Name Surname: Hüseyin Güney İçel

Student Number: 820210302

CRN: 25000

Project 1 Report

The goal is to find a winning sequence of actions and objects that lead to victory within the game. This report provides an analysis of the code structure, its functionality, and potential areas for improvement.

The code consists of the following components:

- Data: A structure to store information about actions, objects, and their associated returns.
- Stack: A structure to manage a stack of Data objects.
- push(): A method to add new entries to the stack.
- print(): A method to print the contents of the stack.
- main(): The main function that includes the loop to find winning inputs in game.

Main Loop:

The main loop continues until a winning sequence is found or the game ends. It iterates through actions and objects, advancing the game state accordingly. The main idea is trying every action with every possible object.

Areas for Improvement:

1. Iteration logic:

```
else if(effectID == -1)//We died (in-game) so we skip that action
{
    if(actions >= 5)
    {
        objects++;
        actions = 1;
    }
    else
        actions++;
}
else if(effectID == 0)//If this action does not change anything, then we skip also
{
    if(actions >= 5)
    {
        objects++;
        actions = 1;
    }
    else
        actions++;
}
```

As you can see, code increases action by 1 after each iteration and sets action to 0 and increases object if action was 5. This is for searching each possible outcome. This logic does not care about prior action and object couples so maybe a different approach could result in a more optimized code

2. “EffectID != 0 & EffectID != -1” Situation:

```
else if(effectID != 0 && effectID != -1)//Thi
our stack.
{
    mystack.push(actions, objects, effectID);
    actions = 1;
    objects = 0;
}
```

In this situation, action and object gets resetted. This is for the purpose of searching each edge cases but I am not sure if this approach is wise enough.

3. Condition blocks for loops:

```
if(mystack.step > 2 && mystack.data[mystack.step - 1].returns == mystack.data[mystack.step - 3].returns)//This conditional
block is for prevent the loop between rooms. Because going between rooms produce positive return value, we can get stuck
between rooms always going one from another... So if there is a "returnA, returnB, returnA" kind of values in our stack we skip
an action after another returnB...
{
    mystack.data[mystack.step - 1].returns = -1;
    if(mystack.data[mystack.step - 2].action >= 5)
    {
        objects = 0;
        actions = 1;
    }
    else
    {
        actions =  mystack.data[mystack.step - 2].action + 1;
        objects = mystack.data[mystack.step - 2].object;
    }
}
```

This conditional code blocks prevents the algorithm to go into loop between toilet and cell (1, 4 and 1, 3) after Guardian (Near Toilet) is killed. Basically it detects if there is a “A, B, A” return sequance and forces itself out of loop by continueing after B.

```
else if(first_state.room_id == 2 & actions == 1)//This condition only prevents us from exiting from the last room before doing
anything purposeful...
{
    actions += 1;
}
```

And this code block is purely for the last room of the game. It forces algorithm to take other actions different from exiting the room first, so player does not exit from the last room immediately and gets stuck in the game.

These solutions for the “loop situations” probably could be solved more wisely.

Conclusion:
In conclusion, the provided code effectively implements a strategy for navigating a game environment using a stack data structure. It demonstrates a clear understanding of stack operations and game state management. However, there are opportunities for improvement in terms of code readability, optimization, and error handling. Overall, the code provides a solid winnig pattern in the end altough due to unoptimized loop detection methods, there are unnecessary steps at the last part.