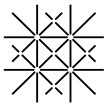

University of Basel

Independent Set Reconfiguration Sequences on Bipartite Permutation Graphs using Sliding Token.

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Research Group
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Liat Cohen

Patrick Steiner
pa.steiner@stud.unibas.ch
2016-053-605

22.06.2022

Acknowledgments

I wholeheartedly thank Dr. Liat Cohen and Dr. Prof. Malte Helmert of the AI-Research group of the University of Basel for the great project idea which I was able to conduct as my bachelors thesis. This has been the most positive working environment I have ever participated in.
To my university friends, that were by my side and heard all of my inane thought processes while you worked on your own projects, thanks for listening when you could.

To my loved ones, that gave me the patience and space that I needed to see this through, from the beginning of my bachelors degree to its culmination in this thesis. Thank you.

Abstract

Bipartite Permutation Graphs have interesting algorithmic properties which make it possible to solve several PSpace-complete problems (including the sliding token problem) in polynomial time when restricted to Bipartite Permutation Graphs.
In this thesis we combined the algorithms of two papers to create a python implementation that checks if a graph is a bipartite permutation graph and if two independent sets for this graph can be reconfigured into one another using sliding token movement in polynomial time. We finished the entire process by giving out an actual reconfiguration sequence after we've checked that one exists. An interesting conclusion which developed during the implementation leads to the discussion if part of the original algorithm could be substituted to reach linear time complexity for the recognition of the existence of sliding token reconfiguration sequences.

Table of Contents

Acknowledgments ii

Abstract iii

1 Introduction 1

2 Background 2
- 2.1 Bipartite Permutation Graphs . . . 2
  - 2.1.1 Bipartite Graphs . . . 2
  - 2.1.2 Permutation Graphs . . . 2
  - 2.1.3 Bipartite Permutation Graphs . . . 3
  - 2.1.4 Illegal structures for BPG . . . 4
- 2.2 Independent sets . . . 4
- 2.3 Reconfiguration Problems . . . 5
  - 2.3.1 Independent Set Reconfiguration . . . 5
  - 2.3.2 Sliding Token . . . 5

3 Sliding Token Algorithm with Bipartite Permutation Graph recognition 7
- 3.1 BPG recognition algorithm . . . 7
  - 3.1.1 Bipartite Recognition Algorithm . . . 7
  - 3.1.2 Permutation Recognition Algorithm . . . 8
- 3.2 Unlocked Graphs and Rigid Tokens . . . 9
  - 3.2.1 Preparation and execution of I+ . . . 12
- 3.3 Finding Reconfiguration sequences . . . 12

4 Python Implementation 13
- 4.1 UML-Diagram . . . 13
- 4.2 Using the Code . . . 13
- 4.3 SlidingToken.main() . . . 14
- 4.4 BPG Check . . . 15
  - 4.4.1 Bipartite Check . . . 15
  - 4.4.2 Permutation Check . . . 15
- 4.5 Rigid Tokens . . . 16
  - 4.5.1 Preparing for I+ . . . 16

4.6 I+ 17
4.7 Breadth First search for Reconfiguration sequences 17

5 Examples **18**
5.1 Rigid tokens with reconfiguration sequence 19
5.2 No rigid tokens with reconfiguration sequence 20
5.3 Rigid tokens without reconfiguration sequence 21

6 Discussion **22**
6.1 Results 22
6.1.1 Possible alternative to I+ 22
6.2 Related Questions 23
6.3 Further Work 23

Bibliography **25**

Appendix A Appendix **26**
A.0.1 Code 26

1 Introduction

In this report we present all collected information to implement the algorithms for "Reconfiguration sequence recognition on bipartite permutation graphs" [3] and "Bipartite permutation graph recognition" [6] together with an output of an actual reconfiguration sequence into one workflow into python.

To complete the implementation of this workflow, from the recognition of a bipartite graph over to the recognition of independent sets to the output of a reconfiguration sequence we studied many papers that dealt with the subject. We have created the groundwork for further work in the research of this specific topic of graph theory with optimization possibilities on the horizon.

Over the course of this project multiple papers were considered, discussed and either accepted as basis or thrown out. Every accepted paper was studied thoroughly to extract the important steps for our workflow, which proved challenge again and again.

The rest of the report is organized as follows: In section Background we introduce all necessary pieces to understand the Main part. Our Main part consists of two sections, we first introduce every important step in the algorithms used and how they relate to each other. In the second, we have an in depth discussion of the actual implementation as it can be found in the code. In the final section we discuss our findings and how this project lays the basis for further research.

2
Background

In this part, the most important concepts and definitions prior to the main algorithm are introduced. It is assumed that Graph Theory is known to the reader to a basic extent.

2.1 Bipartite Permutation Graphs

In this section we're going to introduce Bipartite Permutation graphs (BPG), starting with the greater graph classes they are an intersection of, Bipartite graphs and Permutation graphs.

2.1.1 Bipartite Graphs

Bipartite Graphs are a family of graphs that have the characteristic of being able to be split its vertices into exactly two sets A and B [6]. These vertices inside of each set are not neighbours to each other, the neighbours of each vertex can be found in the other set. Say vertex a is in set A, then the neighbours of a are all inside of set B.

Figure 2.1 depicts a simple bipartite graph that can be split into groups of even and uneven index numbers, and as you can see, none of the even numbers have edges to other even numbers and vice versa for the uneven numbers.

2.1.2 Permutation Graphs

Permutation Graphs are another family of graphs that are recognized by the following characteristic. A graph is a permutation graph if there is some pair P,Q of permutations of the vertices such that there is an edge between vertices x and y iff x precedes y in one of P,Q while y precedes x in the other.
Such a permutation graph can also be described as a permutation diagram. Given two columns where the first consists of the vertices in the ordering given by P, the other by Q. A line connects the vertex of the same value in both columns, then there is an edge between vertices x and y iff the lines of x and y cross in the diagram. There can be multiple

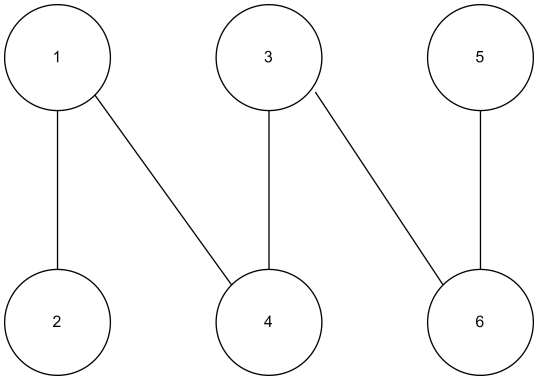

Figure 2.1: A Simple Bipartite Graph

permutation diagrams for the same permutation graph [6].
Figure 2.2 shows a permutation graph and diagram combo. As you can compare the diagram with the graph, it becomes obvious that the graph is a permutation graph.

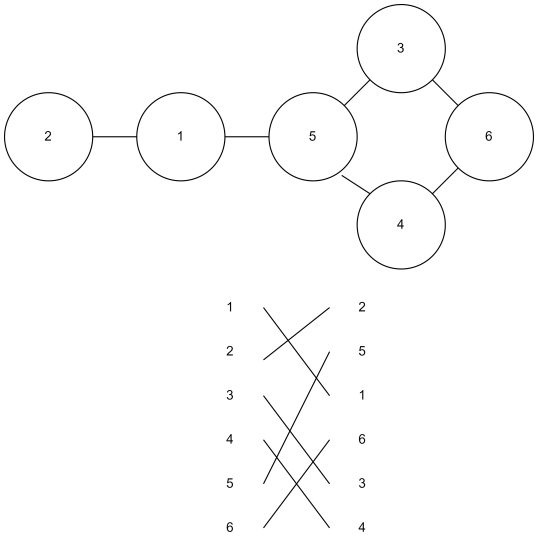

Figure 2.2: A Permutation Graph with accompanying Permutation Diagram[6]

2.1.3 Bipartite Permutation Graphs

Bipartite Permutation Graphs (BPG) are an interesting subfamilies of perfect graphs taking their naming and attributes from the two families of graphs they stem from. "Bipartite Permutation Graphs" shows that BPG have good algorithmic properties and can be recognized in linear time [6]. Sub graphs of BPG are also always BPG.
Most importantly, several NP-complete or of unknown complexity can be solved in polynomial time when restricted to BPG. Among these problems are the Hamilton Circuit problem, a variant of the Crossing Number problem and the minimum fill in problem [3]. Over the course of this paper we will discuss the Reconfiguration problem using Sliding token.

2.1.4 Illegal structures for BPG

Now that we have seen all the characteristics that come together to form BPG, we can talk about illegal structures for BGP. These structures were most important to use whilst testing the validity of the code and come in many examples. For a structure to be illegal for BPG, it must break the characteristics of either bipartite or permutation graphs.

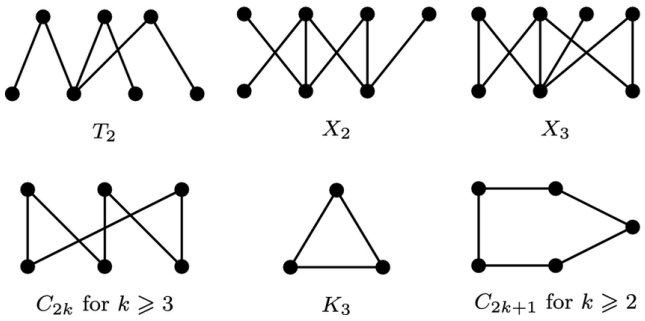

Figure 2.3: Illegal structures of BPG [1]

In those structures, we can see a few that already poke out. The odd length cycles are a violation for Bipartite graphs, whilst more subtly, T2 for example can not be a permutation graph due to its violation of the permutation graph characteristic.

Any graph containing any of the graphs in figure 2.3 as sub graphs are not BPG [1].

2.2 Independent sets

An Independent Set is a subset of the vertices in a graph, of which no two are neighbours to each other. A vertex in an independent set is called token, the size of an independent set is corresponding to the amount of tokens in it [3]. There are further definitions for independent sets, but what we have already established suffices for the rest of the paper. Figure 2.4 shows a simple graph with an independent set of size 2.

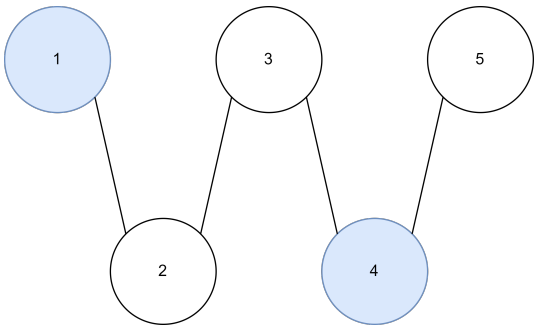

Figure 2.4: Independent set on a graph marked blue

2.3 Reconfiguration Problems

A Reconfiguration problem is the issue of finding a step-by-step transformation between a pair of feasible solutions, where each step in itself is also a valid solution.

2.3.1 Independent Set Reconfiguration

In the context of this paper we want to find reconfiguration sequences between independent sets. As mentioned before, there are other problems that can be solved on BPGs. But the ways that all of these problems can be solved are quite different as well. At the beginning of this thesis, we had chosen the problem of set reconfigurations, but the method of movement to use was not. How were our tokens meant to move? A few papers presented themselves, for which we give a short overview and discuss more deeply the method we finaly chose.
To introduce a few of the presented methods:

1. Token Jumping: A token may move to any field which is a valid field to jump to, i.e. it is not an illegal move in the context of independent sets.
2. Token addition/removal: A token may be removed and in a next step added to another field.
3. Sliding Token: A token may move only over the graphs edges.

Token jumping and token addition/removal are NP-complete whilst sliding token is PSpace-Complete [5]. In the end, we've decided on sliding token movement and the Epstein Paper.

2.3.2 Sliding Token

Sliding Token is a reconfiguration problem in which tokens may only move over the graphs edges, iff they do not break the rules for independent sets in our context.

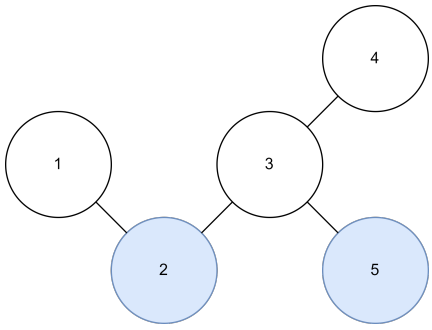

Figure 2.5: An independent set with currently blocked vertex 3

In Figure 2.5 we can see that if we want to move 5 over 3 to 4, we would first have to move 2 to 1, since 2 and 5 are both direct neighbours to 3. If we immediately moved 5 to 3 we would break the adjacency rule for independent sets.

Sliding tokens stand apart from the other methods due to its restrictions on movement, which gives us two more categories in our BPGs: Locked and Unlocked. An unlocked Independent set can move its tokens freely without any restrictions, whilst a locked Independent set has tokens, which may not move at all. We call such tokens Rigid [3].
As shown in figure 2.6, the first independent set can never move without breaking the rules for independent sets using sliding token movement, where the second independent set is unlocked but could not move legally to a position like the first independent set.

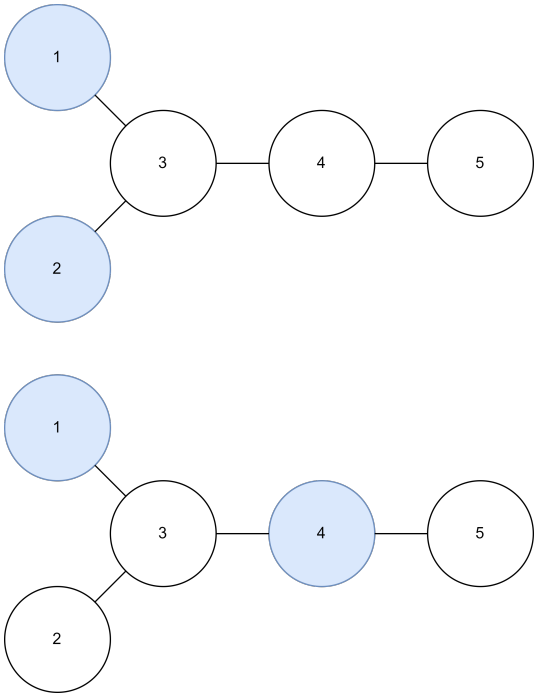

Figure 2.6: The same Graph with a rigid independent set and an unlocked independent set

These restrictions are what make Sliding Tokens so different and interesting to approach.

3

Sliding Token Algorithm with Bipartite Permutation Graph recognition

The main subject of this paper is the fusion of multiple algorithms, expecting a connected graph. The workflow of these algorithms start from a check for BPGs using the authors creation for a recognition algorithm for bipartite graphs and the permutation graph recognition algorithm of the paper "Bipartite Permutation Graphs" [6], over to the recognition algorithm if a reconfiguration sequence can exist [3] to finish with a proper reconfiguration sequence of the authors own creation. The input for this workflow are the edges of a graph G and the independent sets I and J which are going to be reconfigured into each other, if possible. The overall time complexity for this workflow lies in polynomial time.

3.1 BPG recognition algorithm

The BPG recognition algorithm used in this thesis was created through a combination of the authors work for a bipartite check and the paper "Bipartite Permutation Graphs" for the permutation check. The mentioned paper requires the input graph to be bipartite already. The combined algorithm has linear time complexity [6].

3.1.1 Bipartite Recognition Algorithm

The bipartite algorithm was put up by checking all of the edges in the input graph. By comparing the vertices at the edge ends with the elements in bipartite sets A and B, one decides where the edges go or it is not bipartite. The following function describes all possible steps with $\langle x, y \rangle$ being an edge and A and B being initially empty Bipartite sets:

$$F(\langle x,y\rangle) = \begin{cases} x \in A, y \in B & \text{if } x \notin A \text{ and } x \notin B \text{ and } y \notin A \text{ and } y \notin B \\ \text{no additions} & \text{if } x \in A \text{ and } y \in B \text{ or } y \in A \text{ and } x \in B \\ x \in A & \text{if } x \notin A \text{ and } x \notin B \text{ and } y \notin A \text{ and } y \in B \\ x \in B & \text{if } x \notin A \text{ and } x \notin B \text{ and } y \in A \text{ and } y \notin B \\ y \in B & \text{if } x \in A \text{ and } x \notin B \text{ and } y \notin A \text{ and } y \notin B \\ y \in A & \text{if } x \notin A \text{ and } x \in B \text{ and } y \notin A \text{ and } y \notin B \\ \text{Graph is not Bipartite} & \text{else} \end{cases}$$

3.1.2 Permutation Recognition Algorithm

Once we've shown that a graph is bipartite, we can proceed to the recognition algorithm introduced in "Bipartite Permutation Graphs" to check for permutation graphs. The general idea of the algorithm is to maintain a data structure which allows us to represent all possible permutations of a bipartite set A[...] with the adjacancy and enclosure properties (which will not be discussed in this paper but can be found in "Bipartite Permutation Graphs") in linear time. Vertices in B are examined one at a time, each one forcing more restrictions on the (overlaying) permutation [6].

The algorithm starts out by finding a vertex in B which has a minimal amount of neighbours. Those neighbours (without the initial vertex) form the first block in our Block List BList. Vertices from A are called new if they are not in any block of BList, else they are old.The blocks of BList must occur consecutively, and the vertices of N(b) must occur consecutively. We call the left most Block BL and the right most Block BR.

We now go step wise through the rest of the vertices b ∈ B which are adjacent to at least one old vertex. We divide the description of the algorithm into four cases, depending on the composition of the neighbours of b, which we shorten to N(b).

For each vertex b ∈ B, let ADJ(b) be the set of blocks in BLIST which contain at least one vertex from N(b). Let MIX(b) be the set of blocks which contain at least one vertex from N(b), and at least one vertex from A-N(b).

1. Case 1: There is a new vertex in N(b).
   For these cases, we create a new Block BNew, consisting of all new vertices in N(b)

   a) Case 1.1: Some old vertex is not in N(b).
      If BList has only one block, we place BNew at either end. Else, we determine on which end BNew needs to be placed. If BL is not in ADJ(b), or if BL is in MIX(b) and BR is in ADJ(b), then BNew is placed at the position of BR. Otherwise, BNew is placed at the position of BL.
      After BNew has been placed, we modify the old Blocks of BList so that the vertices in N(b) are forced to occur consecutively.If there is any block B in MIX(b), remove the vertices of B which are in N(b) from B, creating a new block C. B

cannot come between C and BNew if the adjacency property is satisfied, so C is placed on the side of B which is closest to BNew.

b) Case 1.2: All old vertices are in N(b).
In this case, BNew is always placed at the right end of BList. If there is only one Block, the choice is completely arbitrary. If there is more than one block the choice is enforced place at the right side to not enclose the initial block in BList. There can only be one block in ADJ(b) for this.

2. Case 2: All vertices in N(b) are old.

a) Case 2.1: N(b) contains vertices from two distinct blocks of BList.
We need to guarantee that vertices of N(b) occur consecutively. For any block B which is in MIX(b), remove the vertices of N(b) from B to form a new block C. Place C between B and the block which is next to B in BLIST which is also in ADJ(b).

b) Case 2.2: N(b) is completely contained in a single block of BList.
If all vertices in B are in N(b), do nothing. Otherwise, remove the vertices of N(b) from B, forming a new block C. If B = BR, then put C at the end of BLIST. If B = BL, place C at the front of BLIST. If B is not BL or BR, then the graph is not a permutation graph.

If we have found a yes instance for this algorithm as well then we are dealing with a BPG. For further information and clarification, see the paper"Bipartite Permutation Graphs" [6].

3.2 Unlocked Graphs and Rigid Tokens

After we've found out if we're dealing with a BPG, we check to see if there are rigid tokens or not. From this section on wards, the algorithms and definitions stem from the paper "Sliding Token on Bipartite Permutation Graphs" [3] unless stated otherwise. We see this part of the workflow through with both input independent sets I and J. Unlike the previous check, their existence does not hinder our ability to continue our workflow, unless if I and J do not share the same rigid tokens. If there are rigid tokens, we just need to take some extra steps.

The following section talks about the algorithm introduced in "Sliding Token on Bipartite Permutation Graphs" to recognize rigid tokens in a BPG. As mentioned before, rigid tokens are tokens that can not move according to the combined rules of sliding token and independent sets. Rigid tokens are defined as: R(G,I) = $\{v \mid v \in \cap_{I' \in [I]_G} I'\}$. Which roughly reads as: A set of rigid tokens of independent set I under Graph G are said vertices which appear in the intersection of the independent sets I' which are part of the equivalence class of I under G. An independent set with R(G,I) = $\emptyset$ is called unlocked.

Naturally, one would think that an algorithm to find any and all rigid tokens would include finding all of the independent sets in the equivalence class of the original set I under G, but

as described in the paper, an algorithm was found, named "Wiggle", which reliably finds all rigid tokens in linear time without the need to find all independent sets in the equivalence class of the original independent set I. This algorithm is split into two parts: Algorithm 1, which calculates the wiggle room around all tokens and algorithm two, which feeds Algorithm 2 twice with its input values in two different orderings of A and B.

The two algorithms function together as a unit. We input the two bipartite sets, the edges of the graph and an independent set into Wiggle. Wiggle will call SwitchSides with the first configuration of bipartite sets, as can be seen in algorithm 2 line 1. We start SwitchSides by building the tables M and C together. In the first loop of algorithm 1, starting at line 3, we examine each element u of the bipartite set labeled B in algorithm 1. We set C[u] to be the intersection of the neighbourhood of u and the input independent set. If C[u] has only one element, then we add u to the set M.

Once we're done with the initialization of C and M, we go over to the main loop. We remove an arbitrary element u of M, and within C[u] we remove the vertex v that is unique. We build the next independent set I_k from the previous independent set by removing u and adding v. Now we check if M needs any additions, by checking each vertex w in the neighbourhood of v. In C[w], we remove the vertex v, and if C[w] is left with only one element after that, we add the vertex w to M.

We continue this algorithm until M reaches zero elements. We return the independent sets that we've built to wiggle, which will start the algorithm again but this time with the inputs of the bipartite sets A and B switched. In the end we will receive two sets of independent sets that hold possible reconfigurations. It is of note that not all possible reconfigurations in the equivalence class of the input independent set are found, but simple steps of the tokens that can move. Those tokens that appear in all output independent sets are rigid.

Figure 3.1 depicts the algorithm as it can be found in the paper "Sliding Token on Bipartite Permutation Graphs". After we have determined the existence of our rigid tokens, we continue on with preparing and finding I+.

Algorithm 1: SwitchSides(A, B, E, I_0)

Input: Bipartite graph $G = (A \cup B, E)$, independent set I_0
Output: Reconfiguration sequence $\langle I_0, \ldots, I_k \rangle$ where $I_0 \cap I_k \cap A = R(G, I_0) \cap A$ and $k = |I_0| - |R(G, I_0) \cap A|$

$M \leftarrow \emptyset$ // Will hold available slides
$C \leftarrow$ table from vertices to subsets of vertices
// Initialize M
foreach *vertex* $u \in B$ **do**
  $C_u \leftarrow N(u) \cap I_0$
  if $|C_u| = 1$ **then**
    $M \leftarrow M \cup \{u\}$
$k \leftarrow 0$
while $|M| > 0$ **do**
  $k \leftarrow k + 1$
  $u \leftarrow$ remove an arbitrary element u from M // $u \in B$ will be in I_k
  $v \leftarrow$ remove the unique vertex v from C_u // $v \in I_{k-1}$
  $I_k \leftarrow I_{k-1} - v + u$
  foreach *vertex* $w \in N(v)$ **do**
    $C_w \leftarrow C_w - v$
    if $|C_w| = 1$ **then**
      $M \leftarrow M \cup \{w\}$
return $\langle I_0, I_1, \ldots, I_k \rangle$

Algorithm 2: Wiggle(A, B, E, I_0)

Input: Bipartite graph $G = (A \cup B, E)$, independent set I_0
Output: Reconfiguration sequence $\langle I_0, \ldots, I_k \rangle$ with $k \leq 4|I_0|$ such that for all $v \in I_0 \setminus R(G, I_0)$, there is some j where $I_j \setminus I_{j-1} = \{v\}$

$\langle I_0, \ldots, I_{k_1} \rangle \leftarrow$ SwitchSides$(A, B, E(G), I_0)$
$\langle I_0 = I'_0, \ldots, I'_{k_2} \rangle \leftarrow$ SwitchSides$(B, A, E(G), I_0)$
return $\langle I_0, \ldots, I_{k_1}, I_{k_1-1}, \ldots, I_0, I'_1, \ldots, I'_{k_2}, I'_{k_2-1}, \ldots, I_0 \rangle$

Figure 3.1: Algorithm to find rigid tokens.

3.2.1 Preparation and execution of I+

After we've shown that I and J share the same rigid tokens, we continue to prepare for the I+ Algorithm.
If there were rigid tokens, we form a new graph G', which is the original graph G with the rigid tokens and their immediate neighbours removed. From there on out, we find the independent sets I' and J', which only have tokens that can be found in G'. In short, we built an unlocked graph and probably disconnected graph.

After the last step, or if the sets were unlocked to begin with, we continue to finding the results of the I+ algorithm for both input independent sets I and J. From Epstein Paper: We find a reconfiguration sequence between I and I+ using dynamic programming over vertex index with a table T[·]. For notational convenience, we define $J^{i,k} = \{v_j \in J | i \leq j \leq k\}$ for any independent set J. Let G_i be the unique component of G N[v_i] containing vertices of higher index. T[i] will be assigned some J = arg $\max_{J \in [I]: v_i \in J} |J_{0,i}|$. From this definition we learn that the tokens between I and J must have the same distribution between connected components, else we can not build this other J correctly. This opens up an argument for a possible linear time approach for this step in the workflow. As a base case, set T[0] = I.

$$Define: W(i,j) = \begin{cases} T[j] & \text{if } v_i \in T[j] \\ T[j]^{0,k} \cup (T[j]^{j+1,n})_{v_i}^{G_j} & \text{if } R(G_j, T[j]^{j+1,n}) = \emptyset \\ \text{invalid} & \text{otherwise} \end{cases}$$

We say that W(i,j) is valid if 0 ≤ i ≤ j. W(i,j) also must be an independent set and not invalid. Among the valid W(i,j) that maximize $|W(i,j)^{0,i}|$, set T[i] to be the W(i,j) where j is minimal. Given T, find the minimal index i where $|T[i]^{0,i}| = |\ I\ |$. This is our I+, found in polynomial time.

We use this algorithm on both I' and J', (or I and J if the sets were initially unlocked). If I+ and J+ have differing components, then we can't have a reconfiguration sequence between them. Else, there exists a reconfiguration sequence between I and J.

3.3 Finding Reconfiguration sequences

Lastly, after proving that there is in fact a reconfiguration sequence between I and J, we want to return one of the possible reconfiguration sequences between I and J. To do so, we use Breadth First Search to find the shortest possible paths between the tokens of I and J sharing the same index i respectively. After these paths have been found, we create the independent set reconfiguration sequence step by step, making sure that every step is in fact an independent set by ensuring that no slide step is taken that steps into another tokens neighbourhood.

4

Python Implementation

In this section, we talk about the before mentioned algorithms, implemented in python. We go over specialities in the code, as well as interconnections and possible changes.We will not go through every single method in this section, just the broad workflow.

4.1 UML-Diagram

A short overview over the different classes that were created for the programming part. The classes were set up in a way that makes sense considering the steps in the workflow.

4.2 Using the Code

In this section we will talk about the utilisation of the project code. To begin, you'll want to create a Graph and two independent sets and put them all in the same folder a level above the code, where your structure should be that the Graph comes first, followed by the independent set files.

The layouts for the text files are specified as follows:

1. Graphs: Each line consists of one edge, which consists of the indexes of the vertices connected by the edge divided by one space.

2. Independent sets: Each line consists of the index of one token.

By ensuring that the files are formatted accordingly you take the first step in correctly using the code. Next one can execute the code using the following command line argument: python SlidingToken.py "path-to-folder". If all files are present, the code will start running, else an error message will appear.

If everything went correctly, we will enter the workflow.

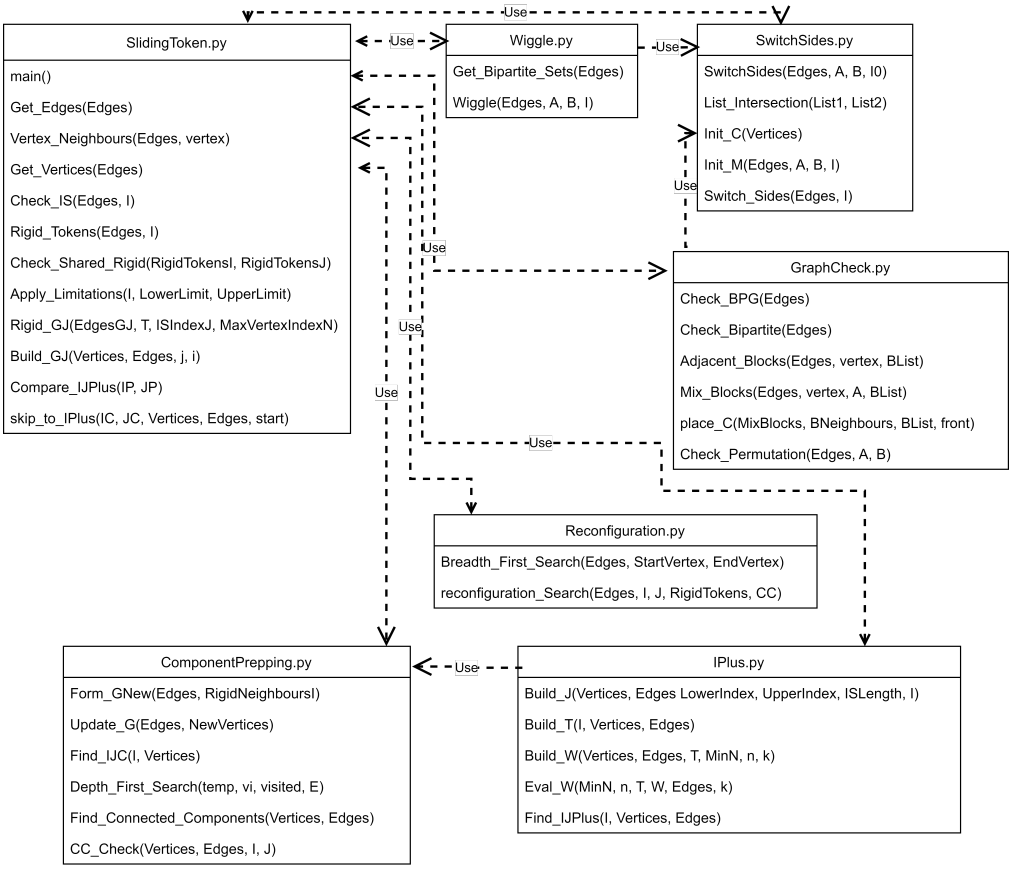

Figure 4.1: UML-Diagram Code-Overview

4.3 SlidingToken.main()

This method is our main director for the entire workflow of our program. As seen in the UML file above, SlidingToken.py includes the most methods of all written classes, due to checks needed before entering into the logic of other classes.

To put the workflow broadly together:

1. Read in command line arguments
2. Check validity of said arguments
3. Start BPG Check
4. Start Rigid Token Check and evaluate it
5. Start IPlus and evaluate it
6. Start Reconfiguration Search.

We are interested in the execution of the main steps of the before hand discussed algorithms.

4.4 BPG Check

This code can be found in the class GraphCheck.py. As talked about in the algorithm section, we divide the BPG check into a check for bipartite recognition and permutation recognition. These checks are worked through consecutively because, as mentioned in 3.1.1, the permutation recognition algorithm requires the input graph to already be bipartite. Both checks need to pass or the algorithm stops, since the rest of the workflow depends on the input graph to be a BPG.

4.4.1 Bipartite Check

The bipartite check found in GraphCheck.Check_Bipartite is a 1:1 python implementation of the function found in 3.1.1.1. A legacy version of this code without the Boolean value can be found in Wiggle.py

```
# Input: Graph Edges.
# Output: True if G is Bipartite Graph, else false.
def Check_Bipartite(Edges):
    A = []
    B = []
    EdgeList = SlidingToken.Get_Edges(Edges)
    for e in EdgeList:
        if e[0] in A and e[1] in B or e[1] in A and e[0] in B:
            continue
        elif not e[0] in A and not e[0] in B and not e[1] in A and not e[1] in B:
            A.append(e[0])
            B.append(e[1])
        elif e[0] in A and not e[0] in B and not e[1] in A and not e[1] in B:
            B.append(e[1])
        elif not e[0] in A and e[0] in B and not e[1] in A and not e[1] in B:
            A.append(e[1])
        elif not e[0] in A and not e[0] in B and e[1] in A and not e[1] in B:
            B.append(e[0])
        elif not e[0] in A and not e[0] in B and not e[1] in A and e[1] in B:
            A.append(e[0])
        else:
            print("Graph is not Bipartite!")
            return False, A, B
    return True, A, B
```

Figure 4.2: Bipartite Check as found in GraphCheck.py

If this check fails, we leave the workflow with the message seen in fig 3.4:

4.4.2 Permutation Check

The permutation check found in GraphCheck.Check_Permutation() is a python interpretation of the algorithm found in 3.1.1.2. This algorithm needed a few helper functions. Adjacent_Blocks() to build ADJ(v), Mix_Blocks() to build MIX(v) and finally place_C(), a function that helps put a block C into the right position in the block List.

```
C:\Users\pstie\Desktop\Uni\6. FS22\BA\Code>SlidingToken.py Test1
Graph G and independent sets I and J from input.
Edges of G: ['1 2', '1 3', '2 3']
Independent set I: ['1']
Independent set J: ['2']
Check if Graph G is BPG.
Graph is not Bipartite!
G is not BPG!
Runtime: 0.010155677795410156
```

Figure 4.3: Command line output after bipartite test fails.

We input the edges of the graph and the bipartite sets A and B that we calculated before in 3.4.2.1. The program then follows the algorithm described in 3.1.1.2 to decide whether the input is a permutation graph or not. If this check fails, we leave the workflow with the message seen in figure 3.5:

```
C:\Users\pstie\Desktop\Uni\6. FS22\BA\Code>SlidingToken.py Test2
Graph G and independent sets I and J from input.
Edges of G: ['1 2', '2 3', '3 4', '4 5', '5 6', '6 1']
Independent set I: ['1', '6']
Independent set J: ['2', '5']
Check if Graph G is BPG.
G is Bipartite.
Set A: ['1', '3', '5']
Set B: ['2', '4', '6']
Graph is not permutation Graph!
G is not BPG!
Runtime: 0.00800180435180664
```

Figure 4.4: Command line output after permutation test fails.

4.5 Rigid Tokens

The rigid token algorithms have been implemented as interpretations of the pseudo code described in chapter 3.1.2 and figure 3.1. Algorithm 1 can be found as SwitchSides.py and algorithm 2 as Wiggle.py. They work according to the interpretation given in 3.1.2, without creating all possible independent sets in the equivalence classes of I and J. The evaluation of the rigid token results for both Independent sets need to be exactly the same or else the algorithm fails.

4.5.1 Preparing for I+

After we found all of the rigid tokens, we need to prepare our data for I+. This is done with arbitrary list manipulations found in ComponentPreppring.py, where if there were any rigid tokens, the indices of those rigid tokens and their neighbours, found by using SlidingToken.Vertex_Neighbours(), are removed from all edge and vertex lists containing them, leaving us with unlocked sets to continue our work with.

In the class ComponentPrepping.py, we also include a function to find the connected components of the graph used in the further workflow. This will become important later on.

4.6 I+

After our data has been prepared, we can start with finding I+. This is a complicated process, as can be gleaned from 3.1.3, for which we need a lot of helper functions. In IPlus.py we start out in the function Find_IJPlus(). We immediately follow into Build_T, the dynamic programming table also found in 3.1.3, which further sends us to Build_J(), in which the elements for T are going to be built as proposed in 3.1.3. We used Depth-First-Search [4] to ensure the tokens of J are in the same equivalence class as the input independent set by ensuring each connected component of the graph has the same amount of tokens in J as in the original independent set.
After T has been built, we send it further to the function Build_W() that follows the case differentiation given in 3.1.3 until completion and finally, the table we receive from there is being processed in Eval_W(). After we have done this entire process for both remainders of the independent sets we compare their results, which must be equal or there can not be a reconfiguration sequence.

4.7 Breadth First search for Reconfiguration sequences

Onto the final part of our workflow. If we have arrived at this point, we have proven that there is a reconfiguration sequence possible in polynomial time. So the last step to add is to give out such a sequence. To do so, we take the tokens pairwise and run them on the remaining edges over Breadth_First_Search() [2] to find the shortest possible path between the tokens of I and J.
After we've found these paths, we must ensure that every single step taken is not in conflict with our definition of Independent sets, i.e. you can not move a token to a point where another tokens neighbour resides on the graph.
With this step done we have finished the implementation of our workflow completely. Next we're going to have a look at some interesting examples used for testing.

5
Examples

In this last section of our main part we'll show a short overview of the most intriguing cases for our workflow, with examples and graphs. We assume that the case "No rigid tokens without reconfiguration sequence" does not exist, as we have not found examples for this specific case. This assumption will be discussed further in section 4.1.1.

5.1 Rigid tokens with reconfiguration sequence

Our first example is going to be a graph with independent sets that share the same rigid tokens and are able to be reconfigured into each other. On figure 3.6 you can see the graph including the independent sets for test 3, with red tokens belonging to set I and blue tokens belonging to set J, violet ones belonging to both. On figure 3.7 you'll find the command line output which incidentally is our biggest flow of operations possible for our code.

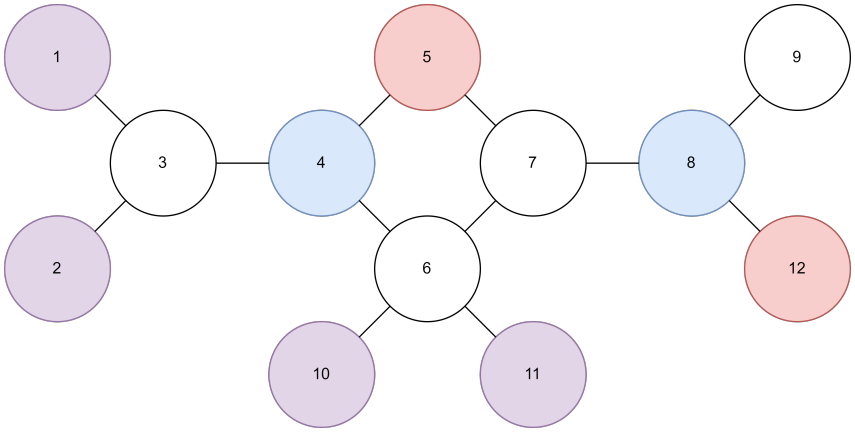

Figure 5.1: Test3 visualized.

```
C:\Users\pstie\Desktop\Uni\6. FS22\BA\Code>SlidingToken.py Test3
Graph G and independent sets I and J from input.
Edges of G: ['1 3', '2 3', '3 4', '4 5', '4 6', '5 7', '6 7', '6 10', '6 11', '7 8', '8 9', '8 12']
Independent set I: ['1', '2', '4', '8', '10', '11']
Independent set J: ['1', '2', '5', '10', '11', '12']
Check if Graph G is BPG.
G is Bipartite.
Set A: ['1', '2', '4', '7', '10', '11', '9', '12']
Set B: ['3', '5', '6', '8']
Graph is permutation Graph.
Check if I and J are Independent sets for G.
I and J are independent sets for G.
Find rigid tokens of I and J.
There are rigid tokens, continue normally.
Rigid tokens for I: ['1', '2', '11', '10']
Rigid tokens for J: ['1', '2', '11', '10']
I and J share the same rigid tokens.
Find the neighbours of the rigid tokens in I.
Rigid tokens including neighbours: ['1', '3', '10', '6', '11', '2']
Form a new graph G' which is G without the rigid tokens and their neighbours.
New Vertices: ['4', '5', '7', '8', '9', '12']
G' Edges ['4 5', '5 7', '7 8', '8 9', '8 12']
Find intersection of components of G' with I and J, forming I' and J'.
Intersection with I: ['4', '8']
Intersection with J: ['5', '12']
Expand I' and J' to I'+ and J'+.
I'+: [['4'], ['4'], ['5'], ['7'], ['9']]
J'+: [['4'], ['4'], ['5'], ['7'], ['9']]
Compare I'+ with J'+.
An independent set reconfiguration sequence exists between I and J under G!
Find reconfiguration sequence using search algorithm.
A possible reconfiguration sequence from I to J:
['1', '2', '4', '8', '10', '11']
['1', '2', '5', '8', '10', '11']
['1', '2', '5', '10', '11', '12']
Runtime: 0.012035131454467773
```

Figure 5.2: Command line output for Test 3.

5.2 No rigid tokens with reconfiguration sequence

Our second example is going to be one without rigid tokens, the same colour distribution as before applies for figure 3.8.

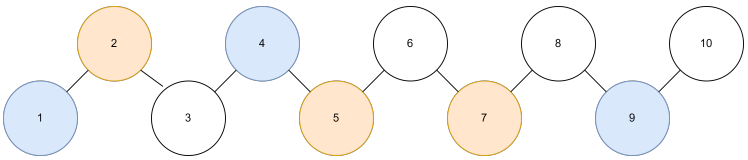

Figure 5.3: Test 4 visualized.

```
C:\Users\pstie\Desktop\Uni\6. FS22\BA\Code>SlidingToken.py Test4
Graph G and independent sets I and J from input.
Edges of G: ['1 2', '2 3', '3 4', '4 5', '5 6', '6 7', '7 8', '8 9', '9 10']
Independent set I: ['2', '5', '7']
Independent set J: ['1', '4', '9']
Check if Graph G is BPG.
G is Bipartite.
Set A: ['1', '3', '5', '7', '9']
Set B: ['2', '4', '6', '8', '10']
Graph is permutation Graph.
Check if I and J are Independent sets for G.
I and J are independent sets for G.
Find rigid tokens of I and J.
There are no rigid tokens, the sets are unlocked.
Expand I' and J' to I'+ and J'+.
I'+: [['3', '1'], ['4', '2'], ['5', '3'], ['6', '4'], ['7', '5'], ['8', '6']]
J'+: [['3', '1'], ['4', '2'], ['5', '3'], ['6', '4'], ['7', '5'], ['8', '6']]
Compare I'+ with J'+.
An independent set reconfiguration sequence exists between I and J under G!
A possible reconfiguration sequence from I to J:
['2', '5', '7']
['1', '5', '7']
['1', '4', '7']
['1', '4', '8']
['1', '4', '9']
Runtime: 0.03787994384765625
```

Figure 5.4: Command line output for Test 4.

5.3 Rigid tokens without reconfiguration sequence

The final example we're going to show here. Same colour rules apply as in the previous examples, with independent set I = [1,5,6,9] and set J = [5,6,8,10]. This example is interesting because it passes every check except of course the I+ check due to the independent sets not being reconfigurable. We can be that the tokens 5 and 6 are rigid, they are blocking their only possible step for each other in vertex 4. Removing the rigid tokens and their neighbour leaves us with two connected components, 1-2-3 and 7-8-9-10-11. Now the reason for the sets not being reconfigurable into each other is that there is no legal step from connected component 1 to connected component 2, but set I has a token in each component whilst set J has no tokens in connected component 1 and two tokens in connected component 2, meaning there's no legal step to get one of the tokens from connected component 2 into connected component 1 for J, which renders our sets not reconfigurable into each other.

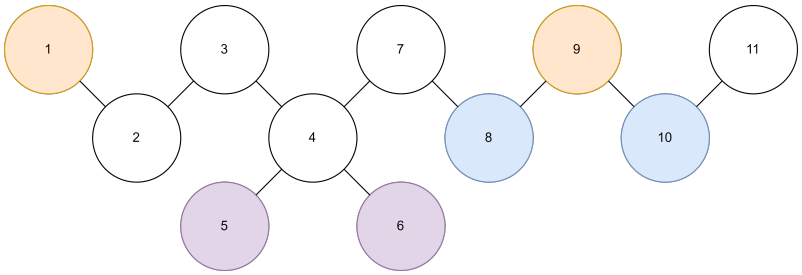

Figure 5.5: Test 5 visualized.

```
C:\Users\pstie\Desktop\Uni\6. FS22\BA\Code>SlidingToken.py Test5
Graph G and independent sets I and J from input.
Edges of G: ['1 2', '2 3', '3 4', '4 5', '4 6', '4 7', '7 8', '8 9', '9 10', '10 11']
Independent set I: ['1', '5', '6', '9']
Independent set J: ['5', '6', '8', '10']
Check if Graph G is BPG.
G is Bipartite.
Set A: ['1', '3', '5', '6', '7', '9', '11']
Set B: ['2', '4', '8', '10']
Graph is permutation Graph.
Check if I and J are Independent sets for G.
I and J are independent sets for G.
Find rigid tokens of I and J.
There are rigid tokens, continue normally.
Rigid tokens for I: ['6', '5']
Rigid tokens for J: ['6', '5']
I and J share the same rigid tokens.
Find the neighbours of the rigid tokens in I.
Rigid tokens including neighbours: ['5', '4', '6']
Form a new graph G' which is G without the rigid tokens and their neighbours.
New Vertices: ['1', '2', '3', '7', '8', '9', '10', '11']
G' Edges ['1 2', '2 3', '7 8', '8 9', '9 10', '10 11']
Find intersection of components of G' with I and J, forming I' and J'.
Intersection with I: ['1', '9']
Intersection with J: ['8', '10']
Expand I' and J' to I'+ and J'+.
I'+: [['3', '7'], ['3', '8'], ['3', '9'], ['3', '10']]
J'+: [['1'], ['1'], ['2'], ['3'], ['7'], ['8'], ['9']]
Compare I'+ with J'+.
There can not be an independent set reconfiguration sequence between I and J under G!
Runtime: 0.02548360824584961
```

Figure 5.6: Command line output for "Test 5", rigid tokens with no reconfiguration sequence.

6 Discussion

6.1 Results

The entire workflow was implemented as planned and works as intended in polynomial time. A possible linear time approach was deduced for the main algorithm that recognizes if two independent sets could be reconfigured into each other.

6.1.1 Possible alternative to I+

The idea stems from I+ (3.1.3) itself, where one must build independent sets J according to a set of rules, including that this J must be in the equivalence class of the given independent set I. To be in the equivalence class of I, J must be able to be reconfigured into I [3]. From an initially wrong implementation for the creation of J the conclusion was drawn that to be reconfigurable, the tokens of the independent sets I and J must respectively be present in the same connected components.

The only part that sets the original algorithm into polynomial time complexity is the I+ part. Said part can hypothetically be replaced by an algorithm that compares if the tokens of each independent set are located in the same connected components. If that is not the case, there may not be a reconfiguration sequence between the two sets, since there exists no legal step between two disconnected components for our algorithm.

This entire hypothesis hangs on the claim that there must always be a reconfiguration sequence between unlocked sets in BPG (unproven). If this were the case, then the workflow from BPG recognition up to reconfiguration recognition could be done in linear time using the connected component approach as substitute for the I+ algorithm. Only the actual reconfiguration search would remain in polynomial time due to the use of breadth-first-search.

This connected component variant to I+ has also been implemented in the code and can be called upon by adding "CC" to the command line argument. Experimentally, the CC-Variant is definitely always faster than the original variant which can be seen in figure 4.1, which is the same test as 3.3.2 as CC-Variant. It is important to note that the CC-variant

outputs the same results as the original I+ algorithm.

```
C:\Users\pstie\Desktop\Uni\6. FS22\BA\Code>python SlidingToken.py Test4 CC
here
Graph G and independent sets I and J from input.
Edges of G: ['1 2', '2 3', '3 4', '4 5', '5 6', '6 7', '7 8', '8 9', '9 10']
Independent set I: ['2', '5', '7']
Independent set J: ['1', '4', '9']
Check if Graph G is BPG.
G is Bipartite.
Set A: ['1', '3', '5', '7', '9']
Set B: ['2', '4', '6', '8', '10']
Graph is permutation Graph.
Check if I and J are Independent sets for G.
I and J are independent sets for G.
Find rigid tokens of I and J.
There are no rigid tokens, the sets are unlocked.
Linear approach using Token Distribution on Connected Components.
I and J, tokens per component: {0: 3} {0: 3}
An independent set reconfiguration sequence exists between I and J under G!
A possible reconfiguration sequence from I to J:
['2', '5', '7']
['1', '5', '7']
['1', '4', '7']
['1', '4', '8']
['1', '4', '9']
Runtime: 0.014043092727661133
```

Figure 6.1: CC variant of Test 4.

6.2 Related Questions

An important question that came up in our work was why do we use BPG, and if our algorithm could work on graphs that are not BPG. The checks for BPG at the beginning aside, our initial problem, the reconfiguration problem, is a NP-complete problem. With BPG we have good algorithmic properties with which we can solve several NP-Complete problems in polynomial time, including the reconfiguration problem [6].

6.3 Further Work

Despite ever important part of the algorithm being implemented, some software engineering aspects were not implemented. One example of this is that there is no check for the correctness of the input data from the command line arguments, as long as all three files are present the algorithm will attempt to run. This could cause problems down the line for the algorithm due to data mishaps.

More work could be put into the CC-variant. Experimentally sound, it is missing some proof

basics. The most important one being the proof that there should always be an independent set reconfiguration sequence between unlocked sets. Doing so would lay a great foundation for a recognition algorithm for independent set reconfiguration in linear time.

Finally, more testing could be done, using bigger, more complex graphs. The ones used in this paper are all designed to test certain characteristics of the entire program.

Bibliography

[1] Łukasz Bożżyk, Novotná Krawczyk, Tomasz, and Karolina Orkasa. Vertex deletion into bipartite permutation graphs. *Algorithmica*, 2022.

[2] One Step! Code. Shortest path in python (breadth first search), Mar 2022. URL https://onestepcode.com/graph-shortest-path-python/?utm_source=rss&utm_medium=rss&utm_campaign=graph-shortest-path-python.

[3] Eli Fox-Epstein, Duc A. Hoang, Yota Otachi, and Ryuhei Uehara. Sliding token on bipartite permutation graphs. *Springer*, 2015.

[4] GeeksforGeeks. Connected components in an undirected graph, Apr 2022. URL https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/.

[5] Marcin Kamiński, Paul Medvedev, and Martin Milanič. Complexity of independet set reconfigurability problems. *Theoretical Computer science*, 2022.

[6] Jeremy Spinrad, Anreas Brandstaedt, and Lorna Stewart. Bipartite permutation graphs. *Discrete Applied Mathematics 18*, 1987.

A
Appendix

A.0.1 Code

The code and all files used for testing can be found under:
https://github.com/PatrickSteiner311/SlidingToken-Implementation