Automata and Logic Project II -- Reachability Solver

In this project the goal is to implement a reachability game solver that obeys the optimal complexity *O(|V| + |E|)*:

Programming Language:

- Python

Prerequisites:

- **Linux** system *(standalone application build for Linux only)*

Structure of the graph text-file:

For the input graph you can create your own by following these instructions:

- create an empty .txt file with any filename you want
- input lines as followed:

  > nodeID nodeType successor(s)

  where

  ```
  ├── nodeID        = any unique integer number representing the node
  ├── nodeTyoe      = 0 if the node is green/diamond and 1 if the node is
                      red/square
  └── successor(s)  = list of successor nodes for the given nodeID, list
                      must not contain any space and must get separated
                      with commas
  ```

- example:

  > 0 0 1,2
  > 1 1 2,3,4
  > ...

Running the Reachability Solver:

Arguments:

- The program always calculates the winning regions and winning strategy for player 1 (= player green)
- for the first argument you have to enter a list of target nodes Z (e.g. 1,2)
  > ***NOTE***: use comma as delimiter and avoid space between the nodes
- for the second argument you have to enter the path of the textfile containing the graph structure
  > (***NOTE***: no path is needed if the file is in the same folder as the program itself)
- Example:
  > ./PENZ_project_II 1,2 G.txt

Is it really the optimal complexity?

CODE-snippet:

```
while queue:
    n = queue.popleft()

    for np in g.getPredecessors(n):
        # for every predecessor in the nodes in queue the algorithm checks if
        # this region has already been checked
        # and if it is a green or red node

        # if green:
        # append the predecessors to the queue and to the winRegion
        # set the visited position with player green and add a transition from
        # predecessor to node in the winning strategy
        #
        # if red:
        # decrease kv until kv = 0
        # then add this node to winning region according to the algorithm, set
        # visited to player green and add the successor node to the queue

        if visited[np] == -1:
            if g.getNodePlayerID(np) == playerGreen:
                queue.append(np)
                visited[np] = playerGreen
                winRegion.append(np)
                winStrat[np] = n

            elif g.getNodePlayerID(np) == playerRed:
                nodeOut[np] -= 1
                if nodeOut[np] == 0:
                    queue.append(np)
```

```
                visited[np] = playerGreen
                winRegion.append(np)
```

Explanation:

The **queue** variable iteratively stores all the nodes inside the graph and for every node the algorithm checks for its predecessors and adds these to the winning region iff this predecessor node is a green/diamond vertex.

In my opinion **while queue:** represents the complexity *O(|V|)* and since we check for all predecessor of a single node in queue this would be *O(|E|)*.

Therefore, the total complexity would result in ***O(|V| + |E|)***

Important Notes during the execution:

The program will export a ***export.dot*** and ***export.png*** file representing the graph you choose to use as argument.