University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

An Efficient Algorithm for Rainbow Hamiltonian Cycles

Nathan Luiz, Willian Mori

Final Essay

mac 499 — Capstone Project

Supervisor: Yoshiko Wakabayashi

São Paulo
2025

Acknowledgment

We would like to express our deep gratitude to our advisor, Yoshiko Wakabayashi, for accepting to guide us in this project. Her patience and guidance were essential for the development of this work.

We also thank Gabriel Morete, for suggesting the theme of this monograph.

Resumo

Nathan Luiz, Willian Mori. **Um Algoritmo Eficiente para Circuitos Hamiltonianos Rainbow**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2025.

Seja $n \geq 3$ e $G = G_1 \cup G_2 \cup ... \cup G_n$ um grafo que é a união de n grafos simples dois a dois aresta-disjuntos G_i de ordem n, todos definidos sobre um mesmo conjunto de vértices, cada qual com arestas monocromaticamente coloridas mas coletivamente usando n cores distintas. Joos and Kim (2020) provou que se cada G_i satisfaz a condição de Dirac (i.e. tem grau mínimo pelo menos $n/2$), então G tem um circuito Hamiltonian rainbow (um circuito em que todas as arestas têm cores distintas). Nesta monografia apresentamos uma versão algoritmica dessa prova, e explicamos passo a passo um algoritmo que constrói um circuito Hamiltoniano rainbow em G. Apresentamos um pseudocódigo para cada procedimento que é descrito, detalhando as estruturas que são usadas e analisando sua complexidade computacional. Mostramos que o algoritmo que implementamos tem complexidade $O(n^3)$, assintoticamente a melhor complexidade possível, pois o grafo de entrada G tem $O(n^3)$ arestas. Adicionalmente, incluímos neste trabalho uma animação gráfica que criamos para ilustrar o processo de construção de um circuito Hamiltoniano rainbow no grafo G.

Palavras-chave: Teorema de Dirac. Versão Rainbow. Circuito Hamiltoniano. Implementação.

Abstract

Nathan Luiz, Willian Mori. **An Efficient Algorithm for Rainbow Hamiltonian Cycles**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

Let $n \geq 3$ and $G = G_1 \cup G_2 \cup \ldots \cup G_n$ be a graph that is the union of n pairwise edge-disjoint simple graphs G_i of order n, all defined on a same vertex set, each one monochromatically edge colored but collectively using n distinct colors. Joos and Kim (2020) proved that if each G_i satisfies Dirac's condition (i.e. has minimum degree at least $n/2$), then G has a rainbow Hamiltonian cycle (a cycle in which all edges have disctinct colors). In this monograph, we present an algorithmic version of this proof, and explain step by step an algorithm that builds a rainbow Hamiltonian cycle in G. We present a pseudocode for each procedure that is described, providing details on the structures that are used and analysing its computational complexity. We show that the algorithm that we implemented has time complexity $O(n^3)$, asymptotically the best possible, as the input graph G has $O(n^3)$ edges. Additionally, we include in this work a graphical animation that we created to illustrate the process of building a rainbow Hamiltonian cycle in G.

Keywords: Dirac Theorem. Rainbow version. Hamiltonian Cycle. Implementation.

Contents

Introduction

The problem we address in this monograph is relatively recent, although its origin traces back to classical concepts. In 1978, Caccetta and Häggkvist (Nathanson, 2006) formulated the conjecture that every simple digraph of order n with minimum outdegree d has a directed cycle of length at most $\lceil n/d \rceil$. This conjecture marked the beginning of investigations of short cycles in digraphs with degree constraints.

Nearly four decades later, in 2017 Ron Aharoni, from the Department of Mathematics at Technion, proposed a stronger version of this conjecture, known as a rainbow version (Clinch *et al.*, 2022). His conjecture states that, given a graph G of order n with edges colored in n colors, if each color appears in at most r edges, then G has a rainbow cycle of length at most $\lceil n/r \rceil$.

In 2019, Felix Joos, from the University of Heidelberg, and Jaehoon Kim, from KAIST, proved the existence of a rainbow Hamiltonian cycle in a graph G that results from the union of n edge-disjoint graphs G_i $(1 \leq i \leq n)$ defined on a same set of n vertices, each G_i monochromatically edge colored but collectively using n distinct colors, and each one satisfying Dirac's condition. A rainbow cycle is a cycle in which all edges are colored differently. This result gave strong support to Aharoni's conjecture.

The proof presented by Joos and Kim uses simple and elegant (yet non-trivial) techniques that enabled the development of an $O(n^3)$ algorithm in the number of vertices. They also extended their result proving the existence of a perfect rainbow matching.

More recently, in 2023 Liqing Gao and Jian Wang (Gao and Wang, 2023) proved the existence of a rainbow Hamiltonian cycle in graphs that satisfy a condition stated in a theorem (also known as Ore's theorem, but different from the one we mention in Chapter 3) using the "shifting operator" tool. This technique, developed by Erdös, Ko, and Rado, is widely used in extremal set theory and has led to significant advances in solving problems in this area. However, we will not cover this work here.

This monograph is structured as follows. In Chapter 1, we present some basic definitions, then we present the well-known Dirac's theorem obtained in 1952 (Dirac, 1952), and state the theorem known as the *rainbow version of Dirac's theorem*, the central topic of this monograph. In Chapter 2, we present a pseudocode and the details of the implementation of the algorithm based on the work of Joos and Kim (Joos and Kim, 2020). In Chapter 3, we present a statement which we call the *rainbow version of Ore's theorem*, which is analogous to the rainbow version of Dirac's theorem, but is based on a (weaker) sufficient condition for a graph to be Hamiltonian proved by Ore in 1960. We conjecture that this statement on

the existence of a rainbow Hamiltonian cycle in the union graph G is true. If so, this result would generalize the rainbow version of Dirac's theorem. We show a partial result that we have obtained (so far) for this statement: that the union graph G, of order n, has a rainbow Hamiltonian path and also a rainbow cycle of length $n - 1$. Finally, in Chapter 4, we make some final considerations about the work we have developed, including a graphical visualizer for our implementation.

The source codes developed in this project are available in GitHub. There is a code written in C++, using the Boost library, and also a code in Python, as it supports an animation using the Graph-Tool framework.

Chapter 1

Preliminary

In this chapter, we present the concepts and theorems that are fundamental to understand the work developed in this monograph. We start by presenting and proving Dirac's theorem (Dirac, 1952), and then we formally present the rainbow version of this theorem, the main topic of this work.

We assume that the reader is familiar with some basic concepts of graph theory. We present some of them to estabilish the terminology and notation, which follow those used in Bondy and Murty, 2008.

1.1 Definitions

We denote by $G = (V, E)$ a graph G with vertex set V and edge set E.

A path P of size ℓ in a graph $G = (V, E)$ is a sequence of vertices and edges $(v_0, e_0, v_1, e_1, \dots, v_{\ell-1}, e_{\ell-1}, v_\ell)$ such that $v_i \in V$, $e_i = \{v_i, v_{i+1}\} \in E$, and $v_i \neq v_j$ for $0 \leq i < j \leq \ell$.

A cycle C of size ℓ in a graph $G = (V, E)$ is a sequence of vertices and edges $(v_0, e_0, v_1, e_1, \dots, v_\ell, e_\ell, v_{\ell+1})$ such that $v_0 = v_{\ell+1}$, $v_i \in V$, $e_i = \{v_i, v_{i+1}\} \in E$, and $v_i \neq v_j$ for $0 \leq i < j \leq \ell$.

If G is a graph with colors on the edges, a path (or cycle) in G is called rainbow if all of its edges are colored differently.

When convenient, if G is the name of a graph, we may refer to its vertex set and edge set as $V(G)$ and $E(G)$, respectively.

A Hamiltonian cycle in a graph G of order n is a cycle in G of length n. The concept of Hamiltonian path is defined analogously (it contains all vertices of the graph). A graph is Hamiltonian if it contains a Hamiltonian cycle. Deciding whether a graph is Hamiltonian is a well-known NP-complete problem. However, there are many sufficient conditions that guarantee the existence of a Hamiltonian cycle in a graph. One of them, based on the minimum degree of the graph, is given by the Dirac's theorem.

1.2 Dirac's Theorem

Theorem 1 (Dirac, 1952) *If a simple graph $G = (V, E)$ with $n \geq 3$ vertices satisfies the condition $d_G(v) \geq n/2$, for all $v \in V$, then G is Hamiltonian.*

Proof. Let $G = (V, E)$ with be a simple graph with $n \geq 3$ vertices that satisfies the condition of the theorem. Suppose, by contradiction, that G is not Hamiltonian.

Let $G' = (V, E')$ be a graph that maximizes $|E'|$ such that G' is not Hamiltonian and $E \subseteq E'$. Clearly, G' is not a complete graph, because otherwise it would be Hamiltonian. Consider a pair $x, y \in V$ such that $e = \{x, y\} \notin E'$. The graph $(V, E' + e)$ must contain a Hamiltonian cycle $C = (v_1, e_1, v_2, e_2, \ldots, v_n, e_n, v_1)$, where $v_1 = x$, $v_n = y$ and $e_n = e$, because otherwise, it would contradict the maximality of G'. Since G is a subgraph of G', $d_G(v) \leq d_{G'}(v)$ for all $v \in V$.

Let $I_1 = \{i \in \{2, 3, \ldots, n-2\} : \{x, v_{i+1}\} \in E'\}$, $I_2 = \{i \in \{2, 3, \ldots, n-2\} : \{y, v_i\} \in E'\}$. We have that $|I_1| \geq d_{G'}(x) - 1$ and $|I_2| \geq d_{G'}(y) - 1$, which implies $|I_1| + |I_2| > n - 3$. Since $|I_1| + |I_2| = |I_1 \cup I_2| + |I_1 \cap I_2|$ and $|I_1 \cup I_2| \leq n - 3$, there exists $i \in I_1 \cap I_2$.

That means that there is a cycle $(v_1, e_1, v_2, \ldots, v_i, \{v_i, v_n\}, v_n, e_{n-1}, v_{n-1} \ldots, v_{i+1}, \{v_{i+1}, v_1\}, v_1)$, which is Hamiltonian and is contained in G', a contradiction. Thus, G' does not exist, and therefore G must be Hamiltonian.

□

1.3 Rainbow version of Dirac's Theorem

We say that a graph of order $n \geq 3$ satisfies *Dirac's condition* if it is simple and each vertex of this graph has degree at least $n/2$.

The theorem which will be the central topic of this monograph is the following one, known as the *rainbow version of Dirac's theorem*. It was proved by Joos and Kim, 2020.

Theorem 2 (Joos and Kim, 2020) *Let $n \geq 3$ and $G = G_1 \cup G_2 \cup \ldots \cup G_n$ be a graph that is the union of n pairwise edge-disjoint graphs G_i of order n, all defined on a same vertex set, each one monochromatically edge colored but collectively using n distinct colors. If each G_i satisfies Dirac's condition, then G has a rainbow Hamiltonian cycle.*

Chapter 2

Algorithmic Approach

In this chapter we present our implementation of the algorithm to find a Hamiltonian cycle based on the proof of Joos and Kim, 2020. We provide the corresponding pseudocode and details of the implementation.

2.1 Definitions, Notation and Conventions

Throughout this chapter, we use some definitions and conventions to make the code simpler.

The total number of vertices is denoted by n, and the graph that is the union of n graphs is represented by $G = G_1 \cup G_2 \cup \cdots \cup G_n$, where each graph G_i satisfies Dirac's condition. Moreover, we use the following notation:

- $d(c, v)$ stands for the degree of vertex v in graph G_c;
- $V(X)$ and $A(X)$ represents the sets of vertices and edges of object X, respectively;
- $N_J(X)$ denotes the set of neighbors of X on graph J.

We use the following conventions:

- Some variables used on a pseudocode may be defined on other parts of the same function;
- The variables on the images are the same as the ones defined in the proof and the code;
- Colors c_x in the proof may be represented as $c(x)$ in the images.

The pseudocode is written using Python methods and conventions, such as array slicing and list appending.

2.1.1 Structures and functions abstractions

We define first some abstract structures and functions.

Structure of $edge(u, v, c)$

Each edge in G has three attributes:

- u, v: vertices connected by the edge;
- c: color of the edge, which also indicates that the edge belongs to graph G_c.

Structure of $Path$

Each path contains two dynamic arrays:

- $vertices$: an array of $vertex$;
- $edges$: an array of $edge$.

If the $Path$ is not empty, it is guaranteed that size of $vertices$ is equal to size of $edges$ plus one. We also added the constraint that a $Path$ may not contain repeated vertices or edge colors.

This structure has the following methods:

- `pop_back()`: removes the last vertex and edge from the path;
- `back()`: returns the last vertex of the path;
- `size()`: returns the size of the path.

Structure of $Cycle$

Same Structure as $Path$. The only difference is that a cycle contains at least three vertices, and $vertices$ length is equal to $edges$ length.

Also, this structure has only one method, `size()`, which returns the size of the cycle.

Function $check\_edge(G, u, v, c)$

This function takes four parameters:

- G: the collection of graphs;
- u and v: vertices of the edge;
- c: color of the edge.

If the edge $\{u, v\}$ belongs to G_c, the function returns this edge. Otherwise, it returns $None$. The function operates in constant time $O(1)$ since we can implement it using a three-dimensional matrix, which requires $O(n^3)$ memory.

2.2 Flowchart

The algorithm begins with a single object representing an empty path. At each iteration, the algorithm processes this object, which may be either a path or a cycle of length ℓ, and transforms it into a new object. The result may be a cycle of length ℓ, a cycle of

length $\ell + 1$, or a path of length $\ell + 1$. In Figure 2.1, an L-Path is a path of length L and an L-cycle is a cycle of length L.

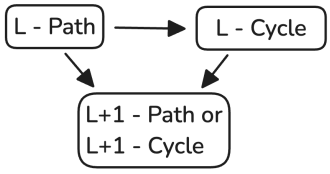

Figure 2.1: *Flowchart of the algorithm*

We will assume that the algorithm is processing a graph of order $n \geq 6$. When $n < 6$ the result is known to hold and can be solved by brute force. In Figure 2.2 we show the possible cases that may occur in the algorithm.

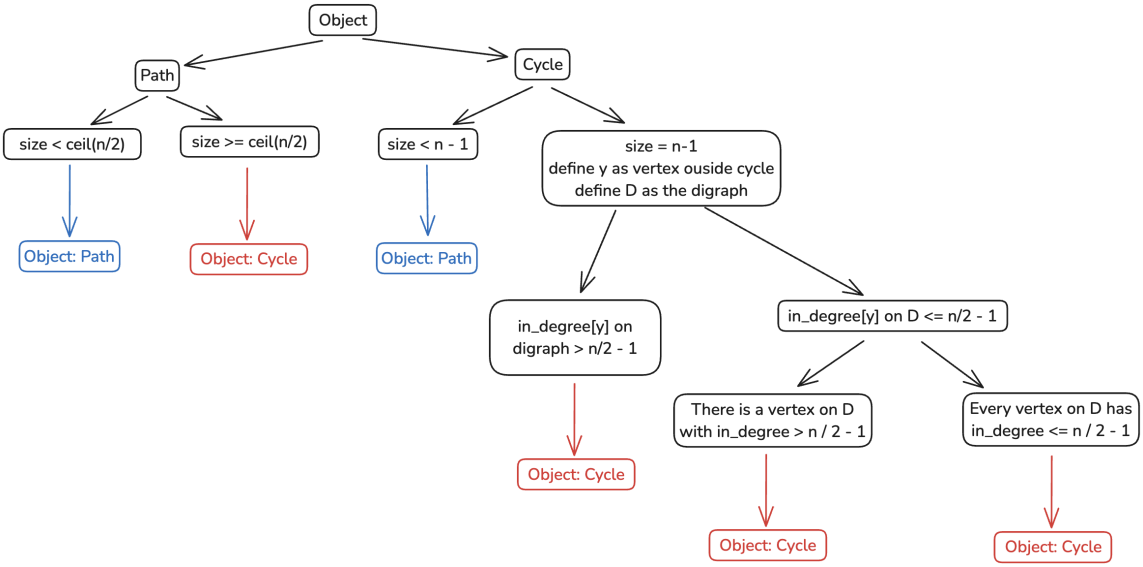

Figure 2.2: *The possible cases that may occur in the algorithm*

2.2.1 Path of length ℓ

In this case, we start with a path $P = (x_0, e_0, \ldots, x_{\ell-1}, e_{\ell-1}, x_\ell)$ of length ℓ. To assist in this process, we define the following variables:

- `colors_in_path`: an array of size n in which `colors_in_path[i]` is `True` if color i is used in the edges of the path.
- `vertices_in_path`: an array of size n in which `vertices_in_path[i]` is `True` if vertex i is included in the path.

Let us divide the proof into two cases: when $\ell \geq \left\lceil \frac{n}{2} \right\rceil$ and when $\ell < \left\lceil \frac{n}{2} \right\rceil$.

Case 1: $\ell < \left\lceil \frac{n}{2} \right\rceil$

In this case, we select a color c that is not present in the edges of the current path P. Since $d(c, x_\ell) \geq \left\lceil \frac{n}{2} \right\rceil$, and there are at most $\ell < \left\lceil \frac{n}{2} \right\rceil$ vertices other than x_ℓ in the path, there exists a vertex y outside the path P that is adjacent to x_ℓ via an edge in $A(G_c)$. This guarantees that we can extend the path P to a longer path by adding the edge $\{x_\ell, y\}$.

The code below implements this algorithm:

Algorithm 1 Path Extension for $\ell < \left\lceil \frac{n}{2} \right\rceil$

```
1:  function EXTEND_PATH_SMALL(G, P)                    ▷ P = (x_0, ..., x_{ℓ-1})
2:      color_outside_path ← next(i for i in range(n) if not colors_in_path[i])
3:      for i ∈ [0, ..., n − 1] do
4:          if not vertices_in_path[i] then
5:              edge ← check_edge(G, P.back(), i, color_outside_path)
6:              if edge ≠ None then
7:                  return Path(G, P.vertices + [i], P.edges + [edge])
8:              end if
9:          end if
10:     end for
11:     assert False, "Should not happen"
12: end function
```

The time complexity of this function is $O(n)$. Since we only need to find a color that is not on the path, and then locate a vertex outside the path that connects to the end of the path (`P.back()`) using the function *check_edge*.

Case 2: $\ell \geq \left\lceil \frac{n}{2} \right\rceil$

First, remove the last vertex (and edge) of the path P and obtain P'. Let c_y be the color of the removed edge and c_x be a color that is not on the path. We can check if the edge $\{x_0, x_{\ell-1}\}$ belongs to $A(G_{c_x})$ or to $A(G_{c_y})$. If it does, we add this edge to P' and obtain a cycle of size ℓ. This can be done with the code below:

Algorithm 2 Path Extension for $\ell \geq \lceil \frac{n}{2} \rceil$

- Part 1

```
function EXTEND_PATH_BIG(G, P)
    c_x ← P.edges[−1].color                          ▷ Color of the last edge in the path
    c_y ← next(i for i in range(n) if not colors_in_path[i])
    P.pop_back()                                      ▷ Remove the last vertex of the path
    for c ∈ [c_x, c_y] do
        edge ← check_edge(G, P.vertices[0], P.vertices[−1], c)
        if edge ≠ None then
            return Cycle(G, P.vertices, P.edges + [edge])    ▷ Return ℓ-cycle, if found
        end if
    end for
end function
```

The time complexity of this function is $O(n)$. This is the time needed to find a color not used on the path and create the object Cycle. The other operations take $O(1)$ time.

We can now check if there exists a vertex y outside the path $P = (x_0, \ldots, x_{\ell-1})$ such that y is adjacent to the vertices x_0 and $x_{\ell-1}$ and uses colors c_x and c_y, as shown in Figure 2.3:

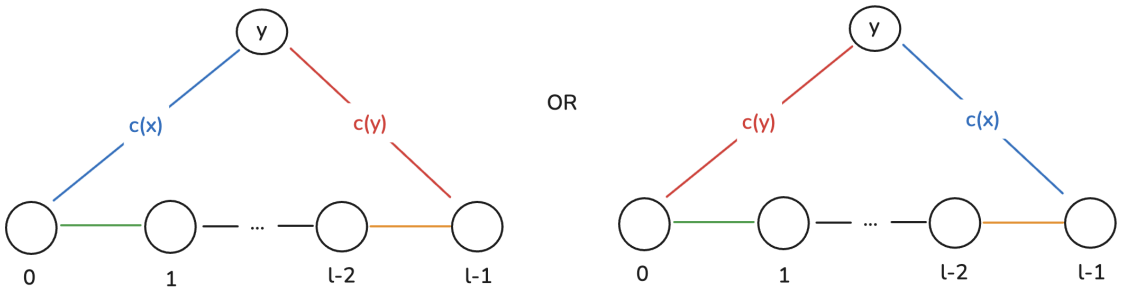

Figure 2.3: *Construction of a cycle of length $\ell + 1$*

This part can also be done in $O(n)$ time with the code below:

Algorithm 3 Path Extension for $\ell \geq \left\lceil \frac{n}{2} \right\rceil$

- Part 2

```
function EXTEND_PATH_BIG(G, P)                                   ▷ P = (x_0, ..., x_{ℓ-1})
    for [c_1, c_2] ∈ [[c_x, c_y], [c_y, c_x]] do                 ▷ Try both color pairs
        for y ∈ [0, ..., n − 1] do
            if not vertices_in_path[y] then
                edge_x ← check_edge(G, P.vertices[0], y, c_1)
                edge_y ← check_edge(G, P.vertices[−1], y, c_2)
                if (edge_x ≠ None) and (edge_y ≠ None) then
                    return Cycle(G, P.vertices + [y], P.edges + [edge_y, edge_x])   ▷ Return ℓ + 1-cycle
                end if
            end if
        end for
    end for
    assert False, "No valid extension found"
end function
```

At this point, $P = (x_0, \dots, x_{\ell-1})$. Let us define:

$$I_1 = \left\{ i \in [0, \ell-3] : \{x_0, x_{i+1}\} \in A(G_{c_x}) \right\} \text{ and } I_2 = \left\{ i \in [1, \ell-2] : \{x_i, x_{\ell-1}\} \in A(G_{c_y}) \right\}.$$

Note that, as there is no vertex $y \in V \backslash V(P)$ such that $\{x_0, y\} \in A(G_{c_x})$ and $\{y, x_{\ell-1}\} \in A(G_{c_y})$,

$$|N_{G_{c_x}}(x_0) \backslash V(P)| + |N_{G_{c_y}}(x_\ell) \backslash V(P)| \leq n - \ell,$$

.

Otherwise, by the Pigeonhole Principle, we would have found a $l + 1$ length cycle with the previous procedures. Thus:

$$|I_1| + |I_2| \geq \frac{n}{2} + \frac{n}{2} - |N_{G_c x}(x_1) \backslash V(P)| - |N_{G_c y}(x_\ell) \backslash V(P)| \geq \ell.$$

As $I_1 \cap I_2 \subseteq [0, \ell-2]$, by the Pigeonhole Principle, $I_1 \cap I_2 \neq \emptyset$. Given an element $i \in I_1 \cap I_2$, we can build a cycle with size ℓ with the following crossing procedure:

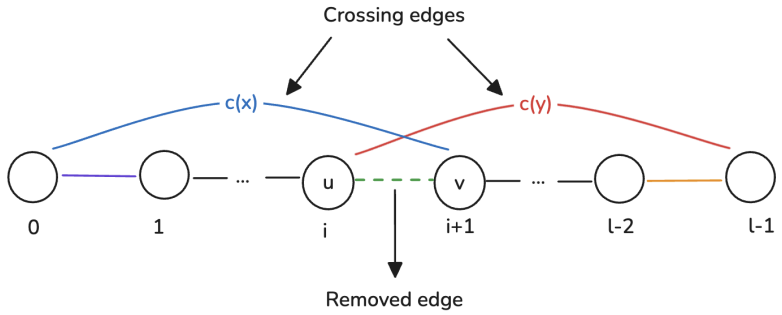

Figure 2.4: *Cycle of length $\ell + 1$ found*

Hence, in this case we just need to find an element in the intersection of I_1 and I_2 and build the desired cycle. This can also be done in $O(n)$ with the code below:

Algorithm 4 Path Extension for $\ell > \left\lceil \frac{n}{2} \right\rceil$

- Part 3

```
1:  function EXTEND_PATH_BIG(G, P)                          ▷ P = (x_0, ..., x_{ℓ-1})
2:      for i ∈ [1, ..., P.vertices.size() − 1] do
3:          u ← P.vertices[i]
4:          v ← P.vertices[i + 1]
5:          edge_x ← check_edge(G, P.vertices[0], v, c_x)
6:          edge_y ← check_edge(G, u, P.vertices[−1], c_y)
7:          if (edge_x ≠ None) and (edge_y ≠ None) then
8:              vertices ← P.vertices[ : i + 1] + [y] + P.vertices[i + 1 : −1][ : : −1]
9:              edges ← P.edges[ : i] + [edge_y] + P.edges[i + 1 : ][ : : −1] + [edge_x]
10:             return Cycle(G, vertices, edges)
11:         end if
12:     end for
13:     return None                                  ▷ Return None if no cycle found
14: end function
```

2.2.2 Cycle of length ℓ

In this case, we start with a cycle $C = (x_0, e_0, \ldots, x_{\ell-1}, e_{\ell-1}, x_\ell = x_0)$ of length ℓ. To assist the process, we define the following variables:

- `colors_in_cycle`: an array of size n in which `colors_in_cycle[i]` is `True` if color i is used in the edges of the cycle.
- `vertices_in_cycle`: an array of size n in which `vertices_in_cycle[i]` is `True` if vertex i is included in the cycle.

The values of these variables may be assigned in $O(n)$ time. We may assume that the cycle has length at least $\left\lceil \frac{n}{2} \right\rceil$. We divide the proof into two cases: when $\ell < n - 1$ and when $\ell = n - 1$.

2.2.3 Case 1: $\lceil \frac{n}{2} \rceil + 1 \leq \ell < n - 1$

Let c_1 and c_2 be any two different colors that are not in the cycle. Suppose, wlog, that there is an edge $edge(u, v, c_1) \in A(G)$ not incident to the cycle. Then, because the degree of u is at least $\lceil \frac{n}{2} \rceil$ and the length of the cycle is $\ell \geq \lceil \frac{n}{2} \rceil + 1$, there is at least one vertex w on the cycle such that $\{u, w\} \in A(G_{c_2})$.

Thus, we can build a path of length $\ell + 1$ by removing from the cycle an edge that is incident to w (see Figure 2.5):

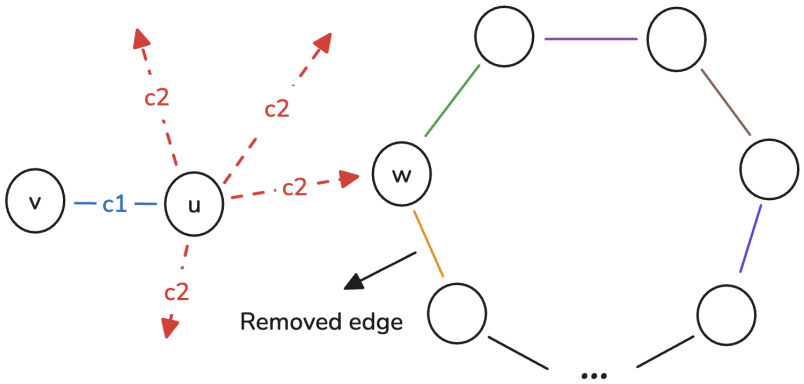

Figure 2.5: *Path of length $\ell + 1$ found*

This can be done in $O(n^2)$ time with the following code:

Algorithm 5 Cycle Extension for $\ell < n - 1$ - Part 1

```
function EXTEND_CYCLE(G, C)
    for [c_i, c_j] ∈ [[c_1, c_2], [c_2, c_1]] do                ▷ Try both color pairs
        for u ∈ [0, ..., n − 1] do
            for v ∈ [u + 1, ..., n − 1] do
                if not vertices_in_cycle[u] and not vertices_in_cycle[v] then
                    edge_u ← check_edge(G, u, v, c_i)
                    if edge_u ≠ None then
                        for k ∈ [0, ..., C.vertices.size() − 1] do
                            w ← C.vertices[k]
                            edge_w ← check_edge(G, w, u, c_j)
                            if edge_w ≠ None then
                                vertices ← [u, v] + C.vertices[k :] + C.vertices[: k]
                                edges ← [edge_u, edge_w] + C.edges[k :]
                                        +C.edges[: k]
                                return Path(G, vertices, edges)
                            end if
                        end for
                        assert False, "Should not reach here"
                    end if
                end if
            end for
        end for
    end for
    ...
end function
```

The complexity is in fact $O(n^2)$ because once we find a valid edge $edge\_u$, we just make a linear search for a valid vertex w in the cycle, that will certainly be found.

From now on, for every vertex u outside the cycle, every edge with colors c_1 or c_2 incident to u must also be incident to the cycle. Take a vertex u outside the cycle and let

$$I_1 := \left\{ i \in [0, \ell - 1] : \{u, x_{i+1}\} \in A(G_{c_1}) \right\} \text{ and } I_2 := \left\{ i \in [0, \ell - 1] : \{x_i, u\} \in A(G_{c_2}) \right\}.$$

We have that $|I_1| + |I_2| = |N_{G_{c_1}}(u)| + |N_{G_{c_2}}(u)| > \ell$. Thus, by the Pigeonhole Principle, $I_1 \cap I_2 \neq \emptyset$. In this case, we can build a cycle of length $\ell + 1$ as indicated in Figure 2.6:

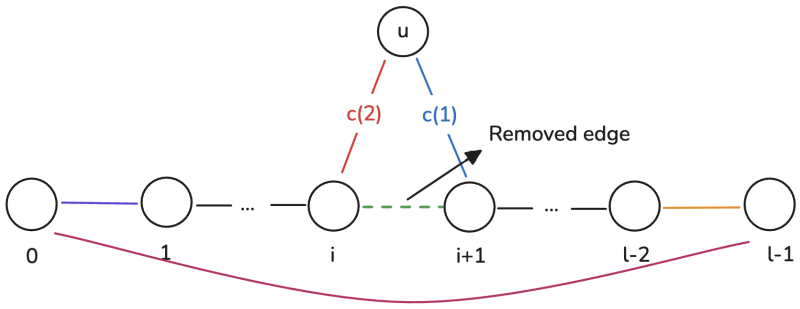

Figure 2.6: *Cycle of length $\ell + 1$ found*

The code to find this cycle is shown below:

Algorithm 6 Cycle Extension for $\ell < n - 1$ - Part 2

```
function EXTEND_CYCLE(G, C)
    u ← next(i for i in range(n) if not vertices_in_cycle[i])
    for i ∈ [0, ..., C.vertices.size() − 1] do
        edge_1 ← check_edge(G, u, C.vertices[i + 1], c_1)
        edge_2 ← check_edge(G, u, C.vertices[i], c_2)
        if edge_1 ≠ None and edge_2 ≠ None then
            vertices ← C.vertices[i + 1 :] + C.vertices[: i + 1] + [u]
            edges ← C.edges[i + 1 :]+
                    C.edges[: i − 1]+
                    [edge_2, edge_1]
            return Cycle(G, vertices, edges)
        end if
    end for
    assert False, "Should not reach here"
end function
```

This part works in $O(n)$ time. We just find a vertex outside the cycle, and make a linear search to find which edge to remove.

2.2.4 Case 2: Cycle of length $\ell = n - 1$

This is the most complicated case. We define first some auxiliary functions to simplify the code.

Auxiliary Functions

- `find_adjacency(u, color, vertex_positions)`. This function gives the indexes of the vertices that are adjacent to u with color $color$, based on the current positions of the vertices. This is useful when we need to change the vertex position in the cycle. The algorithm is shown below.

Algorithm 7 Find Adjacency Index List for a Given Source and Color

```
1: function FIND_ADJACENCY(u, color, vertex_positions)
2:     ans ← []
3:     for tgt ∈ G.adjacency[color][u] do
4:         ans.append(vertex_positions[tgt])
5:     end for
6:     return ans                                  ▷ Return the list of positions
7: end function
```

The complexity of Algorithm 7 is $O(n)$, as we just need to iterate over the adjacency list of u with color $color$.

- `find_answer(G, y, cy, C, i)`. Let G be the input graph collection, C a cycle with size $n-1$, y be the only vertex outside the cycle, c_y be the only color outside the cycle and i the index of the edge e_i to be removed. This function builds a cycle with size n by removing the edge e_i and adding the edges $edge(x_i, y, e_i.color)$ and $edge(y, x_{i+1}, c_y)$.

  The algorithm is shown below.

Algorithm 8 Find Answer for $\ell = n - 1$

```
1: function FIND_ANSWER(G, y, cy, C, i)
2:     vertices ← C.vertices[i + 1 :] + C.vertices[: i + 1] + [y]
3:     edges ← C.edges[i + 1 :] + C.edges[: i]
4:             +[G.get_edge(vertices[−1], y, C.edges[i].color), G.get_edge(y, C.vertices[0], cy)]
5:     return Cycle(G, vertices, edges)
6: end function
```

The complexity of Algorithm 8 is $O(n)$, as we just need to rotate some lists of size n and the other operations are $O(1)$.

Algorithm

This case is slightly more complicated. It is the last iteration of the algorithm and will find the desired cycle.

Let y and c_y be the vertex and color that are not in the cycle. We will rearrange the labels of the colors such that the remaining color has label $n-1$. Let us also define the following variables:

- `new_color_id`: an array of size n in which `new_color_id[i]` is the new label of color i,
- `vertex_position_on_cycle`: an array of size n in which `vertex_position_on_cycle[i]` is the position of vertex i in the cycle.
- `color_in_position`: an array of size n in which `color_in_position[i]` is the color of the edge that connects the vertices i and $i+1$ in the cycle.

These variables can be calculated in $O(n)$ time. We now describe the algorithm.

Algorithm 9 Cycle Extension for $\ell = n - 1$ - Part 1

```
function EXTEND_CYCLE(G, C)
    new_color_id ← [−1] × n
    y ← next(i for i in range(n) if not vertices_in_cycle[i])
    cy ← next(i for i in range(n) if not colors_in_cycle[i])
    new_color_id[cy] ← n − 1
    for i ∈ [0, ..., C.size() − 1] do
        new_color_id[color] ← i
        color_in_position ← C.edges[i].color
    end for
    vertex_position_on_cycle ← [−1] × n
    for i ∈ [0, ..., C.size() − 1] do
        vertex_position_on_cycle[C.vertices[i]] ← i
    end for
end function
```

Let us build the following digraph on the same vertex set of G. (See Figure 2.7 for a visual representation.)

$$A(D) = \bigcup_{i \in \{0,\ldots,n-2\}} \left\{ \{x_i, z\} : z \neq x_{i+1}, \{x_i, z\} \in A(G_{C.edges[i].color}) \right\}.$$

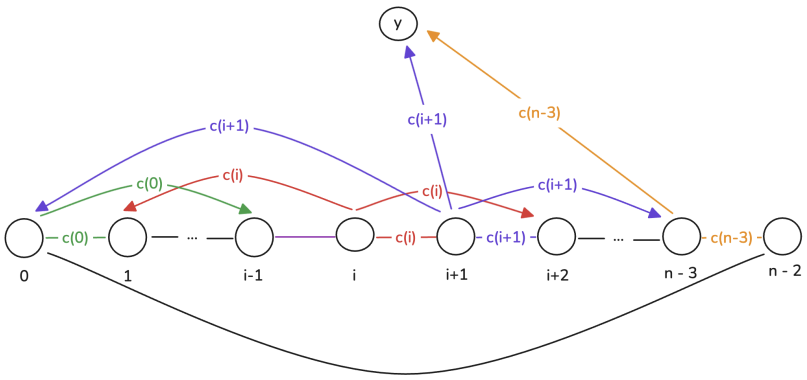

Figure 2.7: *Digraph construction for the case $\ell = n - 1$.*

Since $\delta(G_i) \geq \frac{n}{2}$ for all $i \in [0, n-2]$, we have $d_D^+(x) \geq \frac{n}{2} - 1$ for every vertex x in the cycle. Thus, $|A(D)| \geq (n-1)(\frac{n}{2} - 1)$.

Now, let us define the following variables:

- *in_degree*: an array of size n, in which $in\_degree[i]$ is the in-degree of vertex i in the digraph D.
- *incoming_neighborhood*: an array of size n, in which $incoming\_neighborhood[i]$ is the list of vertices that have an edge entering vertex i in the digraph D.

- $I := \{i \in [n-1] : \{x_i, y\} \in A(D)\}$.

- $\bar{I} := \left\{i \in [n-1] : \{y, x_{i+1}\} \in A(G_{cy})\right\}$.

We can build these variables with the following algorithm.

Algorithm 10 Cycle Extension for $\ell < n - 1$. - Part 2: Building digraph variables, I and $\bar{I}$

```
1:  function EXTEND_CYCLE(G, C)
2:      I ← []
3:      in_degree ← [0] × n
4:      incoming_neighborhood ← [empty list] × n
5:      for i ← 0 to C.size() − 1 do
6:          u ← C.vertices[i]
7:          v ← C.vertices[(i + 1) mod C.size()]
8:          color ← color_in_position[i]
9:          for tgt ∈ G.adjacency[color][u] do
10:             if tgt = v then
11:                 continue
12:             end if
13:             in_degree[tgt] ← in_degree[tgt] + 1
14:             incoming_neighborhood[tgt].append(u)
15:             if tgt = y then
16:                 I.append(i)
17:             end if
18:         end for
19:     end for
20:     Ī ← FIND_ADJACENCY(y, c_y, vertex_position_on_cycle)
21:     Ī ← [(u − 1 + C.size()) mod C.size() : u ∈ Ī]
22: end function
```

The time complexity of this algorithm is $O(n^2)$, as we need to iterate over the vertices of the cycle and iterate over its adjacency list for a specific color. The space complexity is $O(n^2)$, as we need to store the adjacency list of each vertex.

Let us describe the construction for the case $d_D^-(y) \geq \frac{n}{2}$. We have that $|I|+|\bar{I}| \geq d_D^-(y)+\frac{n}{2}$. Thus, by the Pigeonhole Principle, $I \cap \bar{I} \neq \emptyset$. In this case, we can build a cycle of length n with the following crossing, removing the edge e_i:

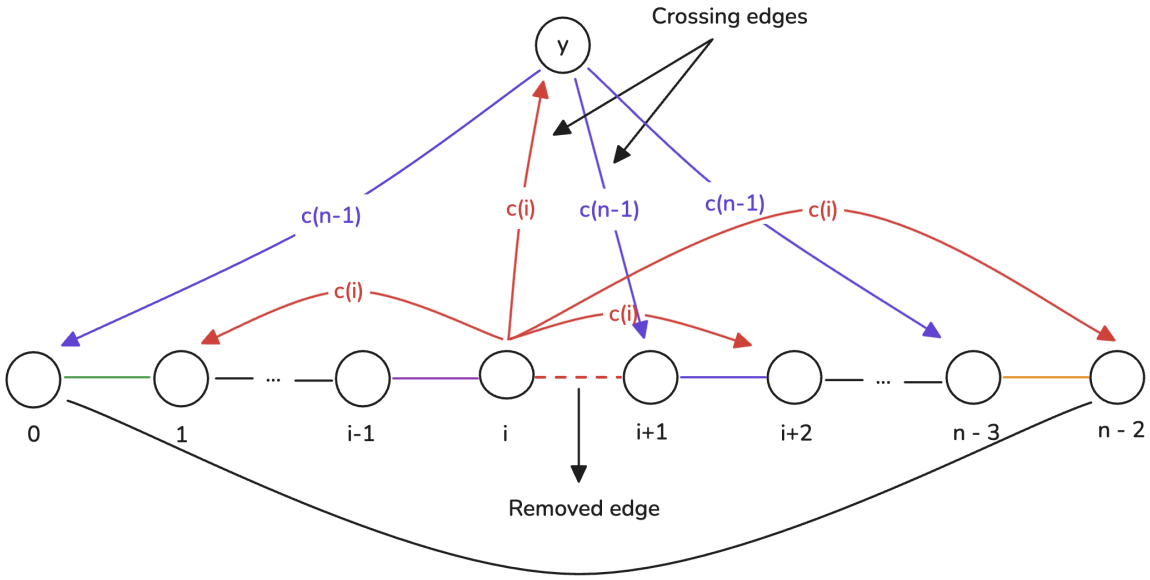

Figure 2.8: *Crossing for the case* $d_D^-(y) \geq \frac{n}{2}$.

Algorithmically, we can find the intersection and build the answer.

Algorithm 11 Cycle Extension for $\ell < n - 1$. - Part 3: Case $d_D^-(y) \geq \frac{n}{2}$

```
1: function EXTEND_CYCLE(G, C)
2:     for i ∈ I do
3:         if i ∈ Ī then
4:             return FIND_ANSWER(G, y, cy, C, i)
5:         end if
6:     end for
7: end function
```

The complexity of Algorithm 11 is $O(n^2)$, as we just need to iterate over I, check if an element in I is in $\bar{I}$ and, once we find an element in $\bar{I}$, we call the function `Find_Answer` that has complexity $O(n)$.

From now on, we assume that $d_D^-(y) < \frac{n}{2}$ and consider two cases:

1. If there is a vertex x_i such that $d_D^-(x_i) > \frac{n}{2} - 1$.

2. Otherwise.

Let us start with the first case. We may assume, wlog, that $d_D^-(x_0) > \frac{n}{2} - 1$. To do this, we can just find a vertex with in-degree greater than $\frac{n}{2} - 1$ and rotate the cycle, as described in the following algorithm:

Algorithm 12 Part 4: Cycle Extension for $\ell < n-1$. Case $d_D^-(y) < \frac{n}{2}$

```
1:  function EXTEND_CYCLE(G, C)
2:      for i ∈ [0, ..., C.size() − 1] do
3:          u ← C.vertices[i]
4:          if in_degree[u] > n/2 − 1 then
5:              C.vertices ← C.vertices[i :] + C.vertices[: i]
6:              C.edges ← C.edges[i :] + C.edges[: i]
7:          end if
8:          for j ∈ [0, ..., C.size() − 1] do
9:              color ← C.edges[j].color
10:             new_color_id[color] ← j
11:             vertex_position_on_cycle[C.vertices[j]] ← j
12:         end for
13:         break
14:     end for
15: end function
```

The time complexity of Algorithm 12 is $O(n)$, as we iterate over the vertices and as soon as we find the desired vertex, we make $O(n)$ operations to rebuild the needed variables and then we break.

Let us define the following variables:

- $I_0 := \left\{ i \in [n-1] : \{x_i, y\} \in A(G_{c_0}) \right\}$
- $I_{n-1} := \left\{ i \in [n-1] : \{y, x_{i+1}\} \in A(G_{c_{n-1}}) \right\}$

We have that $|I_0| + |I_{n-1}| \geq d(0, y) + d(n-1, y) \geq n$. Thus, by the Pigeonhole Principle, $I_0 \cap I_{n-1} \neq \emptyset$. Take $j \in I_0 \cap I_{n-1}$. If $j = 0$, we can just remove the edge x_0 and build the cycle of length n adding crossing edges $edge(x_0, y, c_0)$ and $edge(y, x_1, c_1)$. So, assume $j \neq 0$.

Let us construct a Hamiltonian path P indicated in Figure 2.9:

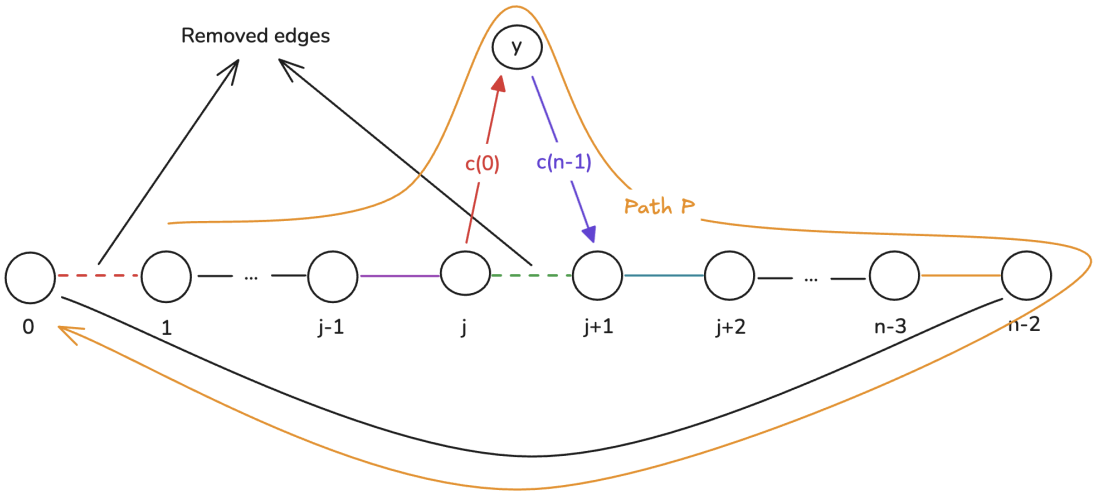

Figure 2.9: *Building a Hamiltonian path P.*

We are going to create the following variables to manipulate the path P:

- $removed\_color$: the color of the edge outside the path P, $C.edges[j].color$;
- $P\_vertices$: the vertices of the path P, $(x_1, x_2, \dots, x_j, y, x_{j+1}, \dots, x_{n-2}, x_0)$;
- P_edges: the edges of the path P, $(e_1, e_2, \dots, edge(x_j, y, e_0.color), edge(y, x_{j+1}, cy), \dots, e_{n-2}, e_0)$;
- P_pos: an array to store the new index of each vertex on the path.

To compute these variables, we can use the following algorithm:

Algorithm 13 Part 5: Cycle Extension for $\ell < n-1$. Case $d_D^-(y) < \frac{n}{2}$

```
1:  function EXTEND_CYCLE(G, C)
2:      j ← −1
3:      removed_color ← −1
4:      P_vertices ← []
5:      P_edges ← []
6:      P_pos ← [0] × n
7:      for i ∈ I_0 do
8:          if i ∈ I_{n−1} then
9:              if i = 0 then
10:                 return FIND_ANSWER(G, y, cy, C, i)
11:             end if
12:             j ← i
13:             removed_color ← C.edges[j].color
14:             P_vertices ← C.vertices[1 : j + 1]+
15:                        ← [y]+
16:                        ← C.vertices[j + 1 : n]+
17:                        ← [C.vertices[0]]
18:             P_edges ← C.edges[1 : j]+
19:                     ← [get_edge(G, C.vertices[j], y, C.edges[0].color)]+
20:                     ← [get_edge(G, y, C.vertices[(j + 1) mod n], cy)]+
21:                     ← C.edges[j + 1 : n]
22:             for j ∈ [0, ..., n − 1] do
23:                 P_pos[P_vertices[j]] ← j
24:             end for
25:             break
26:         end if
27:     end for
28: end function
```

The time complexity of Algorithm 13 is $O(n^2)$, as we iterate over I_0, check if an element in I_0 is in I_{n-1} and, if it is, we build the path P. To build the path it takes $O(n)$ operations.

Let us now create the following variables:

- $J_0 := \{i \in [n-2] : edge(P\_vertices[0], P\_vertices[i+1], removed\_color) \in E(G)\}$.
- $J_{n-1} := \{i \in [n-2] : P\_vertices[i] \in incoming\_neighborhood[P\_vertices[n-1]]\}$.

If $edge(P\_vertices[0], P\_vertices[-1], removed\_color)$ is in G, we can just close the cycle adding this edge. Let us assume that this is not the case. Thus, $|J_0| \geq \delta(G_{removed\_color}) \geq \frac{n}{2}$. Also, as $P\_vertices[n-2] \in \{x_{n-2}, y\}$, from the construction of D we guarantee that $P\_vertices[n-2] \notin N_D^-(x_0)$ and consequently $|J_{n-1}| \geq \frac{n}{2} - \frac{1}{2}$.

By the Pigeonhole Principle, as $|J_0| + |J_{n-1}| \geq n$, we have that $|J_0 \cap J_{n-1}| \geq 2$. Thus, there is at least one element in $J_0 \cap J_{n-1}$, say k, such that $P\_vertices[k] \neq y$. We can then build the following Hamiltonian cycle:

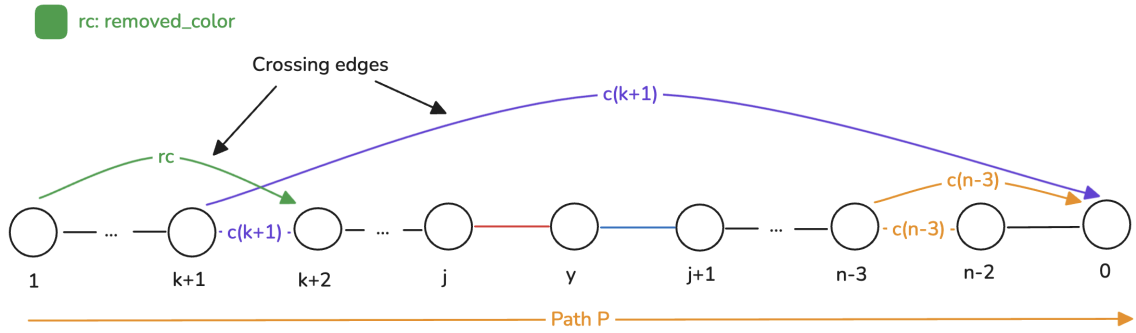

Figure 2.10: *Crossing for the case* $d_D^-(y) < \frac{n}{2}$ *and* $P\_vertices[k] \neq y$.

The algorithm to find the answer in this case is the following:

Algorithm 14 Part 6: Cycle Extension for $\ell < n-1$. Case $d_D^-(y) < \frac{n}{2}$

```
1:  function EXTEND_CYCLE(G, C)
2:      edge ← G.check_edge(P_vertices[0], P_vertices[−1], removed_color)
3:      if edge is not None then
4:          return CYCLE(G, P_vertices, P_edges + [edge])
5:      end if
6:      J1 ← FIND_ADJACENCY(P_vertices[0], removed_color, P_pos)
7:      J1 ← [(u − 1 + n) mod n : u ∈ J1]
8:      Jn ← incoming_neighborhood[P_vertices[−1]]
9:      Jn ← [P_pos[u] : u ∈ Jn]
10:     for i ∈ J1 do
11:         if i ∈ Jn then
12:             if P_vertices[i + 1] = y then
13:                 continue
14:             end if
15:             edge_1 ← G.get_edge(P_vertices[i], P_vertices[−1], P_edges[i].color)
16:             edge_2 ← G.get_edge(P_vertices[i + 1], P_vertices[0], removed_color)
17:             final_vertices ← P_vertices[: i + 1] + P_vertices[i + 1 :][:: −1]
18:             final_edges ← P_edges[: i]+
19:                           [edge_1]+
20:                           P_edges[i + 1 :][:: −1]+
21:                           [edge_2]
22:             return CYCLE(G, final_vertices, final_edges)
23:         end if
24:     end for
25: end function
```

The time complexity of Algorithm 14 is $O(n^2)$, as we iterate over J_0 and check if an element of J_0 is in J_{n-1}. If yes, we build the cycle and return it, in $O(n)$ operations. Therefore, when there exists x_i such that $d_D^-(x_i) > \frac{n}{2} - 1$, we have a total complexity of $O(n^2)$.

Now, let us analyze the case in which every x_i satisfies $d_D^-(x_i) \leq \frac{n}{2} - 1$. Define

$$\mathcal{J} := \left\{ i \in [n-1] \,:\, d_D^-(x_i) = \left\lfloor \frac{n}{2} - 1 \right\rfloor \right\}. \tag{2.1}$$

From the construction of D, as $d_D^+(y) = 0$ and $d_D^-(y) \leq \frac{n}{2} - 1$, we have:

$$|A(D-y)| \geq (n-1)\left(\frac{n}{2} - 1\right) - \frac{n}{2} + 1 > (n-1)\left(\frac{n}{2} - \frac{3}{2}\right). \tag{2.2}$$

Therefore, from 2.2 and 2.1, if we analyze the sum of the in-degrees of the vertices:

$$|\mathcal{J}| \left\lfloor \frac{n}{2} - 1 \right\rfloor + (n - 1 - |\mathcal{J}|) \left\lfloor \frac{n}{2} - 2 \right\rfloor \geq |A(D-y)| > (n-1)\frac{n}{2} - \frac{3}{2}. \tag{2.3}$$

Therefore, $|\mathcal{J}| \geq \frac{n}{2} > \frac{n-1}{2}$. Defining $\mathcal{J}' := \left\{ i \in [n-1] \,:\, edge(x_{i+1}, y, c_y) \in A(G) \right\}$, we have that $|\mathcal{J}'| + |\mathcal{J}| \geq n$. By the Pigeonhole Principle, there exists $j \in \mathcal{J}' \cap \mathcal{J}$. Let us build the following transversal path Q:

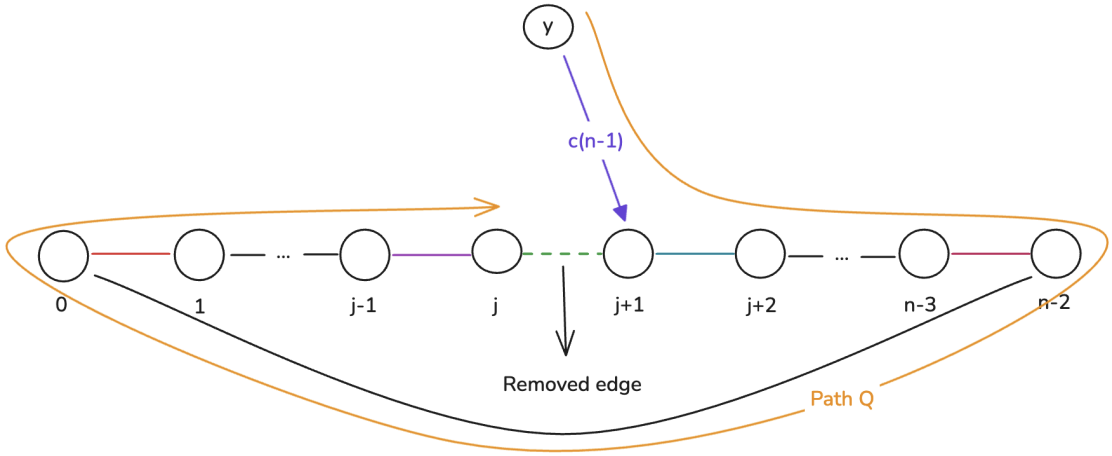

Figure 2.11: *Building the transversal path Q.*

The algorithm to build Q and get the new missing color is the following:

Algorithm 15 Part 7: Cycle Extension for $\ell < n-1$. Case $d_D^-(y) < \frac{n}{2}$

```
1: function EXTEND_CYCLE(G, C)
2:     Q ← Path()
3:     Q_pos ← [0] × n
4:     removed_color ← −1
5:     for j ∈ [0, ..., n − 1] do
6:         edge ← G.get_edge(y, C.vertices[j], cy)
7:         if edge is None or in_degree[C.vertices[j]] ≠ ⌊n/2 − 1⌋ then
8:             continue
9:         end if
10:        Q_vertices ← [y] + C.vertices[j + 1 :] + C.vertices[: j + 1]
11:        Q_edges ← [edge] + C.edges[j + 1 :] + C.edges[: j]
12:        removed_color ← C.edges[j].color
13:        for i ∈ [0, ..., n − 1] do
14:            Q_pos[Q_vertices[i]] ← i
15:        end for
16:        Q ← PATH(G, Q_vertices, Q_edges)
17:        break
18:    end for
19: end function
```

This is $O(n)$ as the most expensive operations are the manipulations on C. Now, define:

$$J_0 := \{i \in [0, n-3] : \{y, Q.vertices[i+1]\} \in A(G_{removed\_color})\}, \tag{2.4}$$

$$J_{n-1} := \{i \in [1, n-3] : Q.vertices[i] \in incoming\_neighborhood[Q.vertices[n-1]]\}. \tag{2.5}$$

If $\{y, Q.vertices[0]\} \in A(G_{removed\_color})$, we can just close the cycle adding this edge. Let us assume that this edge is not in G. We have that $|J_0| \geq \delta(G_{removed\_color}) \geq \frac{n}{2}$. Observe that $Q.vertices[0] = y \notin N_D^-(Q_vertices[n-1])$ and $Q.vertices[n-2] = C.vertices[j-1] \notin N_D^-(Q.vertices[n-1])$, by the construction of D. As $C.vertices[j] \in \mathcal{J}$, we get that $|J_{n-1}| = \lfloor \frac{n}{2} - 1 \rfloor$. We obtain that $|J_0| + |J_{n-1}| \geq n-1$. Thus, from Pigeonhole Principle, there exists $k \in (J_0 \backslash \{0\}) \cap J_{n-1}$. We can finally build the following transversal cycle:

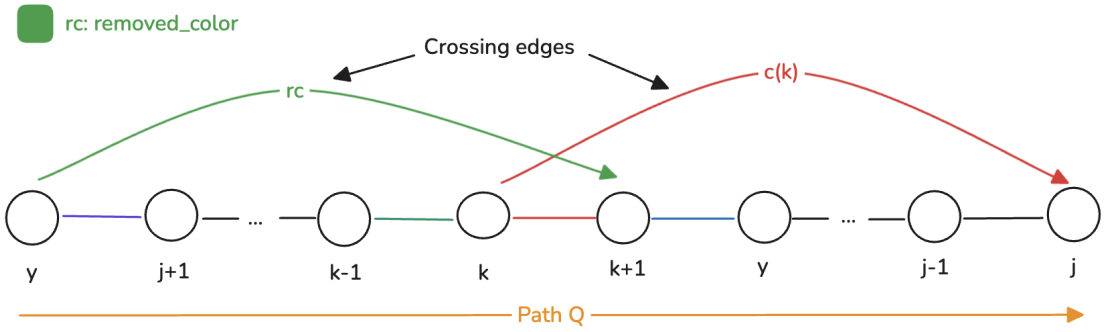

Figure 2.12: *Final crossing for the case* $d_D^-(y) < \frac{n}{2}$ *and* $P\_vertices[k] \neq y$.

The algorithm to find the answer in this case is the following:

Algorithm 16 Part 8: Cycle Extension for $\ell < n-1$. Case $d_D^-(y) < \frac{n}{2}$

```
1:  function EXTEND_CYCLE(G, C)
2:      Q ← None
3:      removed_color ← −1
4:      for j ∈ [2, ..., n − 1] do
5:          edge ← G.check_edge(y, C.vertices[(j + 1) mod n], cy)
6:          if edge is None or in_degree[C.vertices[j]] ≠ ⌊n/2 − 1⌋ then
7:              continue
8:          end if
9:          Q_vertices ← [y] + C.vertices[j + 1 :] + C.vertices[: j + 1]
10:         Q_edges ← [edge] + C.edges[j + 1 :] + C.edges[: j]
11:         removed_color ← C.edges[j].color
12:         Q ← PATH(G, Q_vertices, Q_edges)
13:         break
14:     end for
15:     if Q is None then
16:         raise RuntimeError("Did not find Path Q :(")
17:     end if
18:     edge ← check_edge(G, Q.vertices[−1], Q.vertices[0], removed_color)
19:     if edge is not None then
20:         return CYCLE(G, Q.vertices, Q.edges + [edge])
21:     end if
22:     J0 ← []
23:     for i ∈ [0, ..., n − 2] do
24:         edge ← G.check_edge(Q.vertices[0], Q.vertices[i + 1], removed_color)
25:         if edge is not None then
26:             J0.append((i, edge))
27:         end if
28:     end for
29:     position ← [0] × n
30:     for i ∈ [0, ..., n − 1] do
31:         position[Q.vertices[i]] ← i
32:     end for
33:     inJn ← [False] × n
34:     for i ∈ [0, ..., n − 1] do
35:         inJn[position[i]] ← True
36:     end for
37:     for k, edge_from_first ∈ J0 do
38:         if notinJn[k] then
39:             continue
40:         end if
41:         edge_from_last ← get_edge(G, Q.vertices[n − 1], Q.vertices[k], Q.edges[k].color)
42:         new_vertices ← Q.vertices[: k + 1] + Q.vertices[k + 1 :][: : −1]
43:         new_edges ← Q.edges[: k]+
44:                     [edge_from_last]+
45:                     Q.edges[k + 1 :][: : −1]+
46:                     [edge_from_first]
47:         return CYCLE(G, new_vertices, new_edges)
48:     end for
49:     raise Should never reach this point!
50: end function
```

2.3 Time Complexity Analysis

To evaluate the empirical performance of our implementation, we conducted extensive runtime analysis across different input sizes. For each value of n ranging from 6 to 100, we executed the algorithm 3 times on each collection and recorded the average execution time.

The results are shown in Figure 2.13, where we plot both the actual running times and the theoretical $O(n^3)$ curve for comparison. The empirical results closely follow the theoretical bound, confirming that our implementation achieves the expected complexity.

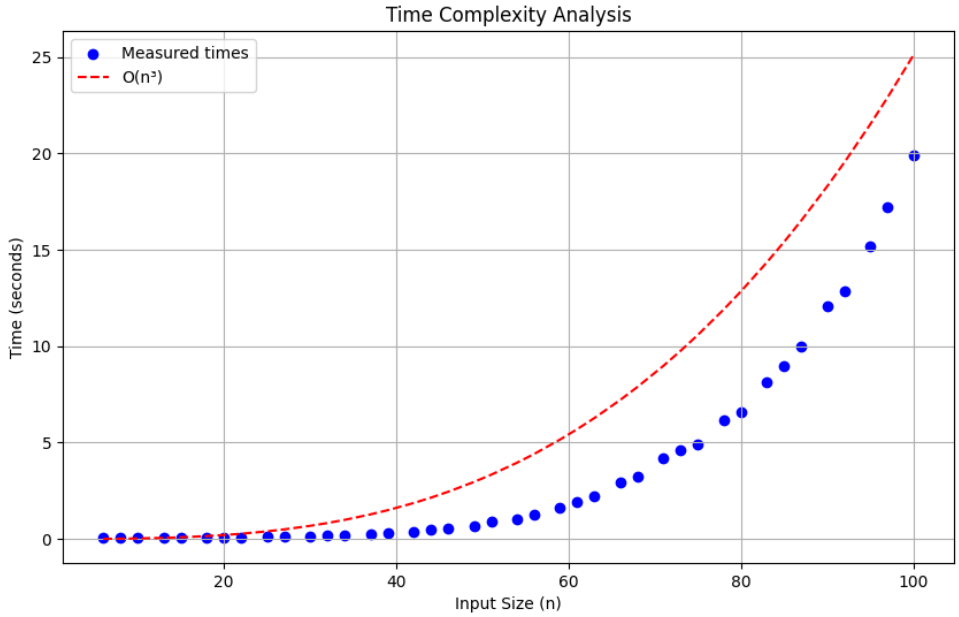

Figure 2.13: *Empirical time complexity analysis. The blue points represent the average execution times for different values of n, while the red curve shows the theoretical $O(n^3)$ bound.*

The dominant factors contributing to the $O(n^3)$ complexity are:

- The initialization of the graph collection data structures;
- Running the algorithm for incrementing object $O(n)$ times.

2.4 Testing Methodology

To validate our implementation, we developed a systematic and as well as a random approach.

2.4.1 Random testing

The core of our testing approach relied on randomly generated test cases:

- We created a generator for random graph collections satisfying Dirac's condition;

- For each graph of order n ranging from $n = 6$ to $n = 100$, we ran multiple test iterations;
- Each test verified that the algorithm successfully found a rainbow Hamiltonian cycle.

2.4.2 Special test case

While random testing provided good coverage, we discovered an interesting edge case that required special attention. This case emerged when two specific conditions aligned:

- A cycle reaching length $n - 1$;
- Every vertex on the digraph having an in-degree below $\frac{n}{2}$.

To handle this case, we developed a dedicated test in `increment.py` called `test_handle_cycle_n_minus_1_upper_bound_degree_D`. This test uses carefully constructed input graphs to verify the algorithm's behavior in this specific scenario.

Chapter 3

Rainbow version of Ore's Theorem

In 1960, Ore (Ore, 1960) proved the following result.

Theorem 1 (Ore, 1960). *If $G = (V, E)$ is a simple graph of order $n \geq 3$ such that $d_G(u) + d_G(v) \geq n$ for all pair of non-adjacent vertices $u, v \in V$, then G contains a Hamiltonian cycle.*

In what follows, we shall refer to the above sufficient condition as *Ore's condition*. It is immediate that Ore's theorem generalizes Dirac's theorem. So, a natural question that arises is whether an analogous rainbow version of Ore's Theorem also holds. We conjecture that the answer is yes. For the moment, we are able to prove that the following (weaker) result holds.

Theorem 2. *Let $n \geq 3$ and $G = G_0 \cup G_1 \cup \ldots \cup G_{n-1}$ be a graph that is the union of n pairwise edge-disjoint simple graphs G_i of order n, all defined on a same vertex set, each one monochromatically edge colored but collectively using n distinct colors. If each G_i satisfies Ore's condition, then G has a rainbow path and a rainbow cycle of length $n - 1$.*

Proof. As in the proof of the rainbow version of Dirac's theorem, we consider two cases.

> **Case (a):** Given a rainbow path in G of length $\ell < n - 1$, how to extend it to a rainbow path of length $\ell + 1$ or to a rainbow cycle of length ℓ ou $\ell + 1$.
>
> **Case (b):** Given a rainbow cycle in G of length $\ell < n - 1$, how to obtain a rainbow cycle of length $\ell + 1$ or a rainbow path of length $\ell + 1$.

We note that, it is immediate that G has a rainbow path of length 2. Thus, starting with such a path, and considering the two cases above, we are able to construct a rainbow path in G of length $n - 1$ and a rainbow cycle of length $n - 1$.

From now on, whenever we refer to a path or a cycle, these are always rainbow, so we omit stating this (but it should be understood). To simplify notation, as here we do not have many indices for the vertices, when referring to an edge with endpoints u and

v, instead of writing $\{u, v\}$, we may represent it as uv. Also, to state that an edge uv has color c, we write $uv \in G_c$.

Case (a): Path of length ℓ

Let $P = (x_0, e_0, \dots, x_{\ell-1}, e_{\ell-1}, x_\ell)$ be a path in G of length ℓ.

Case (a1): $\ell < n/2$

Let c be a color that is not in the path P, and let $u := x_0$ and $v := x_\ell$. If $uv \in G_c$, then by adding the edge vu to P we obtain a cycle of length $\ell + 1$. Else, since $uv \notin G_c$, by Ore's condition, we have that $d_{G_c}(u) + d_{G_c}(v) \geq n$, and hence there is a vertex $w \notin V(P)$ such that $uw \in G_c$ or $vw \in G_c$. Indeed, if such a vertex w does not exist, then $d_{G_c}(u) \leq \ell$ and $d_{G_c}(v) \leq \ell$. But in this case, we have that $d_{G_c}(u) + d_{G_c}(v) \leq 2\ell < n$, a contradiction. Given such a vertex w, we can construct a path of length $\ell + 1$ by appending w to the path P and using precisely one of the edges uw or vw.

Case (a2): $n/2 \leq \ell < n - 1$

Let P' be the path obtained from P after removing its last vertex x_ℓ, i.e., $P' = (x_0, e_0, \dots, x_{l-1})$. There are two colors, say c_0 and c_1, that are not present in P'. We may assume, without loss of generality, that $G_0 := G_{c_0}$ and $G_1 := G_{c_1}$.

Let $u := x_0$ and $v := x_{\ell-1}$. For a vertex $w \in V(G_i)$, let $d^{\text{out}}_{G_i}(w)$ denote the number of neighbors of w in G_i that are not in the path P (they are out of P). Analogously, let $d^{\text{in}}_{G_i}(w)$ denote the number of neighbors of w in G_i that are in the path P. By definition, $d^{\text{in}}_{G_i}(w) + d^{\text{out}}_{G_i}(w) = d_{G_i}(w)$.

If $uv \in G_0$ or $uv \in G_1$, then we can extend P' to a cycle of length ℓ by adding uv to P'. Else, by Ore's condition, $d_{G_0}(u) + d_{G_0}(v) \geq n$ and $d_{G_1}(u) + d_{G_1}(v) \geq n$. If there exists a vertex w not in P' such that both $uw \in G_0$ and $vw \in G_1$, then we can extend P' to a cycle of length $\ell + 1$ by adding w and the edges vw and wv. Such an extension is also possible if $uw \in G_1$ and $vw \in G_0$.

So, let us assume that such a vertex w does not exist. In this case, we have that $d^{\text{out}}_{G_0}(u) + d^{\text{out}}_{G_1}(v) \leq n - \ell$ and $d^{\text{out}}_{G_1}(u) + d^{\text{out}}_{G_0}(v) \leq n - \ell$. So, we have that $d^{\text{in}}_{G_0}(u) + d^{\text{in}}_{G_0}(v) + d^{\text{in}}_{G_1}(u) + d^{\text{in}}_{G_1}(v) \geq 2n - 2(n - \ell) = 2\ell$. We must have that either $d^{\text{in}}_{G_0}(u) + d^{\text{in}}_{G_1}(v) \geq \ell$ or $d^{\text{in}}_{G_1}(u) + d^{\text{in}}_{G_0}(v) \geq \ell$. Suppose, without loss of generality, that $d^{\text{in}}_{G_0}(u) + d^{\text{in}}_{G_1}(v) \geq \ell$. By the Pigeonhole Principle, there exists i such that $ux_i \in G_0$ and $vx_{i-1} \in G_1$. Thus, we can construct a a cycle of length ℓ.

Case (b): Cycle of length $\ell < n - 1$

Let $C = \{x_0, e_0, \dots, x_{\ell-1}, e_{\ell-1}, x_\ell\}$ be a cycle of length $\ell < n - 1$, and let c_0 and c_1 be two colors that are not present in C. Define $G_0 := G_{c_0}$ and $G_1 := G_{c_1}$.

Case (b1): $\ell < n/2$

As G_0 satisfies Ore's condition, G_0 is connected. Then, there exists an edge $uv \in G_0$ such that $u \in C$ and $v \notin C$. Let w be a vertex adjacent to u in C. If $vw \in G_1$, then we have a rainbow cycle of length $\ell + 1$. Else, since $vw \notin G_1$, by Ore's condition, $d_{G_1}(v) + d_{G_1}(w) \geq n$, and therefore there exists a vertex $z \notin C$ such that $vz \in G_1$ or $wz \in G_1$. Indeed, if not, we would have $d_{G_1}(v) \leq \ell$ and $d_{G_1}(w) \leq \ell - 1$, and we could conclude that $d_{G_1}(v) + d_{G_1}(w) \leq 2\ell - 1 < n$, a contradiction. Given such a vertex z, we can construct a path of length $\ell + 1$ if $vz \in G_1$ or $wz \in G_1$.

Case (b2): $n/2 \leq \ell < n - 1$

Suppose there exists two vertices $u, v \notin C$ such that $uv \in G_0$ and $uv \notin G_1$. By Ore's condition, $d_{G_1}(u) + d_{G_1}(v) \geq n$, which implies the existence of a vertex $w \in C$ such that $uw \in G_1$ or $vw \in G_1$ — otherwise $d_{G_1}(u) + d_{G_1}(v) \leq 2(n - \ell - 1) < n$. Thus, we can build a rainbow path of length $\ell + 1$. Otherwise, we have that for all vertices $u, v \notin C$,

$$uv \in G_0 \iff uv \in G_1.$$

Since G_0 and G_1 are connected graphs, if there is a pair of vertices $u, v \notin C$ such that $uv \in G_0$, then we can obtain a path with endpoint in u or v and in a vertex in the cycle, that has length at least $\ell + 1$.

We now assume that there is no pair of vertices $u, v \notin C$ such that $uv \in G_0$. Consider $u, v \notin C$. By Ore's condition, $d_{G_0}(u) + d_{G_0}(v) \geq n$ and $d_{G_1}(u) + d_{G_1}(v) \geq n$. We also know that every neighbor of u and v in G_0 (resp. G_1) is in the cycle C, and therefore, $d_{G_0}(u) + d_{G_0}(v) + d_{G_1}(u) + d_{G_1}(v) \geq 2n > 2\ell$. We can assume without loss of generality that $d_{G_0}(u) + d_{G_1}(v) > \ell$. This means that there exist adjacent vertices w and z in C such that $uw \in G_0$ and $zv \in G_1$. Thus, we can remove the edge wz from C and add to the resulting path $C - wz$ the edges uw and zv and obtain a path of length $\ell + 1$. □

We hope that from the rainbow Hamiltonian path in G guaranteed to exist by Theorem 2, we may succeed proving that G has a rainbow Hamiltonian cycle. This result would give us a theorem that is precisely *the rainbow version of Ore's theorem*. Anyway, we think Theorem 2 is an interesting result by its own right.

Chapter 4

Conclusion

The main objective of this work was to understand well the proof of the rainbow version of Dirac's theorem obtained by Joos and Kim in 2020, and present a detailed algorithmic proof that yields an efficient algorithm to construct a rainbow Hamiltonian cycle guaranteed by this theorem. This objective was attained, as the algorithm that we derived and implemented has asymptotically the best possible time complexity: $O(n^3)$ for input graphs with n vertices and $O(n^3)$ edges.

The biggest challenge we faced in the implementation described in Chapter 2 was in the construction of a rainbow Hamiltonian cycle from a rainbow cycle of length $n-1$, without increasing the time complexity that was achieved up to that step. We tested our implementation on a large number of randomly generated graphs with order in the range from 6 to 100, and observed that the practical results confirmed our theoretical results.

In Chapter 3 we present a statement that we call the rainbow version of Ore's theorem which is a generalization of the rainbow version of Dirac's theorem. In this statement, each graph G_i has to satisfy Ore's condition: $d(u)+d(v) \geq n$ for every pair of non-adjacent vertices u and v in $V(G_i)$. We conjecture that this statement is true. We obtained (so far) only a partial result on this statement: that the union graph G has a rainbow Hamiltonian path and also a rainbow cycle of length $n-1$. This partial result is interesting in its own, but we hope to extend this result and show – in the near future – that our conjecture is true.

We also developed an interactive visualization tool for our algorithm, which is available in our Github repository at https://github.com/wmrmrx/TCC/tree/master/code/src_python.

This monograph and our codes are all public. This way, we hope these studies will bring contribution to the field.

Bibliography

[Bondy and Murty 2008] J. A. Bondy and U. S. R. Murty. *Graph Theory*. Vol. 244. Graduate Texts in Mathematics. London: Springer, 2008 (cit. on p. 3).

[Clinch *et al.* 2022] Katie Clinch, Jackson Goerner, Tony Huynh, and Freddie Illingworth. *Notes on Aharoni's rainbow cycle conjecture*. 2022. arXiv: 2211.07897 [math.CO]. URL: https://arxiv.org/abs/2211.07897 (cit. on p. 1).

[Dirac 1952] G. A. Dirac. "Some theorems on abstract graphs". *Proceedings of the London Mathematical Society* (1952). DOI: 10.1112/plms/s3-2.1.69 (cit. on pp. 1, 3).

[Gao and Wang 2023] Liqing Gao and Jian Wang. "A rainbow version of Ore theorem". *Operations Research and Management Science* 32.10 (2023), pp. 108–113. URL: http://www.jorms.net/EN/10.12005/orms.2023.0327 (cit. on p. 1).

[Joos and Kim 2020] Felix Joos and Jaehoon Kim. "On a rainbow version of Dirac's theorem". *Bulletin of the London Mathematical Society* 52.3 (May 2020), pp. 498–504. ISSN: 1469-2120. DOI: 10.1112/blms.12343. URL: http://dx.doi.org/10.1112/blms.12343 (cit. on pp. 1, 4, 5).

[Nathanson 2006] Melvyn B. Nathanson. *The Caccetta-Häggkvist conjecture and additive number theory*. 2006. arXiv: math/0603469 [math.CO]. URL: https://arxiv.org/abs/math/0603469 (cit. on p. 1).

[Ore 1960] Oystein Ore. *Note on Hamilton Circuits*. Jan. 1960. DOI: 10.2307/2308928. URL: http://dx.doi.org/10.2307/2308928 (cit. on p. 29).