with 100% accuracy, this value is directly 1.)

In expandGraph Function:
The expandGraph function generates all possible next states from the current node based on the available attacks and their effects. For each possible attack, new child nodes are created, representing the game state after the attack. These child nodes are then added to the DoublyList, ensuring they will be processed in subsequent iterations of the expand function.

To calculate the probaility firstly I defined attacknode from the curent game state's whicever pokemon's turn getting that pokemon's attacks

```
DoublyList<attack*> attacks = (status == 'P') ? pikachu->attacks : blastoise-
>attacks;
Node<attack*>* attackNode = attacks.head;
```

Then I defined availableAttacksCount variable integer and a tempNode to count available attacks specific for that game state considering also the first usage attribute of the attacks. So I counted the attack only if it is available for the spesfic level and specific pokemon character. Because it consider the case for the child node, 1 should be added and (level + 1) should be compared with the first usage value.

```
int availableAttacksCount = 0;
    Node<attack*>* tempNode = attacks.head;
    while (tempNode != NULL) {
        if (level + 1 >= tempNode->data->get_first()) {
            availableAttacksCount++;
        }
        tempNode = tempNode->next;
    }
```

The aim to count these is to use it in probability calculation in the next code block.

```
double childProb = prob * (1.0 / availableAttacksCount) * (isEffective ?
effectiveProbability : notEffectiveProbability);
```

The current game'state's probability "prob " and the probability of whether being effective are also included in the concluded probabilty of the child node(the next game state's actualisation probability). I calculated as below.

```
double effectiveProbability = act->get_accuracy() / 100.0;
double notEffectiveProbability = 1.0 - effectiveProbability;
```

While creating the next game state I managed the actiond using attack pointer act. If the pokemon in turn has not enough powerpoint to use the related attack's required pp, in the next game state that action cannot be used, therefore attackNode will iterate to the next available action that can be taken without using creating a child node with that action.

```
    while (attackNode != NULL) {
        attack* act = attackNode->data;

        if ((status == 'P' && pikachu->pp < act->get_pp()) || (status == 'B' &&
blastoise->pp < act->get_pp())) {
            attackNode = attackNode->next;
            continue;
        }
```

With the same logic, if the first usage of the action does not met the attacknode iterate to the next available action that can be take in that Pokemon's attacklist. Because it consider the case for the child node, 1 should be added and (level + 1) should be compared with the first usage value.

```
        // Ensure the attack can be used starting from its first usage level
        if (level + 1 < act->get_first()) {
            attackNode = attackNode->next;
            continue;
        }
```

I used the constructor to create new game state as the child of current game state in expand graph function.

```
 node* newNode = new node(newPika, newBlas, status == 'P' ? 'B' : 'P',
level + 1, newIsLeaf, childProb, this);
```

Then I also set the ifef and attac name attributes of the newly created game state node.

```
newNode->curattack = act->get_name();
newNode->ifef = isEffective ? "effective" : "not effective";
```

The attributes passed into the node constructor "newPika and newBlas" includes their own hp and pp values which also calculated for the next game state in the code block below. I wrote in the expand graph function to set the pokemons' new powerpoints and new health points after taking the specified action by specified pokemon against specified pokemon.

```
      pokemon* newPika = new pokemon(*pikachu);
            pokemon* newBlas = new pokemon(*blastoise);
```

```
            if (status == 'P') {
                newPika->pp += act->get_pp();
                if (isEffective) newBlas->hp -= act->get_damage();

            } else {
                newBlas->pp += act->get_pp();
                if (isEffective) newPika->hp -= act->get_damage();
            }
```

From the node constructor attributes, 'status' of New game state is determined with the expression ' status == 'P' ? 'B' : 'P'' in the node instance creation. Also 'parent*' attribute is set while creating the new node game state using 'this' .

Leaf Node Definition

A node is a leaf node, if one of the competitors are knocked-out or the level limit of the tree is reached. No children of a leaf node is allowed.

```
bool newIsLeaf = (newPika->hp <= 0 || newBlas->hp <= 0 || level + 1 == maxDepth);
```

End condition to expandGraph Function:

As it is requested in the assignment file, while creating tree from the game state possibilities, in case of the current referred game state is reached the maxDepth limitation or being a leaf node, the expandGraph function will stop actualising more with the code line which checkes this condition below.

```
if (isleaf || level >= Depth) return;
```

Addition to the struct Graph

The graph is initialized with a root node representing the initial state of the game. The root node is added to the DoublyList, which acts as a queue.

I added root node pointer and a nodes Doublylist consisting of node gamestates. The aim is to use nodes doublylist to keep nodes in the order of BFS. I wrote a graph constructor including root node. When a graph created it created in the main function,it created a Doublylist of nodes then it adds root node which is out beginning state of the game.

```
  node* root;
    // DoublyList, düğümleri BFS sırasına göre saklamak için kullanılır
    DoublyList<node*> nodes;

    // Constructor
```

```
    graph(pokemon* p, pokemon* b) {
        root = new node(p, b);  // root node oluşturuluyor
        nodes.addBack(root);    // root node listeye ekleniyor
    }
```

I added a expand function to the struct graph. It takes the depth as input from the console in part1 and just printed to consol the nodes in that depth. But in part2 the depth value is entered manually in the skeleton file main function as 9, because the target of the prject in part 2 is to find the earliest winner node and the earliest path. So I pass a bool parameter to expand function to control print condition according to parts.

As the method's functionality, it removes the node in the top of Doublylist first
then it checks the noded starting from root in Doublylist and create its child node by calling the expandGraph function, if the referred level of the checked node compatible with depth entered input from console it prints these nodes to console in part1. In part2 it creates 9 depth of tree consisting of possible game states.

The expand function processes nodes in a BFS manner. It removes nodes from the front of the DoublyList (queue) and expands them to generate child nodes.

This process continues until the DoublyList is empty, meaning all possible game states up to the specified depth have been explored.

```
    void expand(int Depth, bool shouldPrint = true) {
    while (nodes.head != NULL) {
        node* current = nodes.head->data;
        nodes.removeFront();
        if (!current->isleaf && current->level < Depth) {
            current->expandGraph(nodes, Depth);
            if (shouldPrint && current->level == Depth - 1) {
                printNode(current);
            }
        }
    }
}
```

When I create a graph with name g in main function, I call expand function, then it calls expandGraph function defined in struct node and printNodefunction defined in struct graph. The print node printed the powerpoint healtpoint and game state probability attributes og

Pokemons. This allowed me to track the altering game states according to taken actions and actions' attribute values.

```
    void printNode(node* n) {
    // Assuming you will handle output prefixes outside this function or you
adjust this as needed
    cout << "P_HP:" << n->pikachu->hp << " P_PP:" << n->pikachu->pp;
    cout << " B_HP:" << n->blastoise->hp << " B_PP:" << n->blastoise->pp << "
PROB:" << n->prob << endl;
}
```

findShortestWinningPath Funtion

findShortestWinningPath function applies BFS and checks all nodes starting from the lowest level to the greatest level until reaching the winner node with earliest path. It uses queue to actualise it. As a helper, I defined a parent pointer in the node struct to be able to reach the path of the winner node after finding the earliest paths for both of Pokemons.

In function, firstly I defined a queue from Doublylist, then added the root node in it from the back.

```
DoublyList<node*> queue;
queue.addBack(root);
```

Then I defined the shortestWin node pointer and a shortestDepth integer value initialized with a value greater then any available depth in the tree.

```
node* shortestWin = nullptr;
int shortestDepth = 2147483647;
```

I wrote a BFS algorithm to find the winner game state wth the earliest path. According to the states I defined a bool value which is "isWinnignNode". If the case is blastoise: the searcehed node should met the condition while blastoise's healthpoint is greater than 0, pikachu's healtpoint's should be less than or equal to 0 zero. Else If the case is pikachu: the searcehed node should met the condition while pikachu's healthpoint is greater than 0, pikachu's healtpoint's should be less than or equal to 0 zero.

```
bool isWinningNode = (participant == "pikachu" && current->blastoise->hp
<= 0 && current->pikachu->hp > 0) || (participant == "blastoise" &&
current->pikachu->hp <= 0 && current->blastoise->hp > 0);
```

I preferred to manage the process in a while loop with a NULL condition of the queue during BFS operation.

I defined a node pointer with a name of ‘current’ to use in check process of the processed node. After getting the node element from th etop of queue to the current node, it removes that element from the queue.

```
while (queue.head != NULL) {
        node* current = queue.get(0);
        queue.removeFront();
```

If the node is a leaf node and the winning node it checks the level of the node whether its level is less than shortestDepth value defined with a great value in the beginning and updated with fisrt discovery of the winner node, as all teh odes are checked, the shortestWin node will ahve chance to be updated if the new winner node is discovered with a smaller level so is earliest path. So I recorded also the shortestDepth integer value and updated with iterations.

```
            if (current->isleaf && isWinningNode) {
                if (current->level < shortestDepth) {
                    shortestWin = current;
                    shortestDepth = current->level;
                }
            }
```

Then I defined A Node with a type of node pointer called ‘childNode’ to acces current processed node’s child nodes. With checking if its child nodes exist, its all childs from the list will be added to the queue in the while loop. These added nodes in the queue will be processed in the next iteration of while loop.

While explaining these steps, I can prove also from this perspective, the next level’s children nodes are added to the doublylist from the back and while processing check operation to the current node the node is taken from the top of the doublylist, the doublylist is used as a queue.

```
            Node<node*>* childNode = current->child.head;
            while (childNode != NULL) {
                queue.addBack(childNode->data);
                childNode = childNode->next;
            }
```

After finding shortestWin node, to store the pthe earliest path in it, I created doublylist witha type of node with a name bestPath. Within a while loop, using iteration, I add all parents in an order from end to the beginning from the front side of soublylist. Then passed these togethe as a parameter of printWinPath function.

```
if (shortestWin != nullptr) {
            DoublyList<node*> bestPath;
            node* current = shortestWin;
            while (current != nullptr) {
                bestPath.addFront(current);
                current = current->parent;
            }
            printWinPath(bestPath, shortestWin);
        } else {
            cout << "No winning path found." << endl;
        }
```

In printWinPath function which is a member of graph struct, I pritend the probability of the winner node first.
I defined the Node pointer with type of node to iterate path nodes over bestpath. Because the attack names and ifef attributes are recorded in the resulted nodes, and the pokemon acts in previous turn named in previous node, I had to set pokemon names according to status. Whicever node I am prinitng, the acting pokemon should be the previous node's status. Then I iterated paths in a while loop with null condition and also to keep results clear and undertandable, I added the if condition to express the beginning node there is no attack or efficient attribute value yet so just "Beginning state: No action Yet". I wrote altering analse of all turns until the winner node to see how that actions in that turns caused this winning state in detail.

```
void printWinPath(DoublyList<node*>& bestPath, node* winner) {
    cout << "Winning path with probability: " << winner->prob << endl <<
endl ;
    Node<node*>* n = bestPath.head;
    while (n != NULL) {

        string currentPokemon = (n->data->status == 'P') ? "Blastoise"  :
"Pikachu";  // To check the turn in actions
        if( n->data->curattack !=""){
        cout << currentPokemon << " used  " << n->data->curattack << " : "
<< n->data->ifef << endl;
        }else{
            cout<<"Beginning state: No action Yet"<<endl;
        }
```

```
        cout << "Level : " <<n->data->level << " Pikachu HP: "  << n-
>data->pikachu->hp << " PP: " << n->data->pikachu->pp;
        cout << ", Blastoise HP: " << n->data->blastoise->hp << " PP: " <<
n->data->blastoise->pp << endl << endl;
        n = n->next;
    }
    // Print the winner node
    if (winner != NULL) {

        cout << "Final state at Level " << winner->level << ": Pikachu HP:
" << winner->pikachu->hp << ", PP: " << winner->pikachu->pp
             << "; Blastoise HP: " << winner->blastoise->hp << ", PP: " <<
winner->blastoise->pp << endl;
    }
}
```

Outputs

In main function, I create a graph with name g, in part1 I call expand function, then it calls expandGraph function and printNodefunction within that in part 1. Below how my promram can be compiled and run.

Part-1 Outputs

```
test@vm_docker:~/hostVolume/data_hw2$ g++ ./skeleton.cpp
-o skeleton

test@vm_docker:~/hostVolume/data_hw2$ ./skeleton part1 1
P HP:200 P PP:100 B HP:200 B PP:100 PROB:1

test@vm_docker:~/hostVolume/data_hw2$ ./skeleton part1 2
P HP:200 P PP:90 B HP:160 B PP:100 PROB:0.333333
P HP:200 P PP:85 B HP:150 B PP:100 PROB:0.233333
P HP:200 P PP:85 B HP:200 B PP:100 PROB:0.1
P HP:200 P PP:80 B HP:140 B PP:100 PROB:0.266667
P HP:200 P PP:80 B HP:200 B PP:100 PROB:0.0666667

test@vm_docker:~/hostVolume/data_hw2$ ./skeleton part1 3
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
```

It is as requested in the assignment file, it calculates the game state's probabilities and PP HP values after actions taken from previous nodes correctly.

```
.\main part1 0
P_HP:200 P_PP:100 B_HP:200 B_PP:100 PROB:1
```

```
.\main part1 1
P_HP:200 P_PP:90 B_HP:160 B_PP:100 PROB:0.333333
P_HP:200 P_PP:85 B_HP:150 B_PP:100 PROB:0.233333
P_HP:200 P_PP:85 B_HP:200 B_PP:100 PROB:0.1
P_HP:200 P_PP:80 B_HP:140 B_PP:100 PROB:0.266667
P_HP:200 P_PP:80 B_HP:200 B_PP:100 PROB:0.0666667
```

```
.\main part1 2
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
```

Part-2 Outputs

test@vm_docker:~/hostVolume/data_hw2$ ./skeleton part2 pikachu

Winning path with probability: 6.94444e-05

Beginning state: No action Yet
Level : 0 Pikachu HP: 200 PP: 100, Blastoise HP: 200 PP: 100

Pikachu used Thundershock : effective
Level : 1 Pikachu HP: 200 PP: 90, Blastoise HP: 160 PP: 100

Blastoise used Tackle : effective
Level : 2 Pikachu HP: 170 PP: 90, Blastoise HP: 160 PP: 90

Pikachu used Thundershock : effective
Level : 3 Pikachu HP: 170 PP: 80, Blastoise HP: 120 PP: 90

Blastoise used Tackle : effective
Level : 4 Pikachu HP: 140 PP: 80, Blastoise HP: 120 PP: 80

```
Pikachu used  Slam : effective
Level : 5 Pikachu HP: 140 PP: 60, Blastoise HP: 60 PP: 80

Blastoise used  Tackle : effective
Level : 6 Pikachu HP: 110 PP: 60, Blastoise HP: 60 PP: 70

Pikachu used  Slam : effective
Level : 7 Pikachu HP: 110 PP: 40, Blastoise HP: 0 PP: 70

Final state at Level 7: Pikachu HP: 110, PP: 40; Blastoise HP: 0, PP: 70
```

Pikachu's winning state with earliest path is in level 7.
Pikachu still has HP value of 110 and Blastoise has not any, 0.

As I printed in detail, I check every altering state with the related actions and their effects on HP and PP values. By following the path and recording the effects of each action, I can confirm the accuracy of the simulation and the correctness of the resulting states after each move below.

Pikachu used Thundershock (Effective):
Pikachu's PP decreased by 10, Blastoise's HP decreased by 40.

Blastoise used Tackle (Effective):
Blastoise's PP decreased by 10, Pikachu's HP decreased by 30.

Pikachu used Thundershock (Effective):
Pikachu's PP decreased by 10, Blastoise's HP decreased by 40.

Blastoise used Tackle (Effective):
Blastoise's PP decreased by 10, Pikachu's HP decreased by 30.

Pikachu used Slam (Effective):
Pikachu's PP decreased by 20, Blastoise's HP decreased by 60.

Blastoise used Tackle (Effective):
Blastoise's PP decreased by 10, Pikachu's HP decreased by 30.

Pikachu used Slam (Effective):
Pikachu's PP decreased by 20, Blastoise's HP decreased by 60.

```
./skeleton part2 blastoise
Winning path with probability: 2.71267e-05
```

Beginning state: No action Yet
Level : 0 Pikachu HP: 200 PP: 100, Blastoise HP: 200 PP: 100

Pikachu used Thundershock : effective
Level : 1 Pikachu HP: 200 PP: 90, Blastoise HP: 160 PP: 100

Blastoise used Tackle : effective
Level : 2 Pikachu HP: 170 PP: 90, Blastoise HP: 160 PP: 90

Pikachu used Thundershock : effective
Level : 3 Pikachu HP: 170 PP: 80, Blastoise HP: 120 PP: 90

Blastoise used Bite : effective
Level : 4 Pikachu HP: 110 PP: 80, Blastoise HP: 120 PP: 65

Pikachu used Thundershock : effective
Level : 5 Pikachu HP: 110 PP: 70, Blastoise HP: 80 PP: 65

Blastoise used Bite : effective
Level : 6 Pikachu HP: 50 PP: 70, Blastoise HP: 80 PP: 40

Pikachu used Thundershock : effective
Level : 7 Pikachu HP: 50 PP: 60, Blastoise HP: 40 PP: 40

Blastoise used Bite : effective
Level : 8 Pikachu HP: -10 PP: 60, Blastoise HP: 40 PP: 15

Final state at Level 8: Pikachu HP: -10, PP: 60; Blastoise HP: 40, PP: 15

The earliest path winning node for Blastoise participant is on the level – 8. It meets the action requirements. Part2 is not only finding the earliest winning node, it is a test to see all my functions work correctly. I can see that I could implement my expandGraph function very well so that I could acces all attribute values printing the path.

Pikachu used Thundershock (Effective):
Pikachu's PP decreased by 10, Blastoise's HP decreased by 40.

Blastoise used Tackle (Effective):
Blastoise's PP decreased by 10, Pikachu's HP decreased by 30.

Pikachu used Thundershock (Effective):
Pikachu's PP decreased by 10, Blastoise's HP decreased by 40.

Blastoise used Bite (Effective):
Blastoise's PP decreased by 25, Pikachu's HP decreased by 60.

Pikachu used Thundershock (Effective):
Pikachu's PP decreased by 10, Blastoise's HP decreased by 40.

Blastoise used Bite (Effective):
Blastoise's PP decreased by 25, Pikachu's HP decreased by 60.

Pikachu used Thundershock (Effective):
Pikachu's PP decreased by 10, Blastoise's HP decreased by 40.

Blastoise used Bite (Effective):
Blastoise's PP decreased by 25, Pikachu's HP decreased by 60.

Conclusion

In this project, I successfully implemented a simulation of a Pokémon battle between Pikachu and Blastoise, adhering to the specifications provided in the assignment. By simplifying the Pokémon fighting procedures, I focused on two main attributes for each Pokémon: Health Points (HP) and Power Points (PP). The simulation involved creating a graph to represent all possible game states and their transitions based on the Pokémon's attacks.

The project required the use of a doubly linked list to manage the nodes representing game states, ensuring efficient traversal and expansion of the graph. I carefully implemented the expandGraph function to generate new game states based on the available attacks, their accuracy, and their PP costs. Each attack's effect on the opponent's HP and the user's PP was accurately calculated and updated.

For the final output, the simulation was able to determine the shortest path to a winning state for both Pokémon. The results were validated by printing the sequence of actions and their effects, confirming the accuracy of the simulation. The implementation adhered to the

constraints of not using STL and ensuring the code was compatible with the default g++ compiler on Ubuntu OS.

Overall, the project demonstrated the effective application of data structures and graph theory, probability in simulating a turn-based game scenario. The detailed analysis of each action's impact on the game state provided a clear understanding of the mechanics involved, and the final results showed the successful completion of the simulation as per the assignment's requirements.

Summary of Key Points:

The DoublyList is effectively used as a queue to manage nodes for BFS traversal.

The expand function iterates through nodes, expanding each node and generating its children.

The expandGraph function creates new game state nodes and adds them to the DoublyList.

This approach ensures a complete exploration of all possible game states up to the specified depth, with nodes being processed level by level.