BLG 335E - Analysis of Algorithms II
Homework-1 Report

Date: March 24th, 2020
Ece Çınar - 150150138

1. Classes & Methods

- State Class

State class represents a node of the tree. This class holds:

- Health and power points of both Pokémons
- Name of the attack that was carried out to create the current state
- A variable which shows if the attack was successful or not
- Turn value
- Probability of the current state to occur during the battle
- A vector that holds pointers to the current state's children
- A pointer to the parent state
- The level at which the state occurs

I did not use a variable to indicate if a state(node) is leaf or not since I was able to control everything without that variable. I used BFS approach to find the shortest path in part3 and for the very reason I had to add two different variables to my state representation. These two variables hold information about the preceding attack.

- Attack Class

Each attack is represented by the attack class. This class holds:

- Name of the attack
- Amount of power points used to carry out the attack
- Accuracy of the attack
- Damage of the attack
- First usage of the attack in terms of level

- Battle Class

Battle class is a major class that holds many attributes and methods. State calculation, graph(tree) creation, breadth first tree traversal, depth first tree traversal, shortest path search and many other core operations are carried out in this class.

Attributes

- A vector containing all possible Pikachu attacks
- A vector containing all possible Blastoise attacks
- A state pointer that points to the root of the tree
- An integer which holds information of the part that the program is being executed for
- An integer showing the maximum level
- A char to indicate which Pokémon has to win the battle (for part3)

Methods

1. **Battle()**

   Constructor of Battle class takes four arguments which are health and power points of Pikachu and Blastoise respectively. Then it allocates memory for the root State pointer of battle. Attributes of the root are assigned by the constructor.

2. **setPikaAttack()**

   This method takes reference to an Attack object as it's argument. This Attack object is then pushed into pikaAttacks vector which consists of all Pikachu attacks.

3. **setBlastAttack()**

   This method takes reference to an Attack object as it's argument. This Attack object is then pushed into blastAttacks vector which consists of all Blastoise attacks.

4. **initGraph()**

   This method takes two arguments which represent maximum level and current part respectively. These values are assigned to class attributes maxLevel and part. After that, createGraph function is called to create the tree.

5. **createGraph()**

   This method does not work iteratively, it works recursively. It takes an argument of type State pointer and depending on the turn, attackBlast or attackPika methods are called in order to create and calculate attributes of current state's children. Then createGraph method is called again for each child state using pointers to them. This method terminates if maximum level limit is reached or one of the competitors is knocked down.

6. **attackBlast()**

   This method is called if it is Pikachu's turn to attack. It takes an argument of type state pointer. First of all, number of attacks that can be carried out by Pikachu at the current level is calculated. An attack can be carried out only if Pikachu's power point is enough to perform that attack and if the first level that the attack can be used is lower than or equal to the current level. Taking these constraints into consideration, all possible output states are calculated.

If accuracy of an attack is 100%, the attack will be successful and only one output state will be produced. Otherwise, two different output states will be produced for the attack. One of the states is represented with success pointer and the other is with failure pointer.

Health of a Pokémon cannot be lower than zero. If damage caused by the attack is higher than Blastoise's health, health value is set to zero.

First, output attributes of a successful attack are calculated and assigned to the new state pointed by success pointer. Since failure and success situations have a lot in common, it is easier to equalize pointer contents and perform necessary adjustments.

After performing all necessary calculations, pointer/s is/are pushed into the children vector of the state. If this method is being executed for part 1, print function is called to print the last level of the tree.

7. **attackPika()**

   This method is similar to attackBlast. The only difference is that Blastoise is attacking Pikachu in this scenario.

8. **print()**

   This method takes a state pointer argument and prints out content of the state pointed by this pointer.

9. **traverseBFS()**

   This method traverses the tree created previously using breadth-first search approach. While traversing the tree, it also calculates total number of the nodes. A queue is used to determine the order of traversal. Queues work with FIFO principle and this is why breadth-first search can be performed using a queue. First, root pointer is pushed into the queue. Until the queue is emptied, a pointer is popped from the queue and counter is increased by one. Children of each popped pointer are pushed into the queue. When the queue is emptied, function prints out the total number of nodes and the execution time in clock ticks.

10. **traverseDFS()**

    This method traverses the tree created previously using depth-first search approach. While traversing the tree, it also calculates total number of the nodes. A stack is used to determine the order of traversal. Stacks work with LIFO principle and this is why depth-first search can be performed using a stack. First, root pointer is pushed onto the stack. Until the stack is emptied, a pointer is popped from the stack and counter is increased by one. Children of each popped pointer are pushed onto the stack. When the queue is emptied, function prints out the total number of nodes and the execution time in clock ticks.

11. findFinishNode()

This method performs breadth-first search in order to find the last state of the easiest action sequence to end the battle. In order to find the shortest sequence of Pikachu's victory, we are searching for the first state with Blastoise's health equal to zero, and for Blastoise vice versa. Level order traversal guarantees that the first node encountered meeting the constraints will be the one that we are looking for.

In order to prevent redundant expansion of states, a set is used. This set keeps content of unique sub-states encountered until then. Since keeping the states as they are will require major memory usage, strings can be used instead of state pointers.

Sub-states consist of specifically chosen state attributes. These are health and power points of Pikachu and Blastoise, and also turn variable. Combining these five attributes into a string creates a sub-state. This approach is not only space efficient but also time efficient.

Actually, this method is similar to traverseBFS method. However, instead of traversing a tree it builds one partially. Depending on the turn, attackBlast or attackPika methods are called to create children of the current state. Children are pushed into the queue only if their sub-state is not in the set. The unique sub-state is then inserted into the set. Method returns a pointer to the current state if health point of the enemy is zero.

12. findShortestPath()

This method takes two arguments which represent mode and current part respectively. First of all, findFinishNode method is called to find the last node of shortest path and it is assigned to the traverse pointer. Until the root state is reached, traverse pointer moves upwards following parent of each state and the encountered states are pushed into a stack. This stack holds the shortest path. Until the stack is emptied, each state is popped from the stack and relevant information is printed out.

- FileOps Class

FileOps is a class that handles file operations. I will not delve into details like I did with Battle class. Attributes of the class are a file pointer, two strings to be used at file read operations, a string to keep the file name and an instance of Attack class. I will briefly explain the methods of Battle Class below.

Methods

1. FileOps()

Constructor of FileOps class takes a file name argument and opens that file. It also assigns that file name to filename attribute of class.

2. **isOpen()**

   This method checks if a file is successfully opened or not. Returns a Boolean.

3. **readFile()**

   This method takes two arguments which are reference to a Battle object and the mode of file. First line of the file is ignored. Each line of the file is tokenized using comma as a separator. Then, these tokens are assigned to attack attributes respectively. After the assignment process, each attack is pushed into the vector it belongs to. If the mode is 'P', attacks are pushed into pikaAttacks vector, otherwise attacks are pushed into blastAttacks vector.

4. **~ FileOps()**

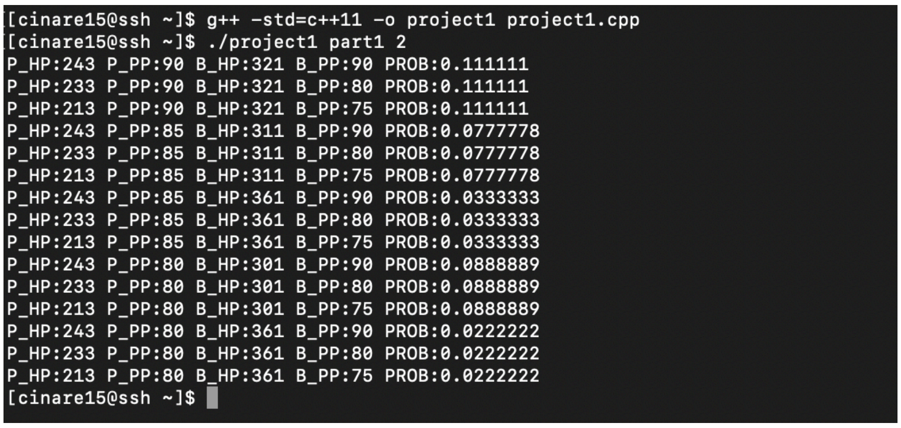
   This method closes the file.

2. Sample Outputs & Analysis

Outputs of part1, part2 and part3 are calculated for:

- Pikachu HP = 273
- Pikachu PP = 100
- Blastoise HP = 361
- Blastoise PP = 100

- Part1



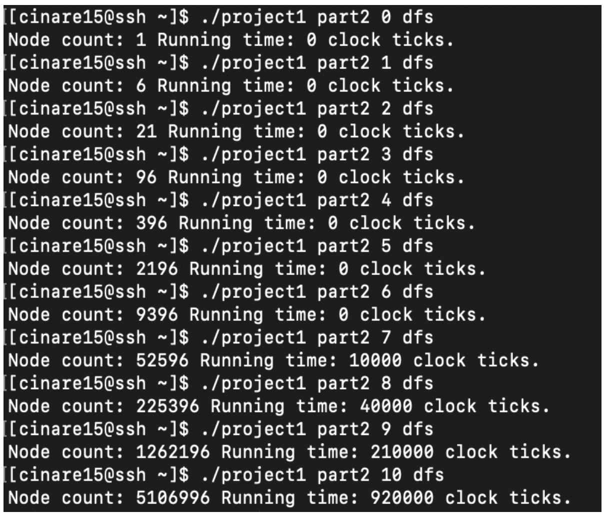
```
[cinare15@ssh ~]$ g++ -std=c++11 -o project1 project1.cpp
[cinare15@ssh ~]$ ./project1 part1 2
P_HP:243 P_PP:90 B_HP:321 B_PP:90 PROB:0.111111
P_HP:233 P_PP:90 B_HP:321 B_PP:80 PROB:0.111111
P_HP:213 P_PP:90 B_HP:321 B_PP:75 PROB:0.111111
P_HP:243 P_PP:85 B_HP:311 B_PP:90 PROB:0.0777778
P_HP:233 P_PP:85 B_HP:311 B_PP:80 PROB:0.0777778
P_HP:213 P_PP:85 B_HP:311 B_PP:75 PROB:0.0777778
P_HP:243 P_PP:85 B_HP:361 B_PP:90 PROB:0.0333333
P_HP:233 P_PP:85 B_HP:361 B_PP:80 PROB:0.0333333
P_HP:213 P_PP:85 B_HP:361 B_PP:75 PROB:0.0333333
P_HP:243 P_PP:80 B_HP:301 B_PP:90 PROB:0.0888889
P_HP:233 P_PP:80 B_HP:301 B_PP:80 PROB:0.0888889
P_HP:213 P_PP:80 B_HP:301 B_PP:75 PROB:0.0888889
P_HP:243 P_PP:80 B_HP:361 B_PP:90 PROB:0.0222222
P_HP:233 P_PP:80 B_HP:361 B_PP:80 PROB:0.0222222
P_HP:213 P_PP:80 B_HP:361 B_PP:75 PROB:0.0222222
[cinare15@ssh ~]$
```

Figure 1 - Information of the 2nd layer

- Part2

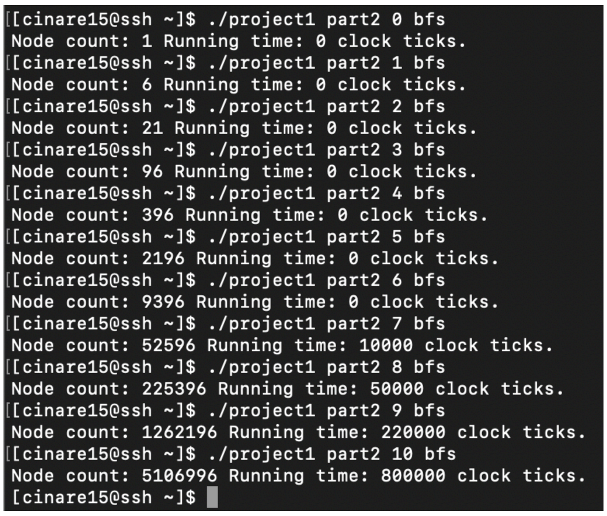
Time complexity of Breadth-First Search and Depth-First Search algorithms are the same and equal to O (V + E). V represents number of vertices and E represents number of edges. Vertices correspond to nodes and edges correspond to attacks in our case. The only difference between

BFS and DFS implementations is the container used to determine the order of elements. Queue is used for BFS implementation while stack is used for DFS. Other than that, they both visit each node once during execution and have the same time complexity. Analyzing the outputs (Figure 2 and Figure 3), we can easily see that their running times are so similar. Since node count increases rapidly, running time of algorithms also increase rapidly.



```
[cinare15@ssh ~]$ ./project1 part2 0 dfs
Node count: 1 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 1 dfs
Node count: 6 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 2 dfs
Node count: 21 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 3 dfs
Node count: 96 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 4 dfs
Node count: 396 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 5 dfs
Node count: 2196 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 6 dfs
Node count: 9396 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 7 dfs
Node count: 52596 Running time: 10000 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 8 dfs
Node count: 225396 Running time: 40000 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 9 dfs
Node count: 1262196 Running time: 210000 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 10 dfs
Node count: 5106996 Running time: 920000 clock ticks.
```

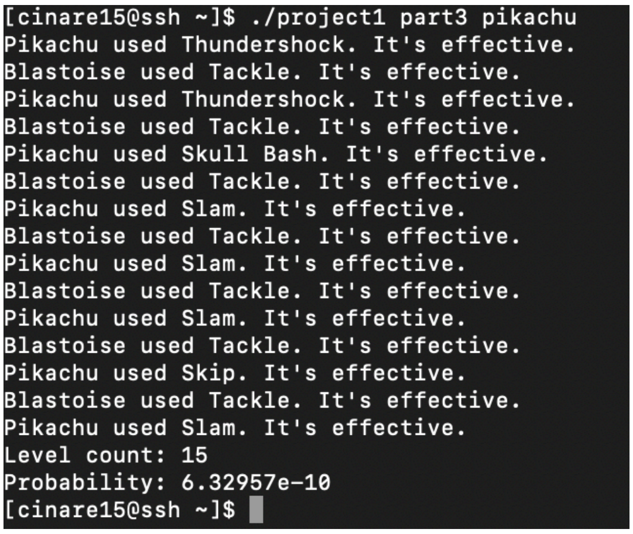
Figure 2 – Node counts of level 0 to 10 using depth-first search approach



```
[cinare15@ssh ~]$ ./project1 part2 0 bfs
Node count: 1 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 1 bfs
Node count: 6 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 2 bfs
Node count: 21 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 3 bfs
Node count: 96 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 4 bfs
Node count: 396 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 5 bfs
Node count: 2196 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 6 bfs
Node count: 9396 Running time: 0 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 7 bfs
Node count: 52596 Running time: 10000 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 8 bfs
Node count: 225396 Running time: 50000 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 9 bfs
Node count: 1262196 Running time: 220000 clock ticks.
[cinare15@ssh ~]$ ./project1 part2 10 bfs
Node count: 5106996 Running time: 800000 clock ticks.
[cinare15@ssh ~]$
```

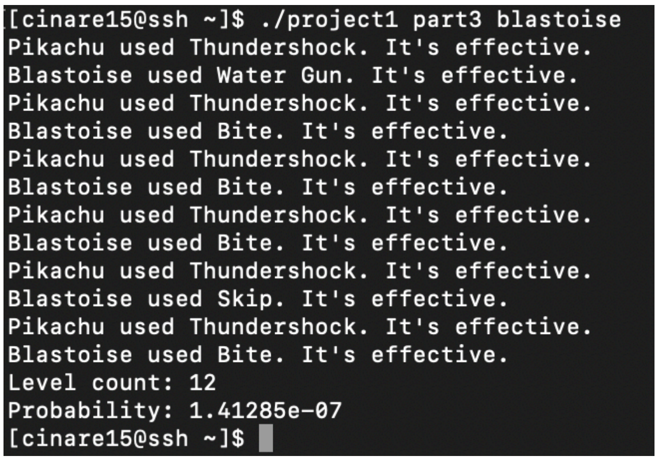
Figure 3 – Node counts of level 0 to 10 using breadth-first search approach

- Part3



```
[cinare15@ssh ~]$ ./project1 part3 pikachu
Pikachu used Thundershock. It's effective.
Blastoise used Tackle. It's effective.
Pikachu used Thundershock. It's effective.
Blastoise used Tackle. It's effective.
Pikachu used Skull Bash. It's effective.
Blastoise used Tackle. It's effective.
Pikachu used Slam. It's effective.
Blastoise used Tackle. It's effective.
Pikachu used Slam. It's effective.
Blastoise used Tackle. It's effective.
Pikachu used Slam. It's effective.
Blastoise used Tackle. It's effective.
Pikachu used Skip. It's effective.
Blastoise used Tackle. It's effective.
Pikachu used Slam. It's effective.
Level count: 15
Probability: 6.32957e-10
[cinare15@ssh ~]$
```

Figure 4 – Easiest action sequence for Pikachu to win.



```
[cinare15@ssh ~]$ ./project1 part3 blastoise
Pikachu used Thundershock. It's effective.
Blastoise used Water Gun. It's effective.
Pikachu used Thundershock. It's effective.
Blastoise used Bite. It's effective.
Pikachu used Thundershock. It's effective.
Blastoise used Bite. It's effective.
Pikachu used Thundershock. It's effective.
Blastoise used Bite. It's effective.
Pikachu used Thundershock. It's effective.
Blastoise used Skip. It's effective.
Pikachu used Thundershock. It's effective.
Blastoise used Bite. It's effective.
Level count: 12
Probability: 1.41285e-07
[cinare15@ssh ~]$
```

Figure 4 – Easiest action sequence for Blastoise to win.