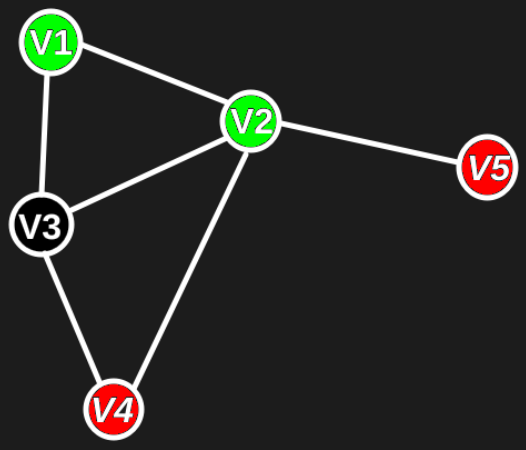
Throughout this model we will be using an adjacency matrix to represent our graph. We are familiar with adjacency matrices from our data structure course. Here is an adjacency matrix M for the below graph:



$$M = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

Since the graph itself does not change throughout the search, thus we will represent the graph in an immutable $n \times n$ adjacency matrix M where n is the number of vertices of our graph G.
Since we are using an adjacency list, this means that we are arbitrary numbering the vertices of our graph from 1 to n. For example: $\{v_1, v_2, \ldots, v_n\}$.

State

Each state of the game is a row $1 \times n$ vector whose elements are from $\{0, 1\}$ where on the ith element of that vector, a 0 represents a red color or a black color vertex, 1 represents a green color vertex. For example for the above graph our state vector can be represented as follows:

$$\text{config} = c = \{1, 1, 0, 0, 0\}$$

We will also use another $1 \times n$ row vector called *Blacks Position* (BP) that is a row of 1 s except for the indices that represent a black vertex in our current state. For the above graph and its corresponding state this vector is as follows:

$$bp = \{1, 1, 0, 1, 1\}$$

This means that v_1 is green, v_2 is green, v_3 is black, v_4 is red and v_5 is red
Each moment of the game is represented as a configuration which gives the player the exact information about the state of the entire game graph.

```
class State:
    # This is the single immutable adjacency list, shared by all the state objects
    status constant int M[n][n]

    # The configuration vector that determines which of the vertices
    # are (green) or (red or black) in this state
    int c[n]

    # The vector that determines which of the vertices are black
    int bp[n]
```

State Space

The set of all possible $1 \times n$ vector pairs $(c[n],\ bp[n])$ over $\{0, 1\}$ constitutes our state space.

Initial State

The initial state of the game can be any $1 \times n$ vector pair $(c[n],\ bp[n])$ over $\{0, 1\}$ from our state space. An example of an initial state was given earlier in the document.

Goal Test Function

The goal test function is straight forward; If all the elements of our configuration vector are 1s then we have reached a state where every node is green.

```
For each v in CurrentState:
 if v != 0:
     return false
 return true
```

Successor Function

Clicking on Red or Green Nodes

Clicking on a red or green vertex causes a color toggle from red to green on the clicked vertex and all it's red and green vertices.
This state change can easily be represented as a sum; suppose in our example we clicked on vertex V_2, then we can calculate the next state as follows:

$$\text{currentState}.c \leftarrow (\text{currentState}.c + \text{currentState}.M_i)\ \ \&\ \ \text{currentState}.bp$$

where M_i is the ith row of our adjacency list. The sum in the above assignment causes all the colors adjacent to V_2 and V_2 itself to toggle from red to green and from green to red.
The binary & operation, makes sure that the black nodes don't change their value and stay 0.
Here is an example of what happens when we click on V_2 in our example graph:

$$c \leftarrow (\{1,1,0,0,0\} + \{1,1,1,1,1\}) \ \& \ \{1,1,0,1,1\}$$

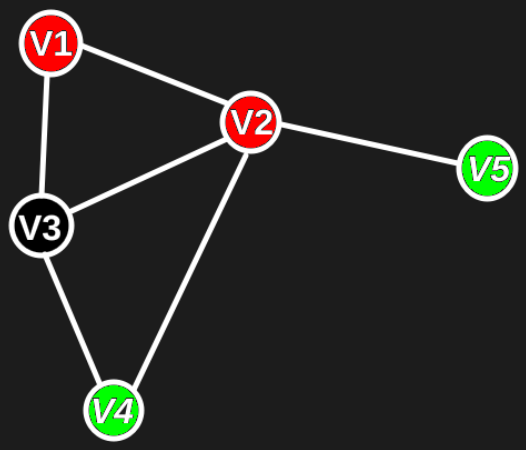
Which results in:

$$c = \{0,0,0,1,1\}$$

and the bp vector stays the same (we didn't click on a black node)

$$bp = \{1,1,0,1,1\}.$$

that corresponds to the following graph:



Clicking on Black Nodes

Clicking on a black node causes a color toggle from red to green on all the nodes adjacent red and green nodes. (This behaviour is the same as clicking on a red or green node)

$$\text{currentState}.c \leftarrow (\text{currentState}.c + \text{currentState}.M_i) \ \& \ \text{currentState}.bp$$

After the toggle, if the majority of the adjacent nodes are red, we would simply flip the entry corresponding to this black node in the bp vector from 0 to 1, otherwise if the majority of the adjacent nodes are green we would also change the element corresponding to this black node in the configuration vector from 0 to 1 (making the node green), otherwise, we make no further changes to the state.
This concept is actually quiet easy to represent just by using simple binary operations on our state.
To count the number of non-black neighbors, we can simply use the following equation:

$$neighborCount = sum(st.M[i] \& st.bp)$$

The *sum* operator simply sums all the elements of a vector given to it. For the above example the number of non-black neighbors of node V_3 is:

$$neighborCount = sum(\{1,1,1,1,0\} \ \& \ \{1,1,0,1,1\}) = 3$$

To count the number of green neighbors, we can simply use the following equation:

$$greenNeighborCount = sum(st.c\&(st.M[i]\&st.bp))$$

For our example:

$$greenNeighborCount = sum(\{0,0,0,1,1\} \ \& \ (\{1,1,1,1,0\} \ \& \ \{1,1,0,1,1\})) = 1$$

Now we do a simple comparison: if the number of green neighbors is more than half the amount of the total non-black members, then we turn this node to a green node, else, if the number of green neighbors is less than half the amount of the total non-black members, we turn this node into a red node, otherwise, this node remains black.
The pseudocode for all these operations is available in the next page.

Pseudocode for the successor function

```
successor(state st, action ac) {
    state nextState
    nextState.bp = = st.bp

    if ac is clicking on a green or red node v with index i:
        nextState.c = (st.c + st.M[i]) & st.bp
    else if ac is clicking on a black node v with index i:
        nextState.c = (st.c + st.M[i]) & st.bp

        # count the number of non-black neighbors
        neighborCount = sum( st.M[i] & st.bp )

        # count the number of green neighbors
        greenNeighborCount = sum( st.c & ( st.M[i] & st.bp ) )

        if greenNeighborCount > (neighborCount / 2):
            # turn black node to green
            nextState.bp[i] = 1
            nextState.c[i] = 1

        else if greenNeighborCount > (neighborCount / 2):
            # turn black node to red
            nextState.bp[i] = 1
            # black nodes are already 0, this is just to increase readability
            nextState.c[i] = 0

        else:
           # there are equal number of red and green nodes around this black node
           # don't do anything

    return nextState
}
```

Cost

Each click on a vertex costs 1 unit.
We are looking for a minimum number of clicks to turn all the nodes of the given graph to green.