Sprouts

Group 8 - 17

Bernard Kwee (Leader)
Koh Shao Bing
Sean Tan
4S1

Written Report

1 Introduction

1.1 Terminology

Term	Definition
Vertex (Plural: vertices)	A node
Edge	An edge connects vertices. 1 edge starts from a vertex and ends at a vertex. (The start and end vertices can be the same vertex)
Degree of a vertex	The number of points of connection due to an edge of a certain vertex. When a vertex has a point of connection to an edge, the degree of the vertex will increase by 1.
Self-loop	An edge that starts and ends at the same vertex

1.2 What is *Sprouts*?

Sprouts is a 2 player game. The game starts with V vertices. Each vertex does not have any edges connected to it. On each player's turn, they must draw 1 edge (including a self-loop) connecting at most 2 vertices. Failure to do so will result in that player losing.

The following constraints are imposed when each player is drawing an edge:

(a) The new edge cannot intersect other drawn edges.

(b) After the drawing of an edge, degree of every vertex must be at most 3.

(c) After every valid move, a new vertex is generated at the midpoint of the edge. The new vertex will be connected to the edge and will hence have a starting degree of 2.

(d) A draw is reached if it is determined that the game can continue indefinitely.

Appendix A will include visualisations of the rules of the game.

2 Literature Review

Sprouts is a game that was introduced in 1960. Up till today, not much research has been conducted on this topic. Most of the research conducted on this topic is focused mainly on the winning strategies of the game. There are mainly 2 types of research done on this topic, non-computational and computational.

For non-computational research, researchers have so far determined the winning strategy for each player up till when $V = 7$. *Focardi and Luccio (2003)* has determined using modular arithmetic that when $V = 3$, 4 and 5, player 1 will have a winning strategy; while when $V = 1$, 2, 6 and 7, player 2 will have a winning strategy. Other researchers who have conducted

research has results similar to that of *Focardi and Luccio (2003)*. It was proposed that when $V \equiv 3, 4$ or 5 (mod 6), player 1 has a winning strategy; whereas when $V \equiv 0, 1$ or 2 (mod 6), player 2 has a winning strategy.

For computational research, researchers have created computer programmes to calculate and determine the winning strategy for each player. *Applegate et. al. (1991)* has computed the winning strategy for the highest $V = 11$. They have proposed the *Sprouts Conjecture* as stated in the above paragraph. As technology advances, more computing power has allowed *Lemoine and Viennot (2015)* to calculate the winning strategy for each player, the highest being $V = 47$. However, one limitation in their results is that they were only able to do so for certain values of V, but not for all values of V up to $V = 47$. For example, they were unable to compute the winning strategy for each respective player from $V = 42$ to $V = 46$.

Furthermore, computing poses a problem: there is a need to verify the correctness of the strategy provided by the programme. The programme may be prone to mistakes in which the programmer nor other researchers may not recognise. Additionally, the algorithm *Lemoine and Viennot (2015)* used is mainly based on the algorithm of "brute-force", where the programme generates all the possibilities of how the game would end, rendering it inefficient. Although the programme attempted to draw an equivalence between the different cases generated, hence leading to a significant cut in computation time, it may lead to false equivalencies and hence wrong results since it ignored minor differences between different cases (for example, the manner of how the edges are connected to certain vertices). It is also not feasible to check all the cases generated by the programme manually given the exponentially increasing number of cases as V increases, since it will be time-consuming and not practical to do so. Another limitation in this project is that computation is determined by the use of supercomputers, which makes it expensive and inaccessible to compute results for large values of V.

Up till date, there exists only a non-polynomial time complexity solution and the problem is classified as NP-hard. Ultimately, researchers must find ingenious ways to cut down on the number of cases computers need to compute, while ensuring the correctness of the code and also the efficiency of the programme.

3 Objectives and Research Question

This project aims to achieve the following objectives:

- To determine the winning strategy for each player for $V \leq 5$.
- To determine the winning strategy for each player when the rules of the game is modified.
- To determine the winning strategy for each player by computing.

Hence, 3 research questions were proposed:

1) What is the winning strategy for each player when $V \leq 5$?
2) When the maximum degree for any vertex is allowed to be D where $D \neq 3$, what is the winning strategy for each player?
3) Is there a more efficient way to determine the winning strategy of each player?

4 Results

4.1 Research Question 1: Which player has a winning strategy when $V \leq 5$?

Note that visualisations of the proof is at Appendix B.

It is obvious when $V = 1$, there is only 1 way the game can proceed. Player 2 will win.

For $V = 2$, the game can proceed in 2 ways as shown at Appendix B. Regardless of how the game proceeds, player 2 will win.

For $V = 3$, 4 and 5, there are multiple ways in which the game can proceed. Considering optimal play from the 2 players, player 1 has a winning strategy. The visual proof is shown at Appendix B. However, a novel way has been used to solve this problem as compared to other researchers, such as *Focardi and Luccio (2003)*, who employ the use of "brute-force" to solve the problem - that is, they considered all cases in which the game can end.

2 new ways to solve this problem has been proposed. These new ways also give strategies into how to win the game.

First, since we already know who wins when $V = 1$ and 2 (that is player 2 wins), it is not necessary to continue the game when a smaller game of $V = 1$ or 2 is played, since we are able to determine which player is on a certain winning position once the game of $V = 1$ or 2 is touched. Since player 2 wins a game of $V = 1$ or 2, this means that the first player that starts a subgame of $V = 1$ or 2 will be at a losing position for a subgame of $V = 1$ or 2.

Since the amount of "brute-force" can be cut by knowing the results of smaller subgames of $V = 1$ or 2, it gives players a new winning strategy. The new winning strategy is as follows: Since the first player who starts a subgame of $V = 1$ or 2 will definitely lose, a player

can force the other player to enter a subgame of $V = 1$ or 2, in which the player knows he would win that subgame. Similarly, if a player can skillfully create a subgame of $V = 3$, 4 or 5 and starts first, the player would win that subgame.

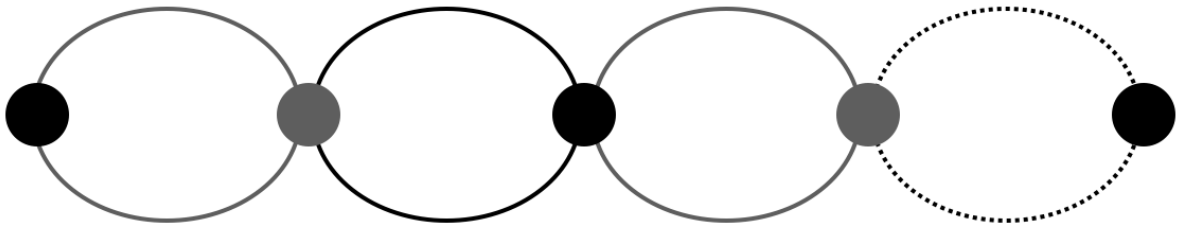
In order to force other players to play a subgame of smaller V, it is necessary to isolate V vertices such that edges from other vertices in the full game cannot be connected to the V vertices, and the V vertices have any edges drawn on them. In this way, it effectively ensures that the subgame of V vertices can be played, ensuring the correctness of the proof. Hence, players would need to effectively isolate V vertices such that when a game of V vertices is played, they would be able to win that subgame. For example, when a game of $V = 5$ is played, a player can consider isolating 2 vertices and force the other player to start on the subgame $V_{sub} = 2$, as such the player would win the subgame of $V_{sub} = 2$.

4.2 When the maximum degree for any vertex is allowed to be D where $D \neq 3$, what is the winning strategy for each player?

There are 2 cases: $D = 2$ and $D \geq 4$.

(i) For $D \geq 4$,

When a new edge is drawn, by the rules of the game, the midpoint of the edge is generated as a new vertex, in which edge(s) can be drawn if permitted. With the new midpoint, a new self-loop can be drawn. Similarly, on the new self-loop, the midpoint of the self-loop will be marked out, in which the next player can mark out also. A self-loop can be drawn on the midpoint of any new edge, because the midpoint vertex has a degree of 2 (before the addition of a self-loop). After a self-loop is drawn, the midpoint vertex has a degree of 4, which is allowed since $D \geq 4$.



The above figure shows how the game can proceed. The vertex on the left (marked in black) is the starting vertex. A player can draw a self-loop on the vertex, or draw another edge to another vertex. The existence of the ability to draw the first edge is justified since no matter the value of V, the first player can always draw an edge. The midpoint of the edge is marked as gray. With the midpoint, a new self-loop can be drawn, in which the midpoint of the new

self-loop is marked in black. This process can repeat itself indefinitely. Hence, the game ends in a draw.

(ii) For $D = 2$,

When a new edge is drawn and the midpoint of the edge is generated, that midpoint would already have a degree of 2, which is the maximum degree. Hence no other edge can be connected to the generated midpoint and we can omit discussing and drawing the midpoint for the proof.

The proof is done through strong induction, with the induction hypothesis that when the initial number of vertices, V = odd, player 1 wins, whereas when V = even, player 2 wins. The base cases of $V = 1$ and 2 can easily be manually computed and supports the hypothesis. (Refer to Diagrams 1 and 2 in Appendix C)

Consider $V = 2k + 1$, player 1 simply performs a self-loop on any vertex, ensuring that no other vertices are encircled within. (Refer to Diagram 3 in Appendix C) It is plain that this has become a $V = 2k$ game in which player 1 makes the second move. By the induction hypothesis, player 1 will win.

Consider $V = 2k$. There are 2 possible moves by player 1, one that connects 2 distinct vertices, or a self-loop.

Case 1: If 2 distinct vertices are connected, player 2 simply connects the same 2 vertices, while ensuring that no other vertices are contained within. (Refer to Diagram 4 in Appendix C) Then it is plain that this has become a $V = 2k - 2$ game. By the induction hypothesis, player 2 will win.

Case 2: If a self-loop is made, player 2 first counts the number of vertices encircled within the loop, and outside the loop. Do note that the number of vertices will add up to $2k - 1$, which is odd. Thus the parity of the number of vertices will be different. Player 2 selects the group of vertices (inside/outside the loop) which are odd, and performs a self-loop on 1 vertex, ensuring no other vertices are contained within. (Refer to Diagrams 5 and 6 in Appendix C) It is plain that this is 2 separate games in which $V = 2j$ for some integer j. By the induction hypothesis, player 2 will win both of these separate games.

Thus, by strong induction, player 1 will win games with $V = 2k + 1$ and player 2 will win games with $V = 2k$.

4.3 Is there a more efficient way to determine the winning strategy of each player?

This research question involves the use of computation by programming. The algorithm used for the programme is written in Appendix D.

V	Player with winning strategy	Computation Time
1	Player 2	Negligible
2	Player 2	Negligible
3	Player 1	4 minutes
4	Player 1	~30 minutes
5	Player 1	~3 hours
6	Player 2	~12 hours

Figure 4.3.1: Table showing the results of computation time as V increases.

As seen, as V increases, computation time increases tremendously, which shows that the computation method used is not efficient. As such, the algorithm has been modified by using the results in research question 1 (shown in Appendix D), and rerun.

V	Computation Time without optimisation	Computation Time with optimisation
1	Negligible	Negligible
2	Negligible	Negligible
3	4 minutes	4 minutes
4	~30 minutes	~15 minutes
5	~3 hours	~90 minutes
6	~12 hours	~4 hours

Figure 4.3.2: Table showing the comparison between when the optimisation is implemented and when it is not implemented

When the optimisation is implemented, the computation time drops drastically. However, the code still runs at a slow pace, as seen when $V = 6$, the computation time even with optimisation is still around 4 hours.

5 Future plans

In the future, more ways will be used to solve the problem such as using modular arithmetic and also improve on the efficiency of the computing of winner for bigger values of V, since the computing time taken for bigger values of V is extremely long, rendering the method inefficient.

6 Conclusion

For research question 1, the results obtained confirm previous research done on this topic. When $V = 1, 2$, player 2 wins. When $V = 3, 4, 5$, player 1 has a winning strategy. Furthermore, a more efficient way has been proposed to solve this problem by reducing the number of cases needed for manual checking.

For research question 2, it was found that when $D \geq 4$, the game would end in a draw. When $D = 2$, the player that has the winning strategy will depend on the parity of the number of vertices V. Player 1 has a winning strategy when V is odd, if not player 2 has a winning strategy when V is even.

For research question 3, the code used produced same results in line with *Lemoine and Viennot (2015)*. Furthermore, the code was optimised by using results in research question 1, cutting down on computing time significantly.

7 References

Applegate, D., Jacobson, G., & Sleator, D. (1991). Computer Analysis of Sprouts. Carnegie Mellon University Computer Science Technical Report, N. CMU-CS-91-144.

Focardi, R., & Luccio, F. (2004). A modular approach to Sprouts. Discrete Applied Mathematics, 144(3), 303-319. doi: 10.1016/j.dam.2003.11.008

Lemoine, J., & Viennot, S. (2015). Computer analysis of sprouts with Nimbers. Games Of No Chance 4, 63, 161-181.

Appendix A:

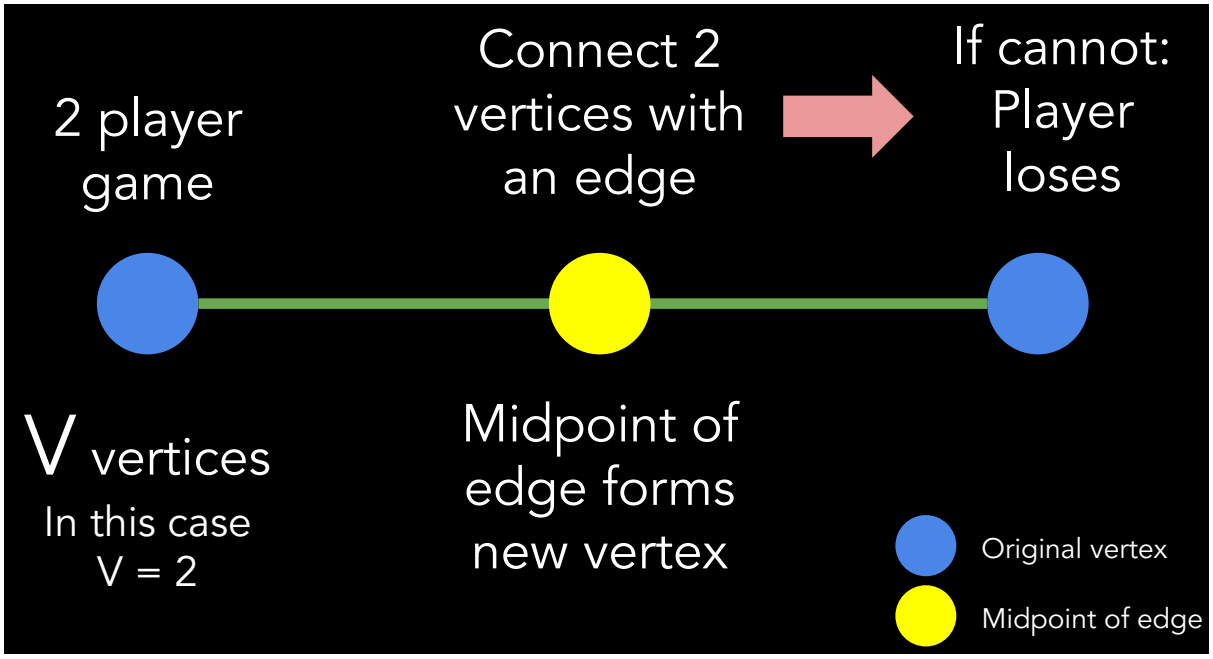


Diagram 1: How 2 vertices are connected, with midpoint marked out as a new vertex.

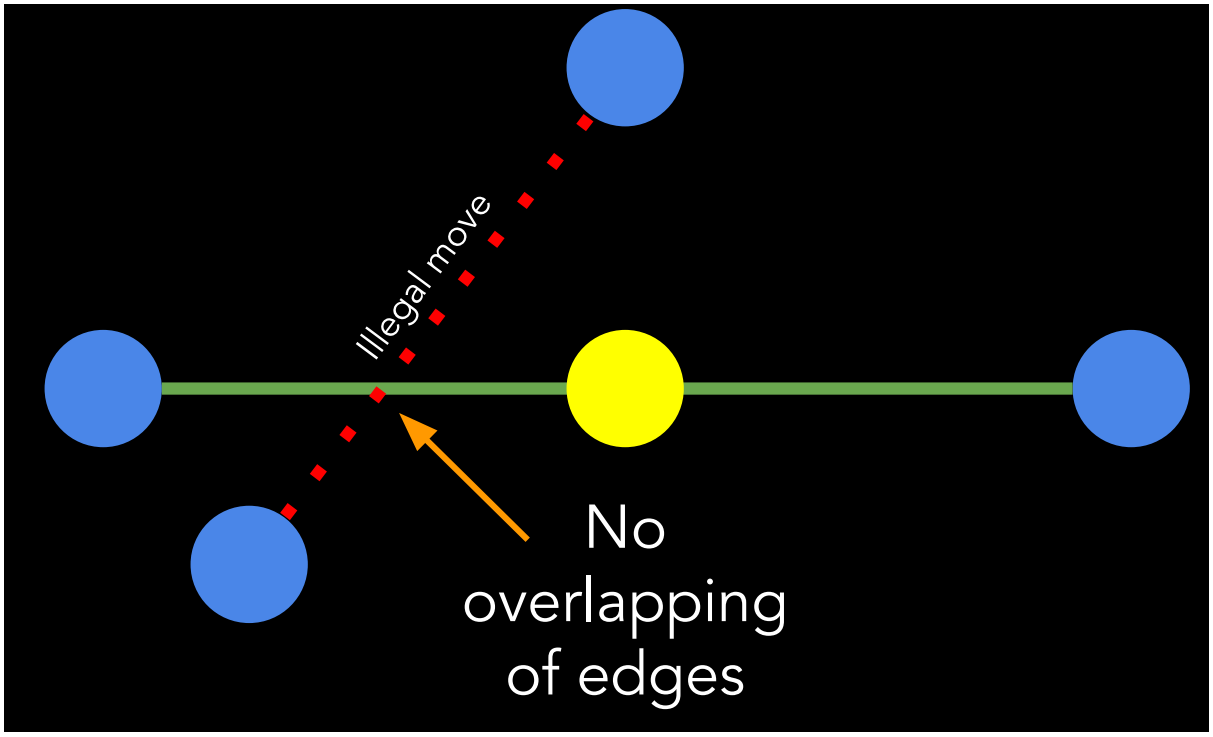


Diagram 2: Edges cannot overlap one another.

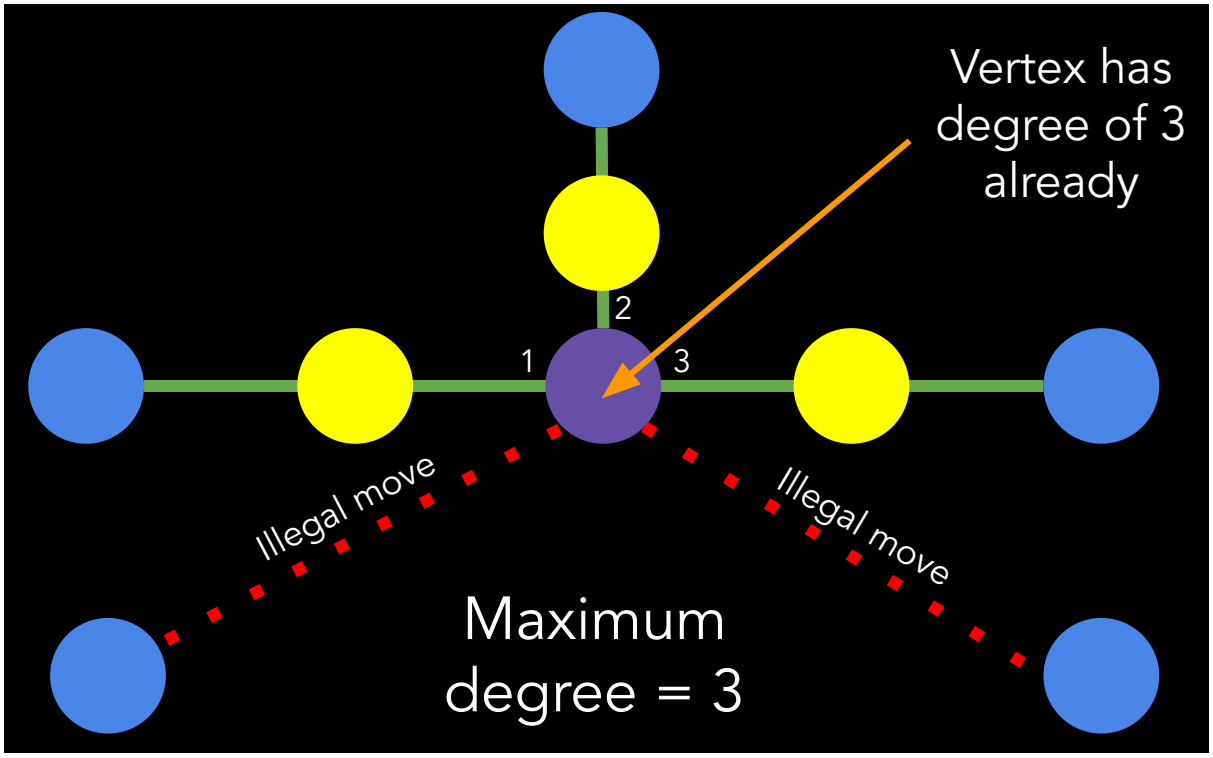


Diagram 3: Each vertex must have a maximum degree of 3

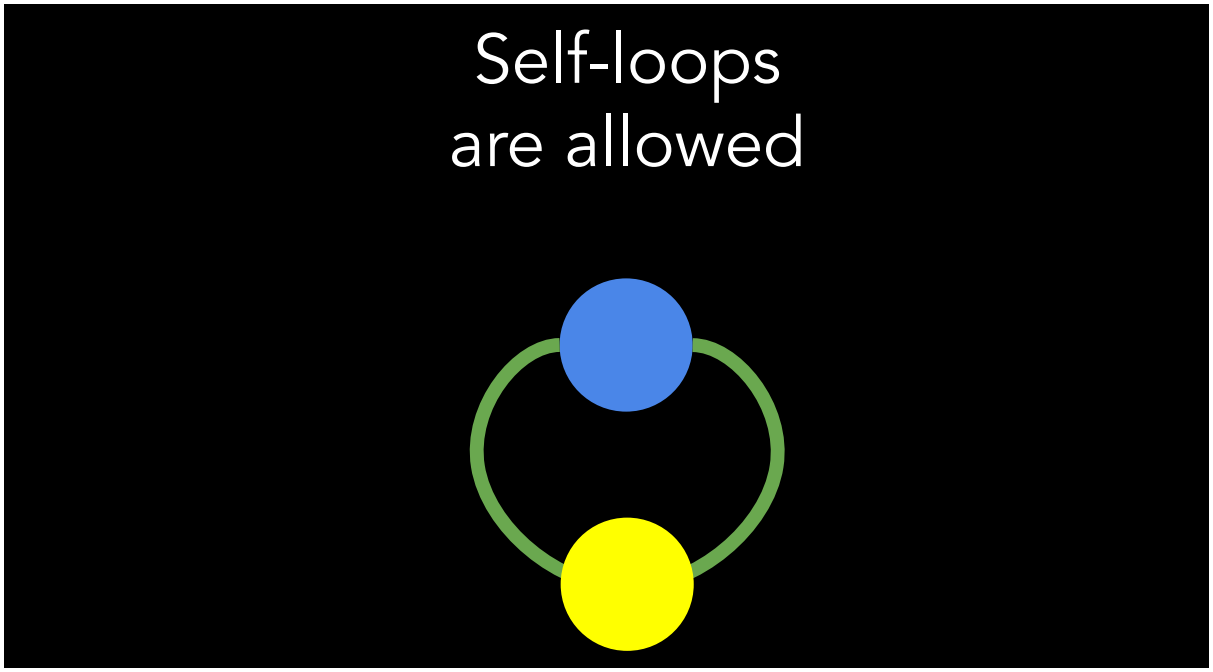


Diagram 4: Self-loop, an edge that has the same starting and ending vertex.

Appendix B:

All diagrams of the proof are hand drawn. All diagrams are at the very end of this document.

Appendix C:

For all diagrams below, the number in the green circles represent the order of moves made. Note: Midpoints have been omitted from diagrams as they are redundant for reasons stated in the proof.

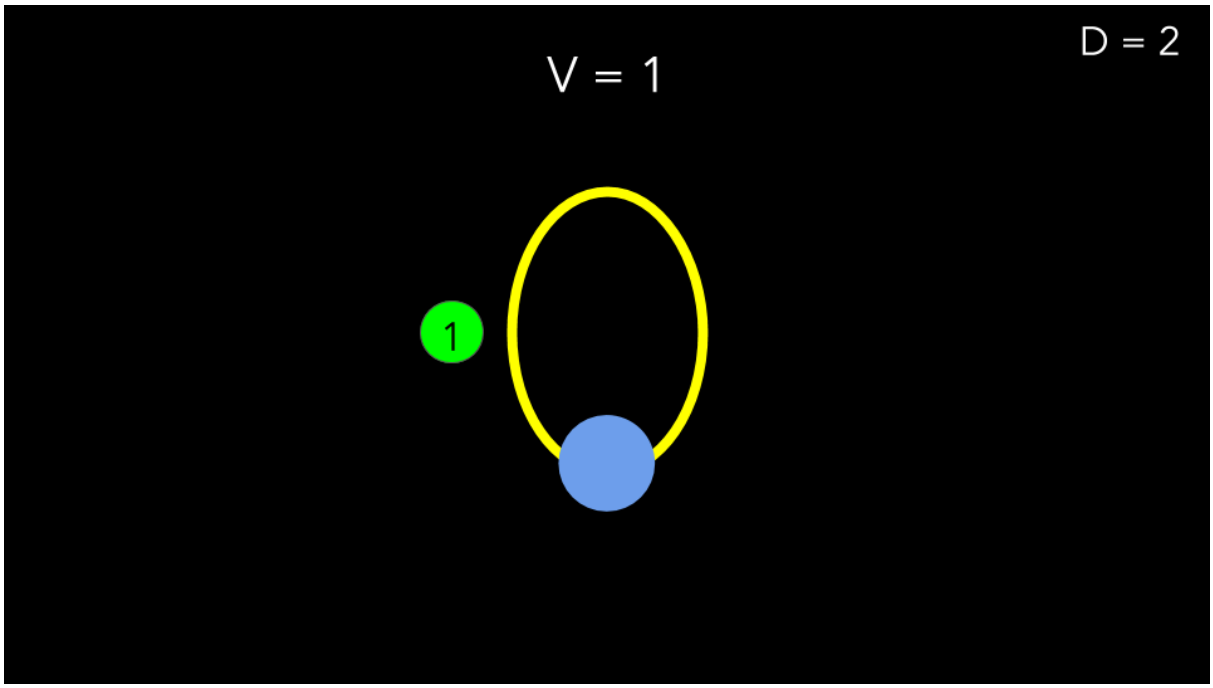


Diagram 1: Proof of D = 2 when $V = 1$

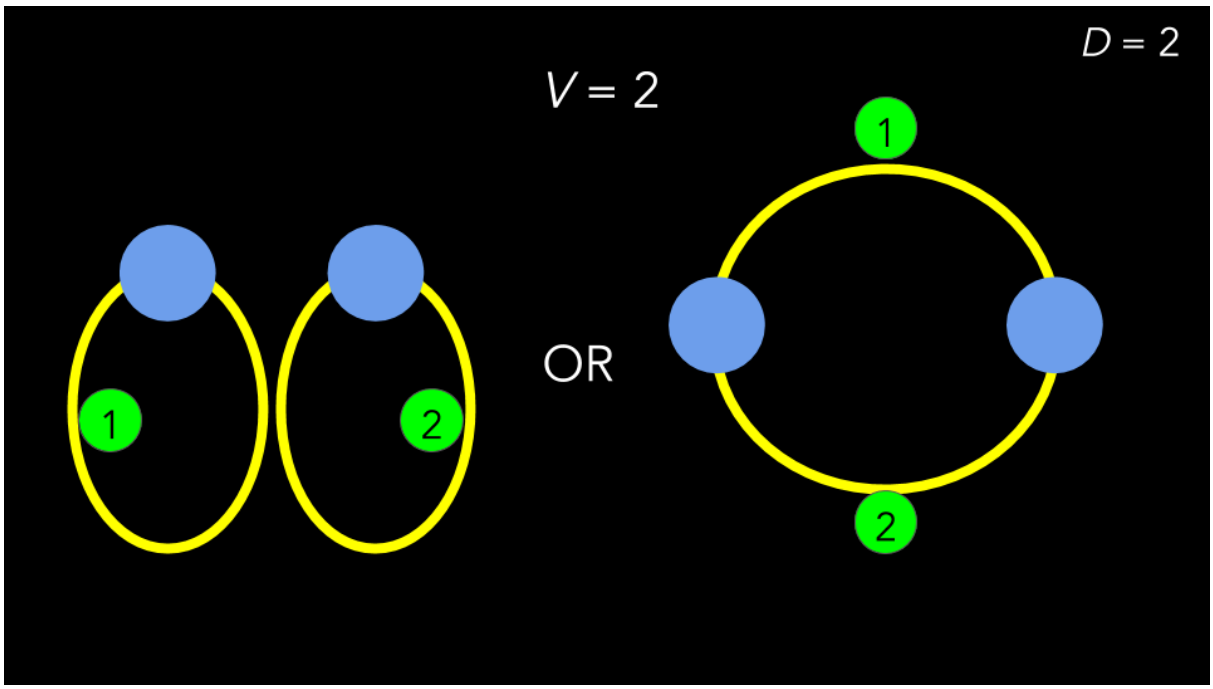


Diagram 2: Proof of D = 2 when $V = 2$

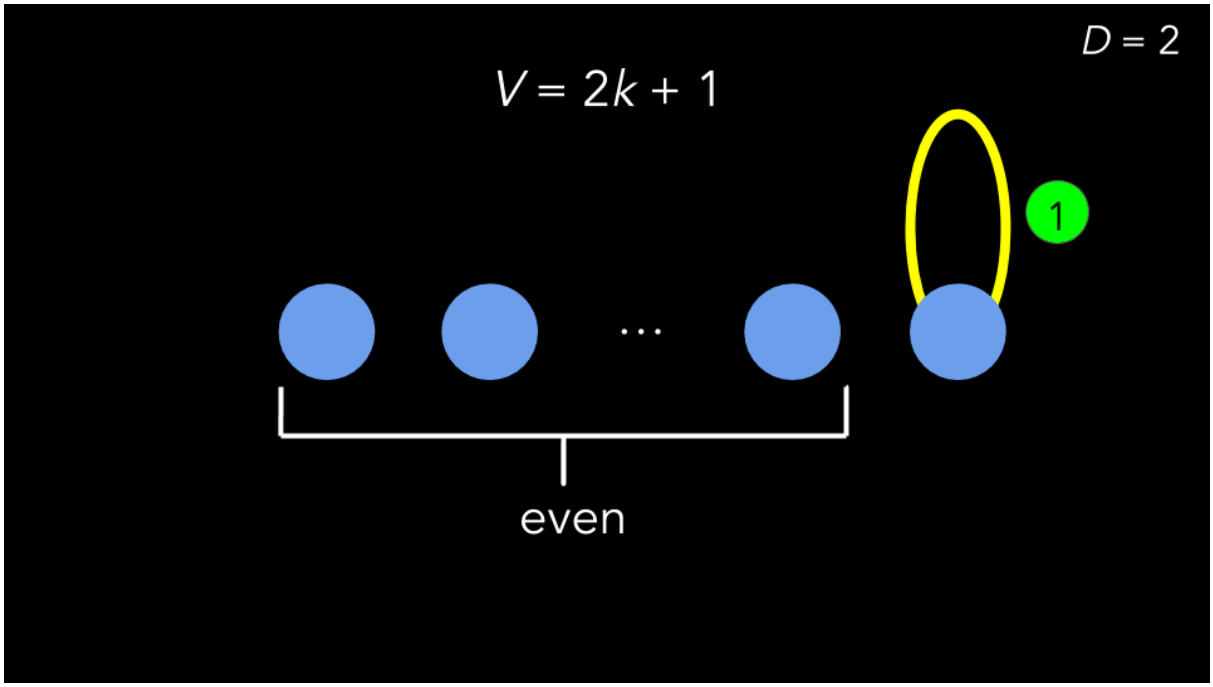


Diagram 3: Proof of D = 2 when $V = 2k + 1$

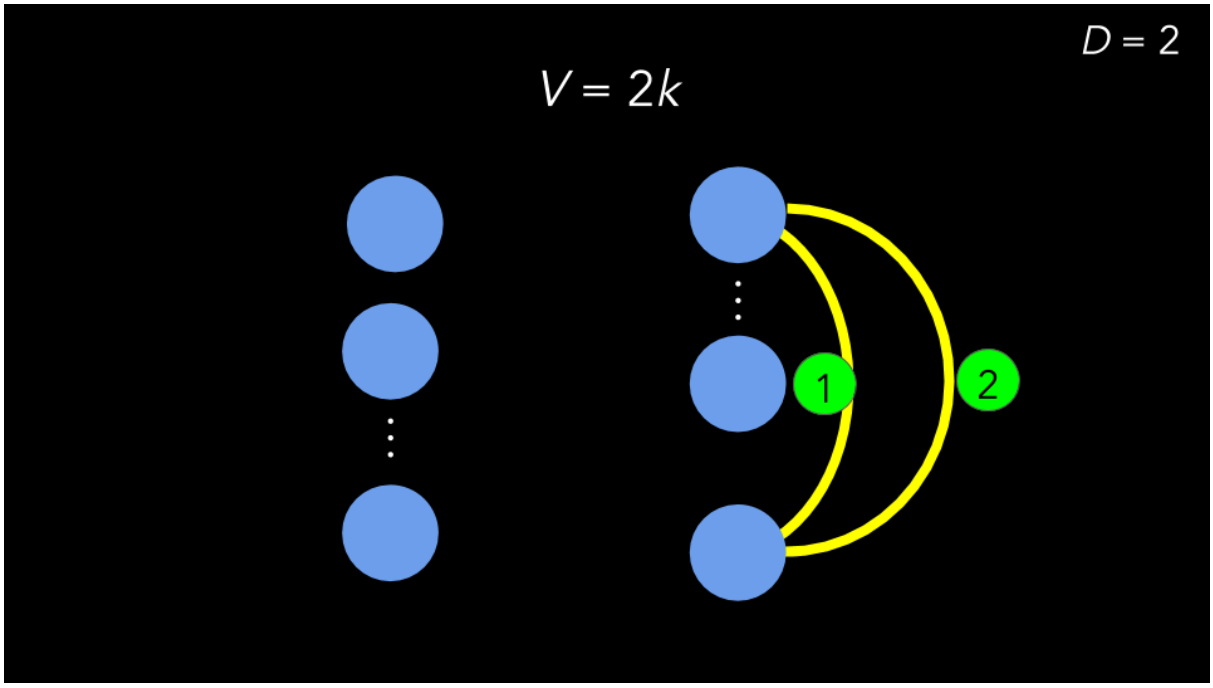


Diagram 4: Proof of D = 2 when $V = 2k$ (Case 1)

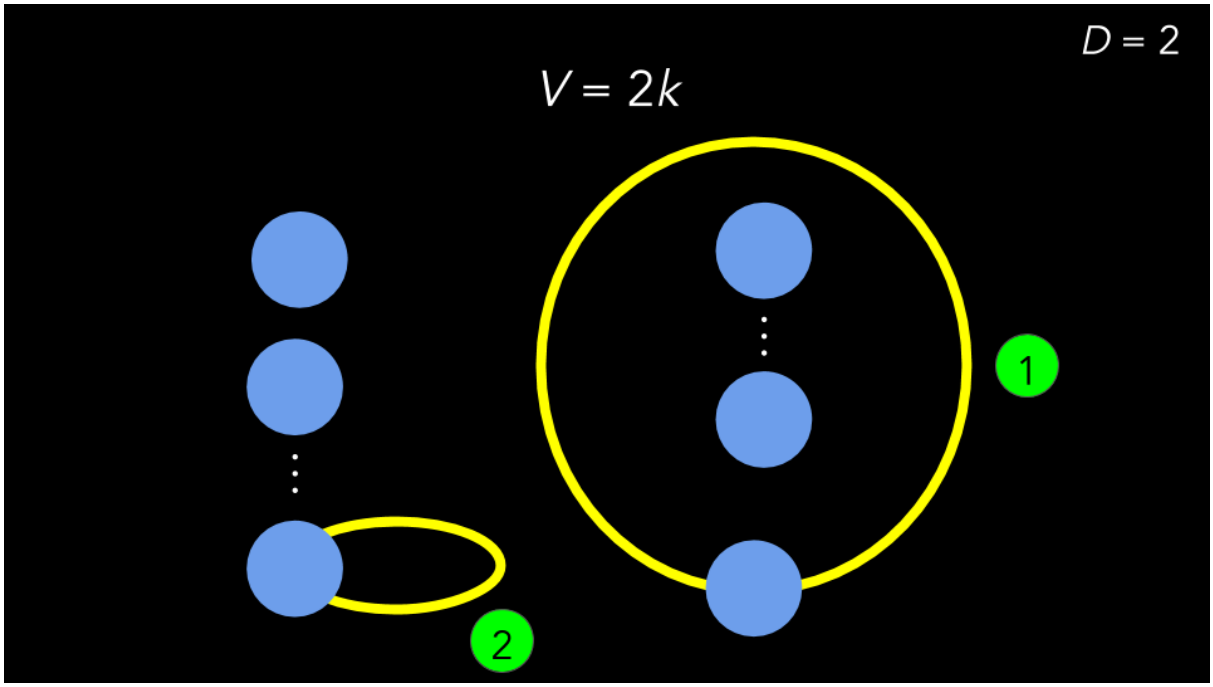


Diagram 5: Proof of D = 2 when $V = 2k$ (Case 2 Part 1: Number of vertices outside loop is odd)

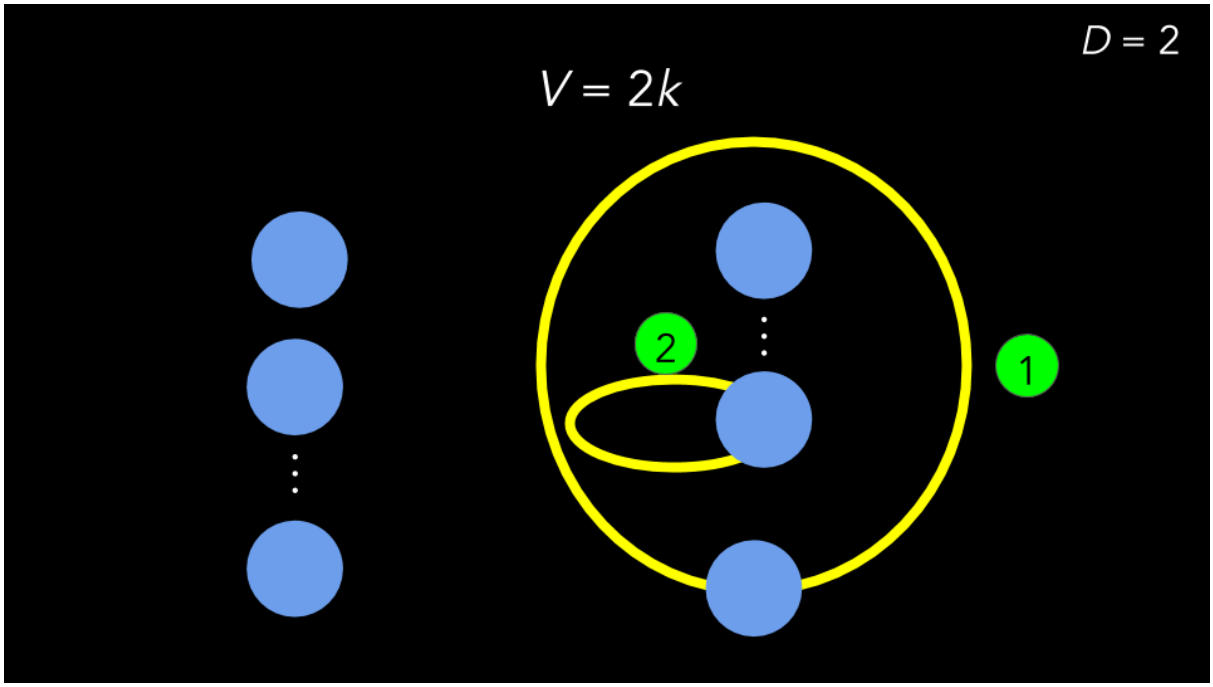


Diagram 6: Proof of D = 2 when $V = 2k$ (Case 2 Part 2: Number of vertices inside loop is odd)

Appendix D:

Pseudo-code of the algorithm used without optimisation strategies from RQ1 (*Lemoine and Viennot, 2015)*:

```
function compute-win-loss(position P){
    let S = set of states of game after one move
    for each element in S:
        if element is a loss, return “P is a win”
    return “P is a loss”
}
```

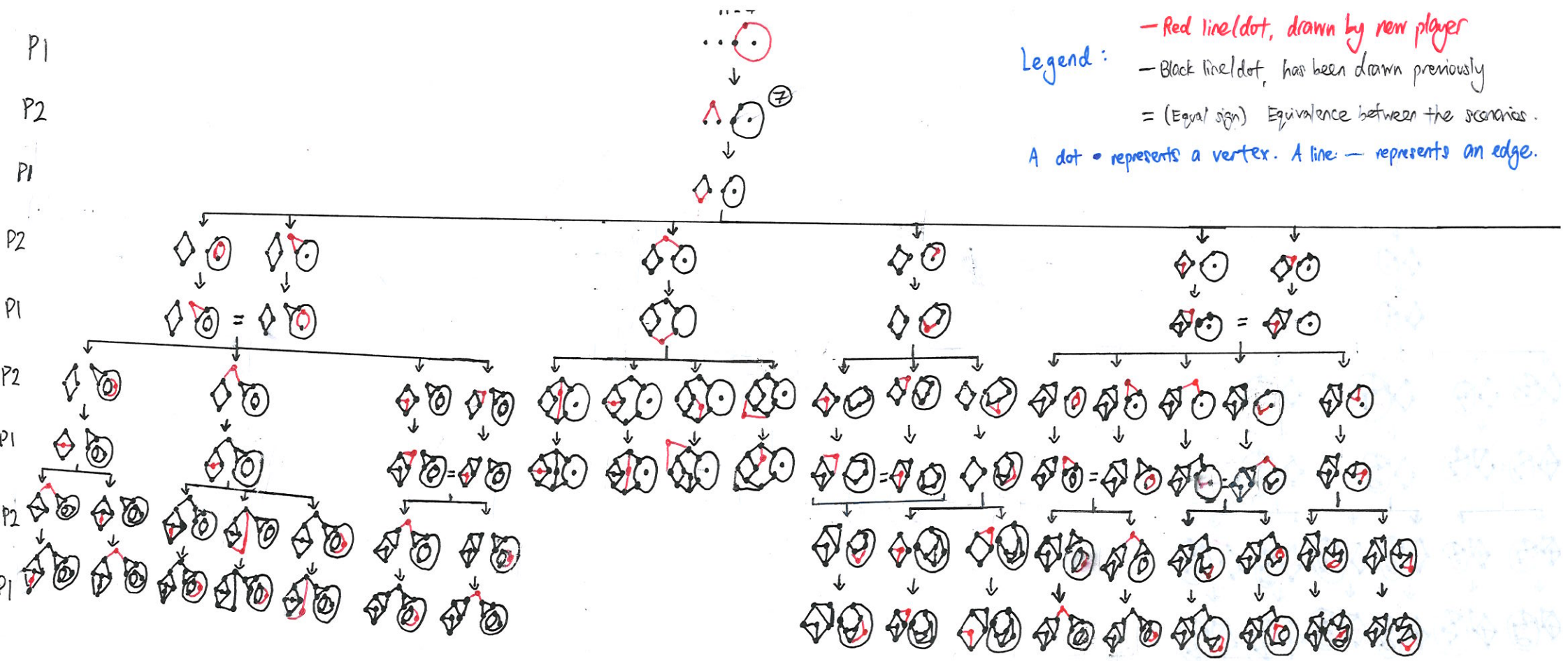
Sample code in Python 3:

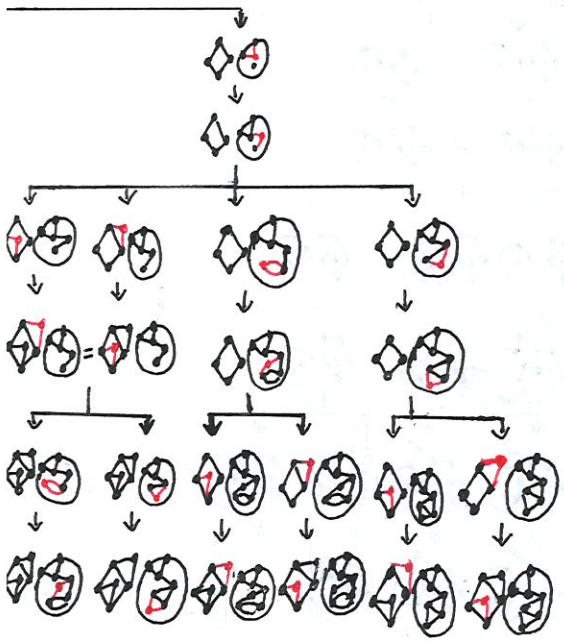
```
import planarity
import networkx as nx
graph = nx.Graph()
max_nodes = 3
connected = {}

def winloss(P, curmax, mp): # P stands for the position, curmax stands for number of new edges added
    for i in range(1, max_nodes + curmax):
        for j in range(1, max_nodes + curmax):
            if i == j: # self loop?
                continue
                if i in mp: # found
                    if mp[i] > 1: # cannot perform self loop on node i
                        #print("Fail overload ", i, i)
                        continue

                    else: # mp[i] = 1
                        nextmp = mp.copy() # pass in function
                        nextmp[i] = 3
                        newnode = curmax + max_nodes + 1
                        Q = P # pass in function
                        Q.add_edge(i, newnode)
                        Q.add_edge(newnode, i)
                        nextmp[newnode] = 2
                        if planarity.is_planar(Q) == False:
                            #print("Fail not planar ", i, i)
                            continue

                        result = winloss(Q, max_nodes + 1, nextmp)
                        if result == False:
                            return True
                        # no need to check for planarity
                else: # not found
                    #print("Creating .", i)
                    nextmp = mp.copy() # pass in function
                    nextmp[i] = 2
                    newnode = curmax + max_nodes + 1
```

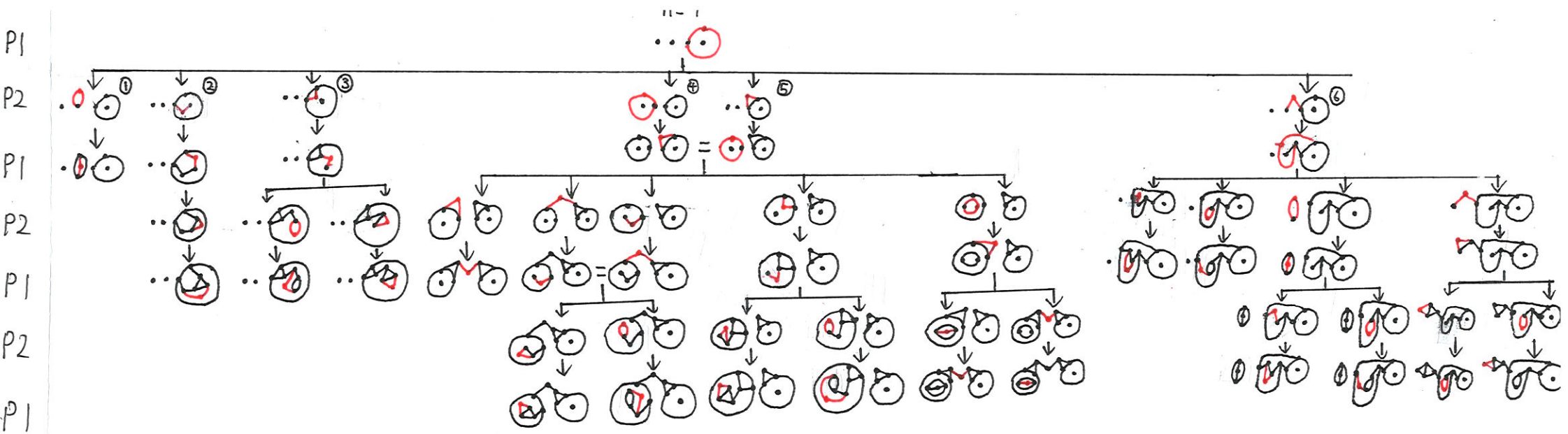
```
                    Q = P # pass in function
                    Q.add_edge(i, newnode)
                    Q.add_edge(newnode, i)
                    nextmp[newnode] = 2
                    if planarity.is_planar(Q) == False:
                        #print("Fail not planar ", i, i)
                        continue
                    result = winloss(Q, max_nodes + 1, nextmp)
                    if result == False:
                        return True
                    continue
                    # no need to check for planarity
            else: # no self loop
                if i in mp and mp[i] > 2: # exceeds
                    #print("Fail overload ", i)
                    continue

                if j in mp and mp[j] > 2: # exceeds
                    #print("Fail overload ", j)
                    continue

                nextmp = mp.copy() # pass in function
                if i not in nextmp: # checks for existence
                    #print("Creating ..", i)
                    nextmp[i] = 0

                if j not in nextmp:
                    #print("Creating ..", j)
                    nextmp[j] = 0

                nextmp[i] += 1
                nextmp[j] += 1;
                newnode = curmax + max_nodes + 1
                nextmp[newnode] = 2
                Q = P # pass in function
                Q.add_edge(i, newnode)
                Q.add_edge(newnode, j)
                if planarity.is_planar(Q) == False:
                    #print("Fail not planar ", i, j)
                    continue

                result = winloss(Q, max_nodes + 1, nextmp)
                if result == False:
                    return True

    return False

def main(): # main function
    result = winloss(graph, max_nodes, connected)
    print(result)

main()
```

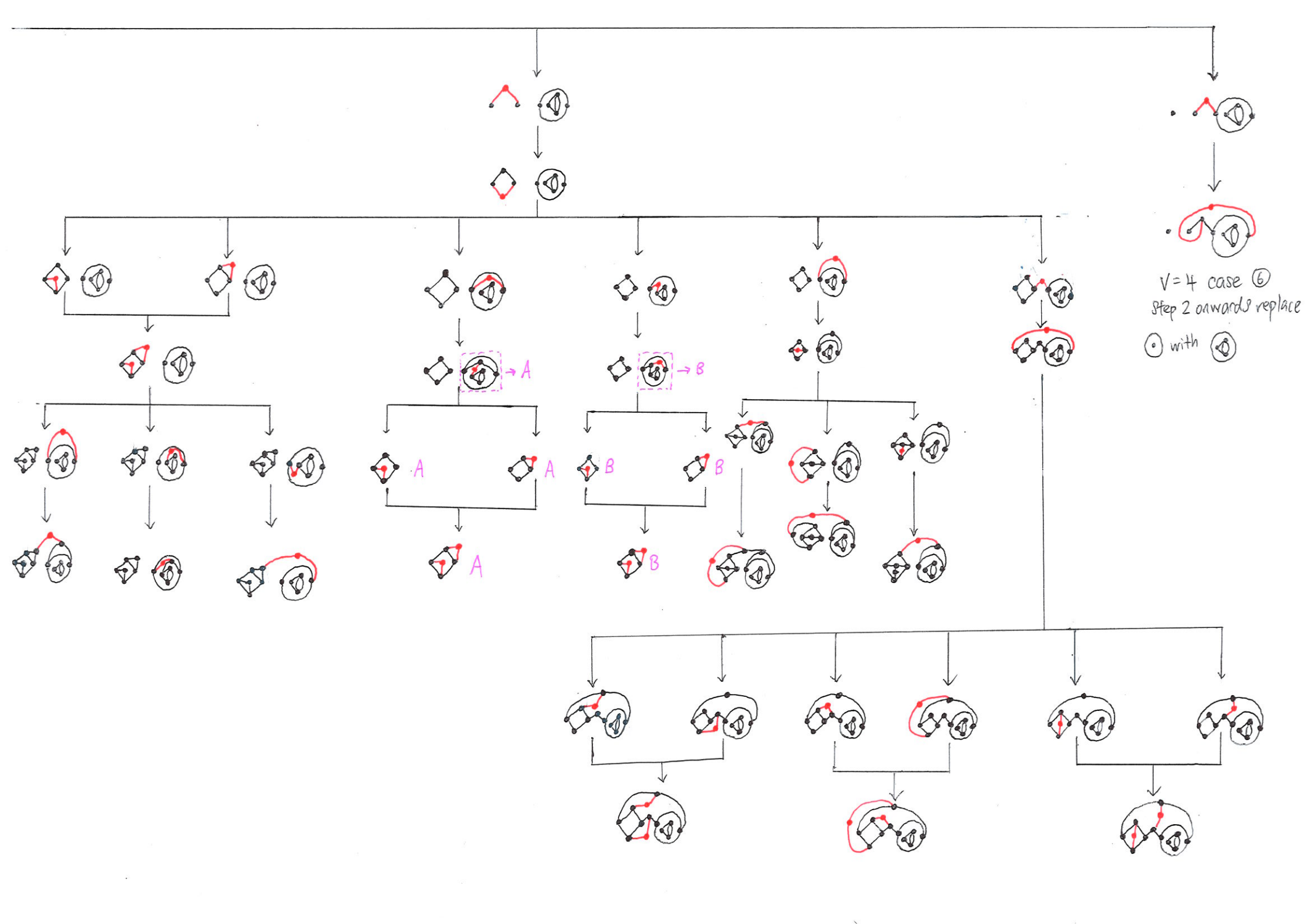
Pseudo-code of the algorithm used with optimsation strategies from RQ 1:

```
let A = set of precomputed states of games // (from RQ1)
function compute-win-loss(position P){
    let S = set of states of game after one move
    for each element in S:
        if any element of A = element in S, return win/loss
        if element is a loss, return “P is a win”
    return “P is a loss”
}
```


P1
P2
P1
P2
P1
P2
P1
P2
P1
Legend :
– Red line/dot, drawn by new player
– Black line/dot, has been drawn previously
= (Equal sign) Equivalence between the scenarios.
A dot • represents a vertex. A line — represents an edge.




P1
P2
P1
P2
P1
P2
P1
①
②
③
④
⑤
⑥


A
A
A
A
B
B
B
B
V = 4 case ⑥
Step 2 onwards replace
⊙ with