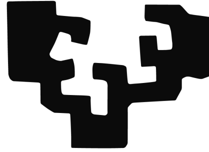
eman ta zabal zazu



Universidad del País Vasco   Euskal Herriko Unibertsitatea

Bachelor Degree in Computer Engineering

Computation

Thesis

Implementation of the “Snakes and Ladders” Heuristic for solving the Hamiltonian Cycle Problem

Author

Manuel Torralbo Lezana

2021

Abstract

The Hamiltonian Cycle Problem is a popular $\mathcal{NP}$-complete problem belonging to the field of Graph Theory and an intrinsic part of the famous Travelling Salesman Problem. Both paradigms are highly regarded in Mathematics and Computer Science due to the immense consequences that would suppose to achieve an optimal solution in investigation and research as well as in the optimization of numerous real life scenarios. As a result, there is no lack of material engaging the issues from numerous mathematical approaches, one of them being the "Snakes and Ladders" Heuristic. First introduced in 2014, the "Snakes and Ladders" Heuristic is a state of the art polynomial-time deterministic algorithm for solving the Hamiltonian Cycle Problem, which inspired by the Lin-Kernighan Heuristic uses "k-opt" transformations to search for a possible solution, achieving astounding results even with difficult graphs of different characteristics. What follows in this document is a proposal for a functional implementation of the "Snakes and Ladders" Heuristic, including in-depth analysis of the process which took place in order to conceive it.

Acknowledgements

All credit regarding the conception of the "Snakes and Ladders" Heuristic goes to Pouya Baniasadi, Vladimir Ejov, Jerzy A. Filar, Michael Haythorpe and Serguei Rossomakhine, the authors of the algorithm [4].

Contents

1 Introduction **1**

2 Project Management **3**
2.1 Planning . . . 3
2.1.1 Work Breakdown . . . 3
2.1.2 Risk Management . . . 5
2.2 Monitoring and Evaluation . . . 6

3 Context and State of the Art **7**

4 The "Snakes and Ladders" Heuristic **9**
4.1 Terminology . . . 10
4.1.1 Arrangement . . . 10
4.1.2 Ordering . . . 10
4.1.3 Segment . . . 11
4.1.4 Snake . . . 11
4.1.5 Ladder . . . 11
4.1.6 Gap . . . 12
4.2 Description of the algorithm . . . 12

5 Implementation of SLH **15**
5.1 Objects and Data Structures . . . 15
5.1.1 Graph . . . 17
5.1.2 Ordering . . . 17
5.1.3 Gap . . . 25
5.1.4 Solver . . . 25
5.2 Isomorphisms . . . 26
5.3 Transformations . . . 28
5.3.1 Closing Transformations . . . 31
5.3.2 Floating Transformations . . . 32
5.3.3 Opening Transformations . . . 33
5.4 Stages . . . 33
5.4.1 Stage 0 . . . 35

5.4.2 Stage 1 35
5.4.3 Stage 2 35
5.4.4 Stage 3 36
5.5 Run Time 39
5.6 Possible Improvements 39
5.6.1 Ordering fingerprint 39
5.6.2 Blind Improvement Space 40
5.6.3 Multithreading 41

6 Experimental Validation **42**
6.1 Benchmarks 42
6.2 Results 43

7 Web Application **46**

8 Conclusions **51**

List of Figures

2.1 Work Breakdown Structure Diagram. 4
2.2 Gantt Diagram showing the time period each of the identified tasks takes place. 5

4.1 Example of the class equivalence of the two arrangements (A and B) shown in the upper section. The bottom section proves this equivalence by transforming arrangement A to B; the first step performs a reversal and the second a two position clockwise rotation. 11
4.2 Representation of the ordering ($a \frown b$,...,c | d,...,$e \frown f$,...,a) containing the snakes $a \frown b$ and $e \frown f$, the gap c | d and the ladders between the vertices a-d and c-e. The ordering can be divided in the segments (b,. . . ,c), (d,. . . ,e) and (f,. . . ,a). 12
4.3 Isomorphism γ. 13
4.4 Isomorphism $\aleph$. 13

5.1 Class Diagram showing the relationships, properties, methods and attributes of each of the classes in the implementation. . . . 16
5.2 Example of the arrays *vertices* and *indices* for the ordering (7,9,4,3, 6,8,2,5,1,10,7). For the sake of clarity, the two arrays can be viewed as mirrors of each other since opposite to the *vertices* array, the *indices* array has the vertices as indices and their positions as values. 18
5.3 Comparison of arrangements A (2,5,1,3,4,6,2) and B (5,4,3,1,6,2,5), where $A > B$. 19
5.4 Effect of a segment reversal on the *vertices* array. 22
5.5 Floating 4-flo type 1 transformation. 28

5.6 State Diagram showing the transition between the different stages of SLH. The conditions being as follows: A) An ordering with 0 gaps found. B) The number of orderings in the stack is greater than n^2 or no more floating transformations can be performed. C) An ordering with a new minimum number of gaps found. D) All opening transformations considering the ordering at the top of the stack and one of its gaps have been considered. E) The number of orderings in the stack is greater that n^3 or the ordering at top of the stack has no possible opening transformation. . . . 34
5.7 Comparison of fingerprints of orderings A (2,5,1,3,4,6,2) and B (5,4,3,1,6,2,5), where $A > B$. Same as Figure 5.3. 40

7.1 SLH Web Application input window. 47
7.2 SLH Web Application loading screen. 48
7.3 SLH Web Application image gallery showing the initial ordering for the symmetric cubic graph C2048.25. 49
7.4 SLH Web Application image gallery showing the final ordering of the symmetric cubic graph C2048.25, which contains a Hamiltonian cycle. 50

List of Algorithms

1 Swap Segments 24
2 Isomorphism γ 26
3 Isomorphism $\aleph$ 27
4 Floating 4-flo type 1 29
5 Stage 0 of SLH 35
6 Stage 1 of SLH 36
7 Stage 2 of SLH 37
8 Stage 3 of SLH 38

List of Tables

5.1 SLH closing transformations in order of priority. Where k refers to the maximum degree of graph G. 31
5.2 SLH floating transformations in order of priority. Where k is the maximum degree of gap graph G and s is the length of segment $(x, ..., a)$, see Figure 5.5 and Algorithm 4, lines 9-11. 32
5.3 SLH opening transformation. Where k is the maximum degree of graph G and s is the length of segment $(x, ..., a)$ [see 4, page 9]. 33

6.1 Combined time required to solve all graphs in the TSPLIB challenge set and the 795 symmetric cubic graphs containing Hamiltonian cycles. 43
6.2 Average time required for 10 different tests using random initial orderings for all TSPLIB graphs. 43
6.3 Average time required for 10 different tests using random initial orderings for all the symmetric cubic graphs of 2048 vertices. . . 44
6.4 Five distinct runs of the first ten graphs in the Flinders HCP Challenge set using random initial orderings. All instances were successfully solved. 44

Chapter 1

Introduction

The Hamiltonian Cycle Problem (HCP), named after the Irish mathematician William Rowan Hamilton who first presented and studied it alongside Thomas Kirkman around 1856, is an $\mathcal{NP}$-complete problem belonging to Graph Theory and close relative of the widely known Travelling Salesman Problem (TSP).

Given a graph G containing n vertices, the HCP consists on proving whether G is a Hamiltonian graph, by finding the existence of at least one Hamiltonian cycle, or on the contrary, if G is a non-Hamiltonian graph, assuring the absence of any of such cycles. A Hamiltonian cycle being a sequence of n distinct interconnected edges, visiting all n vertices once and starting and ending at the same vertex.

The TSP goes a step beyond and determines of all Hamiltonian cycles present in G, if there are any, which has the the minimal cost, given by the sum of the weights of the edges. If the edges of G are not weighted, or all have the same weight for that matter, the TSP is simply reduced to the HCP. In other words, the HCP is an intrinsic part of the TSP, and a breakthrough in the first would inevitably affect the latter.

Such advancement would also have implications in Mathematics and Computer Science, specially in the branches of Optimization and Complexity Theory. Not only that, but these problems real world applications are numerous, just to name a few, these include logistics, data storage, circuitry, cytogenetics, etc. For this reason, the HCP and TSP problems have been studied extensively and a vast amount of literature is available, engaging the issue from multiple mathematical approaches, some of which will be discussed further along in this document.

However, this work's main focus is the state of the art "Snakes and Ladders" Heuristic (SLH) for solving the HCP. Presented for the first time in Baniasadi *et al* (2014) [4], SLH is a polynomial complexity deterministic algorithm inspired by the Lin-Kernighan heuristic [10] for solving the TSP, the primary influence

being the usage of “k-opt” transformation techniques to transition and search for a Hamiltonian cycle making incremental improvements.

The lack of an available source code for the SLH brings up this project’s motivation; to propose and facilitate a functional implementation of the algorithm. Throughout this paper, in detail documentation of the implementation is provided, alongside the rational process that took place in order to conceive it, which in some cases may open the opportunity to discuss further improvements. Subsequently, the implementation’s performance is tested against a large pool of graphs, varying in size and difficulty.

In addition, a web application with a user interface has been developed which provides a visual representation of the internal behaviour of the SLH.

Chapter 2

Project Management

The following chapter defines this projects scope and time constraints and the measures taken to control the given limitations, reduce the risks and ultimately successfully complete the project.

2.1 Planning

Keep in mind that the planning process is being conceived at the early stages of the project and may be subject to change during its course.

2.1.1 Work Breakdown

The project's work load has been divided and the identified tasks have been organized for easier management. The Work Breakdown Structure can be seen in Figure 2.1 and, in addition, a brief explanation for each task is provided. Lastly, the time period which each task will take place is shown in the Gantt Diagram, see Figure 2.2.

- **T1 Research**
  - T1.1 SLH Research: Deep analysis of the "Snakes and Ladders" Heuristic.
  - T1.2 "k-opt" Transformations: Study other algorithms which use "k-opt" transformation techniques.
  - T1.3 Graph Test Set: Gather a set of graph of different qualities to debug and benchmark the implementation.
  - T1.4 Flask WAF: Learn how to use "Flask" to develop the web application.

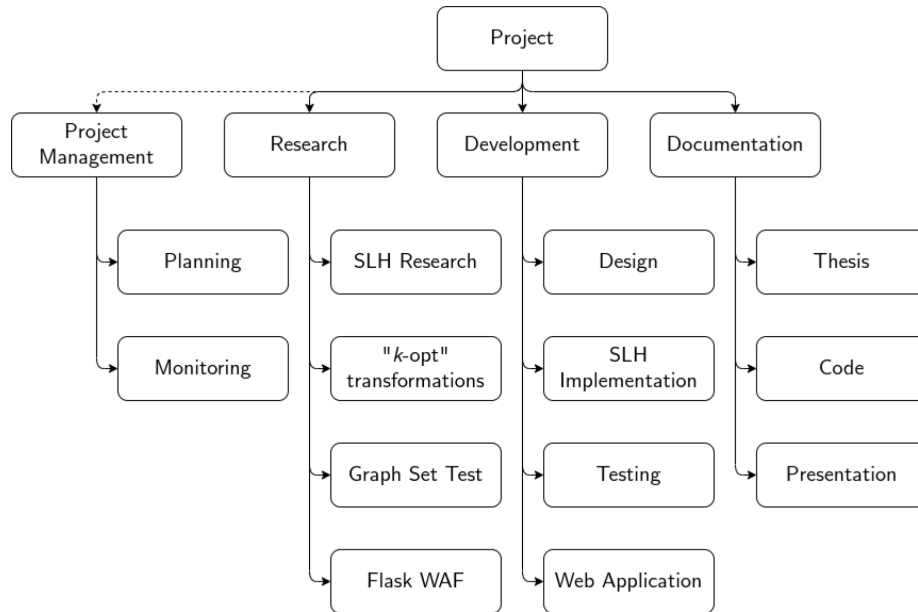


Figure 2.1: Work Breakdown Structure Diagram.

- **T2 Development**
  - T2.1 Design: Conceive the software design of the implementation.
  - T2.2 SLH Implementation: Implement the "Snakes and Ladders" Heuristics.
  - T2.3 Testing: Debug and benchmark the implementation.
  - T2.4 Web Application: Implement the web application.
- **T3 Documentation**
  - T3.1 Thesis: Write this document.
  - T3.2 Code: Document the implemented source code.
  - T3.3 Presentation: Create the required material to present this project.
- **T4 Project Management**
  - T4.1 Planning: Assess and split the work to be done in order to reduce the risks and successfully complete the project.
  - T4.2 Monitoring: Control the work done to identify potential problems.

Task	2021																							
	January				February				March				April				May				June			
1 Research																								
1.1 SLH Research	■																							
1.2 "k-opt" Transformations		■																						
1.3 Graph Test Set			■																					
1.4 Flask WAF																				■				
2 Development																								
2.1 Design			■	■																				
2.2 SLH Implementation				■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■					
2.3 Testing				■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■				
2.4 Web Application																					■	■		
3 Documentation																								
3.1 Thesis		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■		
3.2 Code				■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■		
3.3 Presentation																							■	■
4 Project Management																								
4.1 Planning	■	■																						
4.2 Monitoring			■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■

Figure 2.2: Gantt Diagram showing the time period each of the identified tasks takes place.

2.1.2 Risk Management

The following risks have been identified and, if possible, a contingency plan has been proposed:

- **R1**: As of January of 2021 the world is subject to a global pandemic of the virus COVID-19 and the possibility of contagion and further quarantines seems a likely scenario. Since this is a single person project, all the work will be done from home and a lockdown situation would not pose a serious thread. However, in the case of suffering grave symptoms the project could be delayed from one to two weeks.

- **R2**: The SLH is a state of the art algorithm with no available implementation, the only available information being the one given by the authors. This means that the implementation must be done from scratch and from just one source, for which only the given results can be contrasted. The risk of misunderstanding the authors guidance is probable and will inevitably change the output of the resulting algorithm to a greater or lesser extent.

- **R3**: Providing a publicly accessible web application entails difficulties from a couple of standpoints. Since the interface requires a user input, a secure parser and website is consequently needed. Furthermore, SLH makes use of a considerable amount of computational power and space. Even if the chance of multiple concurrent users is disregarded, acquiring the necessary infrastructure may be problematic. For these reasons,

just making a local web application for demonstrative purposes is not discarded.

- **R4**: The period this project takes place coincides with the second school quadrimester, an excessive total workload may delay this project.

2.2 Monitoring and Evaluation

During the period in which the project took place several decision where made which affected the initially planned guidelines. At the time of writing this being in the latest stages of the project the final outcome asks for a recapitulation and evaluation of the most significant events.

One of the mayor changes is a nearly three month delay of the project, until September of 2021. This is to a large extent due to risks already listed in the planning phase. First of, as assessed in R4, the month of May was in its entirety dedicated to other projects and exams, which made impossible any real advancement. Secondly, R2 posed to be even a harder challenge than expected; a working implementation of the SLH was achieved earlier in the project, however, the acquired results did not meet the standard set in the projects scope. What followed where a series of optimizations and various versions of the implementation, until the outcome was considered satisfactory enough. As of now, the implementation exceeds the expectations regarding the time required to solve large graphs, nevertheless, it still lacks a consistent record of solving difficult instances.

On another note, as predicted in R3, making a publicly accessible website, containing user inputs and needing a significant amount of infrastructure, due to the computational requirements of the SLH, ended up being an unfeasible task. Therefore, the web application made is set to be a local user interface with the sole purpose of showing how the SLH internally works.

Chapter 3

Context and State of the Art

As part of the mathematical branch of graph theory, many of the solutions provided for the HCP base their approach solely in vertices and edges of a graph and the properties within them. There are even algorithms designed to exploit the peculiarities of certain families of graphs. Such is the case with the algorithm proposed by Eppstein (2003) [6], which provides a list of all existing Hamiltonian cycles in a cubic graph in time $O(2^{3n/8})$. Using a recursive backtracking structure the algorithm takes advantage on the fact that the graph has maximum degree of three to discern if a certain edge will be considered in a cycle or not.

Trying to solve the problem by other means, most notable are the studies which, given the structural similarities between the HCP and Markov chains, use the tools and techniques of Markov decision processes that graph theory does not have access to. The "Determinant Interior Point Algorithm" proposed by Haythorpe (2010) [8], embeds the HCP in a Markov decision process creating a doubly-stochastic probability transition matrix containing the probabilities of all vertices transitions between each other. The algorithm then is described as an optimization problem which tries to find a deterministic doubly-stochastic transition matrix representing a Hamiltonian cycle.

On another note, it is not possible to talk about the HCP without mentioning the TSP, more so taking into account that any TSP solver available also tries to find Hamiltonian cycles. Consider a graph with n vertices originally intended for the HCP, but the same weight w has been assigned to all of its edges. If this very graph is provided to an algorithm designed to solve the TSP and a tour with a cost of $n \times w$ is returned, then the graph has a Hamiltonian cycle.

The highly regarded “Concorde TSP Solver” [2] is an exact algorithm, always returning the tour with the best cost, which poses the issue as a linear programming problem. Implemented as a complex branch and bound algorithm, Concorde uses cutting planes constraints to reduce the search space and correct itself as to produce a valid tour. However, it has to be noted that if the default configuration of Concorde is used, the initial solution from which the solver will build upon is constructed with the “Chained Lin-Kernighan” algorithm [1].

In the context of SLH, the “Lin-Kernighan Heuristic” [10] has mayor importance, being the first algorithm which considered an exchange of edges over a TSP tour as a means to obtain a new one with better cost, a technique now known as “k-opt” transformation, k being the number of edges exchanged. The idea was based on the basic approach of other heuristics for combinatorial optimization problems, which iteratively improved considering a random set of valid solutions.

Over the time multiple improvements have been proposed for the Lin-Kernighan Heuristic, the most notable one being Helsgaun’s Lin-Kernighan implementation [9]. With an improved and more meticulous search strategy and an added in-depth analysis focusing and restricting the search space, the running times are highly reduced compared to the original algorithm, specially in graphs with larger amount of vertices.

Chapter 4

The "Snakes and Ladders" Heuristic

Having already available the full description of the SLH by the authors themselves makes it unnecessary and redundant trying to capture in detail the algorithm here again. For this reason, only a brief explanation of the SLH will be presented, emphasizing in the concepts that provide context to the proposed implementation. The reader is of course referred to [4] for the detailed description.

Let G be a graph containing n vertices, for which edges have neither direction nor weight. Conceptually, what the SLH does is arranging all vertices along the perimeter of a circle and then adding the edges in one of two ways. If the vertices which the edge joins are contiguous in the perimeter, the arc of the circle connecting both vertices is underlined and will be called a "Snake". If, on the contrary, the vertices are not next to another, a straight line is drawn between the two, making a chord in the circle, and will be called "Ladder". Is this conceived image and its relative similarity to the popular board game "Snakes and Ladders" what gives name to the algorithm.

For any given configuration of the vertices, since the number of vertices is n, the total number of snakes can only be as many as n, and if that is the case, it would mean that all vertices along the perimeter are connected in G to their contiguous two vertices, a Hamiltonian Cycle for that matter. Searching for such arrangement of vertices containing n snakes is the purpose of the SLH.

4.1 Terminology

In order to explain the SLH, and subsequently the proposed implementation, a particular terminology and notation is introduced.

4.1.1 Arrangement

A permutation of all vertices contained in a graph placed along the perimeter of a circle. An arrangement has two directions, forward and backward, or more specifically, clockwise and counterclockwise. Having the same number of vertices as the graph it refers to, two vertices are said to be adjacent in the arrangement if they are next to each other, or put in other words, if there is no additional vertex between them.

If the contained vertices are to be specified, they are listed ordered clockwise, using commas and between parenthesis, repeating the initial vertex at the end to denote its cyclic nature and emphasize that it covers the entire perimeter of the circle. Three suspension points indicate an ellipsis of 0 or more vertices.

One example of this notation could be $(a,b,\ldots,c,a)$; a being the initial vertex is adjacent and previous of vertex b, 0 or more vertices follows ending with the final vertex c.

4.1.2 Ordering

Quoting Baniasadi *et al* (2014) [see 4, page 4]:

"The arrangement of vertices on the circle form natural equivalence classes. Namely, two arrangements are said to be equivalent if either one can be transformed to the other via a rotation or reversal, or a composition of both. ... . We use the term *ordering*, or *cycle ordering* to denote such an equivalence class, ...".

To be able to test the class equivalence of different arrangements to begin with, they have to refer to the same graph, problem which is intrinsically solved since G is the only mentioned graph.

An ordering of G is noted as C. Listing all $2n$ possible arrangements of the equivalence class would be inefficient, therefore, just one of the members is used to represent the ordering. For this reason, the term ordering is also used referring to its representative arrangement when no confusion is possible. Lastly, the number of gaps in the ordering C in noted as $g(C)$.

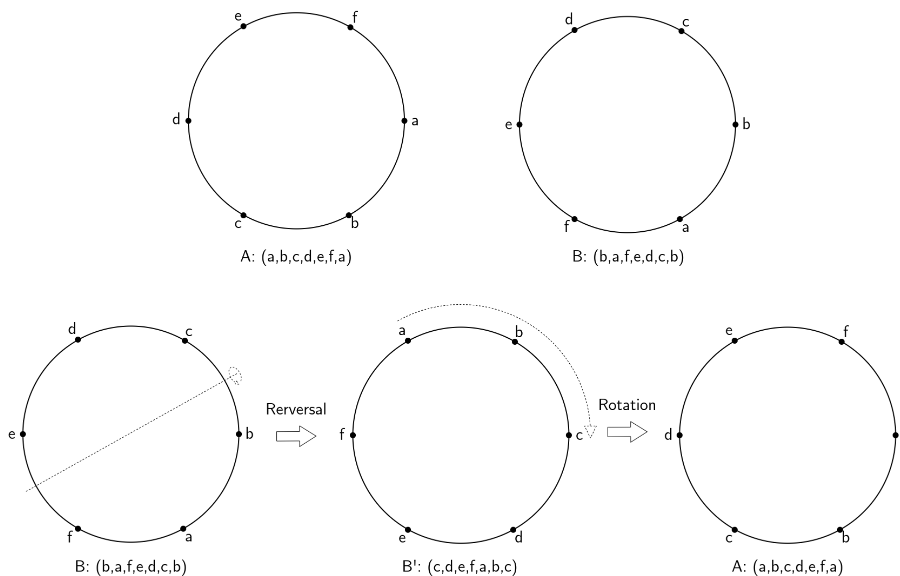


Figure 4.1: Example of the class equivalence of the two arrangements (A and B) shown in the upper section. The bottom section proves this equivalence by transforming arrangement A to B; the first step performs a reversal and the second a two position clockwise rotation.

4.1.3 Segment

A contiguous section of an ordering which does not contain the whole perimeter of the circle. If the contained vertices are to be specified, the same notation as an arrangement is used, without repeating the initial vertex at the end.

An ordering could be represented as the union of multiple segments, for instance, the ordering $(a,\ldots,b,c,\ldots,d,a)$ could be decomposed in the segment A: $(a,\ldots,b)$ and segment B: $(c,\ldots,d)$ and redefined as (A,B). In addition, A^R: $(b,\ldots,a)$ refers to the reverse of segment A.

4.1.4 Snake

Any adjacent pair of vertices in an ordering, also connected by an edge in the graph. The snake between vertices a and b is specified as $a \frown b$ and drawn as a continuous bold arc in the circle.

4.1.5 Ladder

Any pair of not adjacent vertices in an ordering connected by an edge in the graph. A ladder is drawn as a chord in the circle.

4.1.6 Gap

Opposite to a snake, as the name indicates, a gap is any adjacent pair of vertices in an ordering not connected in the graph. Noted as g, the gap between vertices a and b is specified as $a \mid b$ and drawn as a discontinuous arc in the circle.

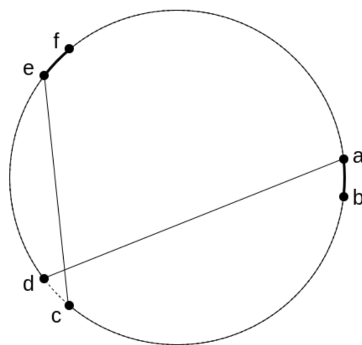


Figure 4.2: Representation of the ordering ($a \frown b$,...,$c \mid d$,...,$e \frown f$,...,a) containing the snakes $a \frown b$ and $e \frown f$, the gap $c \mid d$ and the ladders between the vertices a-d and c-e. The ordering can be divided in the segments ($b,\ldots,c$), ($d,\ldots,e$) and ($f,\ldots,a$).

4.2 Description of the algorithm

The SLH starts with an initial ordering and performs transformations over it and the subsequently obtained orderings as a means to gradually improve and ultimately find a Hamiltonian Cycle, in the form of an ordering containing n snakes or 0 gaps. The SLH is a deterministic algorithm and the initial configuration of the vertices will alter the run time and outcome, which will always be identical given the same starting ordering.

There are multiple transformations but all of them are the result of a composition of two "isomorphisms", a term derived from the Ancient Greek meaning "equal form or shape" and used broadly in mathematics referring to operations mapping an element to another of the same properties. Named γ and $\aleph$, seen in Figure 4.3 and Figure 4.4 respectively, in the context of the SLH an isomorphism is the minimal form of transformation which converts an ordering to another.

- Isomorphism γ: Having the segments $A \leftarrow (x, ..., b)$ and $B \leftarrow (a, ..., y)$, the ordering $(x, ..., b, a, ..., y, x)$ or (A, B) is mapped to the ordering $(b, ..., x, a, ..., y, b)$ or (A^R, B).
- Isomorphism $\aleph$: Having the segments $A \leftarrow (x, ..., e)$, $B \leftarrow (c, ..., a)$, $C \leftarrow (b, ..., f)$ and $D \leftarrow (d, ..., y)$, the ordering $(x, ..., e, c, ..., a, b, ..., f, d, ..., y, x)$ or (A, B, C, D) is mapped to the ordering $(e, ..., x, a, ..., c, d, ..., y, f, ..., b, e)$ or (A^R, B^R, D, C^R).

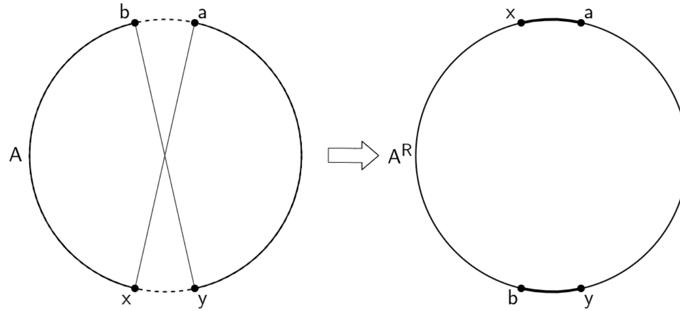


Figure 4.3: Isomorphism γ.

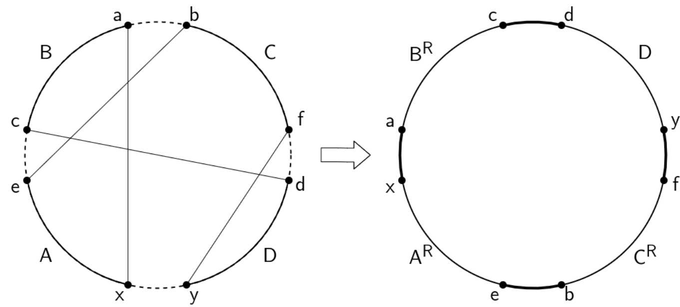


Figure 4.4: Isomorphism $\aleph$.

A transformation being a sequence of one or more isomorphisms, also maps an ordering C to another ordering C'. Regarding the number of gaps in C' compared to C, three types of transformations are defined:

- Closing Transformation: The number of gaps in C' is less than the number of gaps in C $(g(C') < g(C))$.
- Floating Transformation: The number of gaps in C' is less than or equal to the number of gaps in C $(g(C') \leq g(C))$.
- Opening Transformation: The number of gaps in C' is greater by one, equal to or less than the number of gaps in C $(g(C') + 1 \leq g(C))$.

These transformations are applied using a certain criterion, for which an algorithm with four stages is defined, trying to find a solution using different approaches:

- Stage 0: Simplest of the four, just closing transformations are applied, creating as many snakes as possible leaving the burden of closing the remaining gaps to the other stages.

- Stage 1: Only floating transformation are used with the intention of performing a lateral search until a gap is closed. The number of transformations is limited to n^2 since a gap is being closed.
- Stage 2: To avoid a local minimum an opening transformation is performed and Stage 1 is repeated. If the number of gaps is less than the previously obtained minimum return to Stage 1.
- Stage 3: An opening transformation is performed, then closing transformations are made until no more gaps can be closed. If the number of gaps is less than the previously obtained minimum return to Stage 1, otherwise, this stage is repeated. The number transformations is limited to n^3.

The worst case time complexity of the SLH is $O(n^6 \, log(n) \, k^4)$, however encountering an instance with such requirements is highly unlikely and the more realistic bound is set to $O(n^5 \, log(n) \, k^4)$. Furthermore, the algorithm solves most graphs in the first iteration of "Stage 1", which requires $O(n^4 \, k^4)$.

Chapter 5

Implementation of SLH

The proposed implementation is designed as an object-oriented program and is available in both C++ and Python programming languages. Object-oriented programming (OOP) is based on the concept of objects which interact with one another and change their state with the use of their stored data, in the form of attributes, and code, in the form of methods.

Applied to the SLH algorithm, an OPP approach makes possible, among other things, to create collections containing instances of classes representing elements such as gaps and orderings. An ordering instance in turn, also contains gaps and stores vertices in a specific order, giving it its identity. Is this relationship between different objects and the joint stored data what defines the particular state of the SLH algorithm.

Furthermore, OOP can make use of auxiliary objects, such as iterators, which allow to transverse orderings despite being cyclic, and in conjunction with ordering class methods, ease the implementation of SLH transformations.

The given explanation starts from the lower level notions, and as they are gradually acquired they are used as a means to build more complex and abstract algorithms. Therefore, the employed objects and data structures are introduced first, and then, the SLH transformations and stages are described, in that order.

5.1 Objects and Data Structures

What comes ahead is a detailed description of the objects and data structures used along the algorithm's course, these being a global manager instance called "Solver" and the class representations of graphs, orderings and gaps. An overall description of each class and the relationship between each other can be seen in the Class Diagram shown in Figure 5.1.

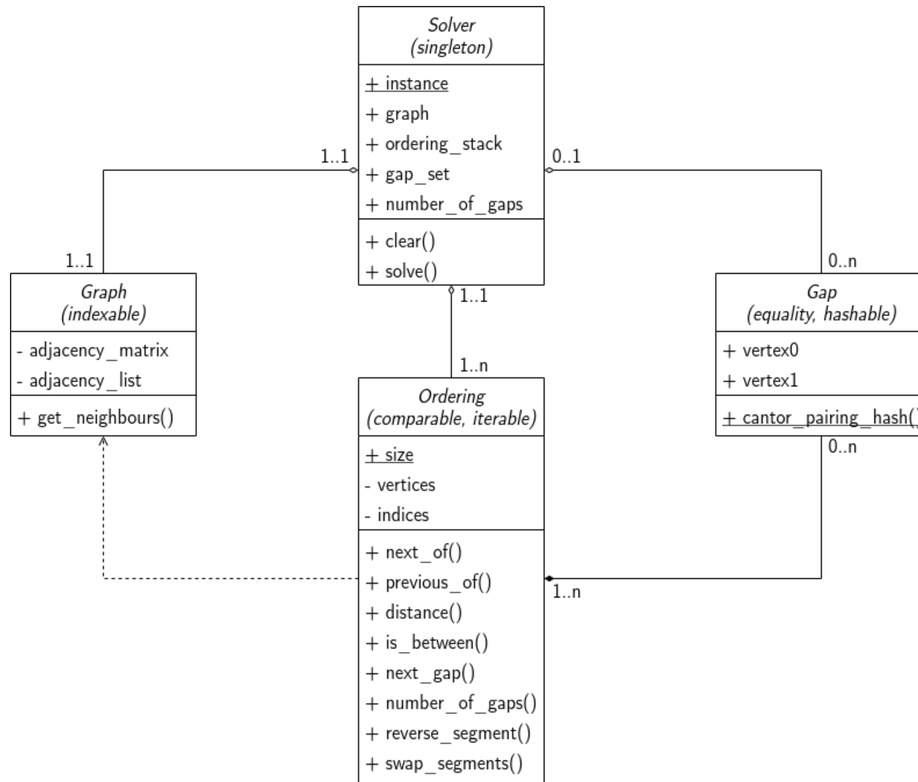


Figure 5.1: Class Diagram showing the relationships, properties, methods and attributes of each of the classes in the implementation.

Notice that there is no class describing vertices but all objects contain a representation of one or multiple of them. As of this implementation the only important information regarding a vertex is its identifier, therefore all vertex references are just unsigned integers. It has been decided that two bytes will provide a good balance of memory saving and the capacity of supporting graph up to 65535 vertices. The graphs tested identify the vertices starting the count from 1 upwards, however, for the sake of convenience, this algorithm starts from number 0.

5.1.1 Graph

Class representing the undirected and unweighted graph G by the connections between its n vertices. For time optimization purposes, explained further in Section 5.3, both the Adjacency Matrix and the Adjacency List of the graph are stored at the cost of having redundant information.

Attributes

Adjacency Matrix. Square n x n matrix such that the element located at row i and column j indicates if there is a connection between vertices i and j for $0 \leq i, j < n$. Stored as a bidimensional array of bits.

Adjacency List. Collection of lists such that the v-th list contains the neighbors of vertex v for $0 \leq v < n$. Stored as a bidimensional array of unsigned two byte integers.

Properties

Indexable. In order to gain access to the adjacency matrix the class is twice indexable, such that the expression $G[i][j]$ returns a Boolean indicating if there is a connection between vertices i and j for $0 \leq i, j < n$.

Time Complexity: Constant.

Methods

Get the neighbors of a vertex. Returns the list of neighbors for vertex v at the Adjacency List.

Notation: *get_neighbours()*.

Arguments: Vertex v.

Time Complexity: Constant.

5.1.2 Ordering

Class representing an ordering by a particular arrangement of the equivalence class. All instances refer to the same graph G and, as a result, the same number of vertices n.

Attributes

Vertices. Arrangement of vertices. Stored as an array of unsigned two byte integers.

Indices. Position of each of the vertices in the arrangement, stored as an array of unsigned two byte integers, such that the expression $indices[a]$ returns the index of vertex a in the *vertices* array.

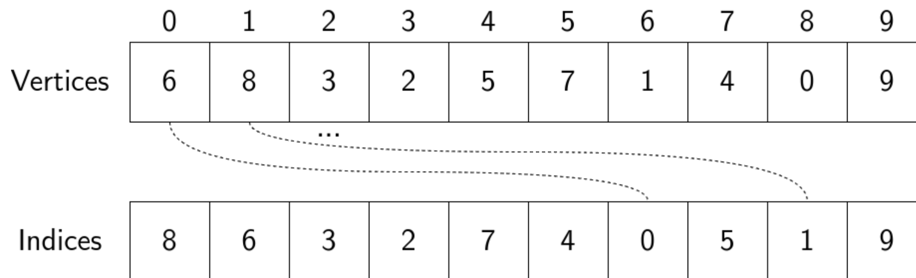


Figure 5.2: Example of the arrays *vertices* and *indices* for the ordering (7,9,4,3, 6,8,2,5,1,10,7). For the sake of clarity, the two arrays can be viewed as mirrors of each other since opposite to the *vertices* array, the *indices* array has the vertices as indices and their positions as values.

Properties

Iterable. In order to iterate over the arrangement and access the vertices, an auxiliary iterator class is implemented. Most importantly, this class supports the cyclic nature of the orderings, internally traversing between both ends of the sequentially stored arrangements, which eases the implementation of otherwise convoluted functions. In addition, the iterator class has the following properties:

- Bidirectional. The iterator can go both in clockwise and counterclockwise directions.
- Random Access. The iterator can be initialized and displaced to any position in the arrangement.
- Deference. The iterator has access to the value of the vertex it points to.
- Equality. Two iterators are the same if they point to the same vertex.

Comparable. Enables a binary search in an ordered container of orderings. As previously described, two different arrangements belong to the same equivalence class if either one can be transformed to the other using a reversal, rotation or both.

However, since looking for such transformation has no optimal implementation the following method is proposed in order to compare two instances:

1. For both orderings, create an iterator at the position of the same starting vertex. This starting vertex could be any in the arrangement but it needs to be fixed for a consistent comparison criterion. Since all graphs have at least one vertex, vertex 1 is recommended.
2. Compare the vertices adjacent to the starting one to determine the direction of the iterators; if the previous is greater than the next the iterator moves clockwise, counterclockwise otherwise.

3. The vertices pointed by both iterators are compared:
   (a) If the first is greater than the second, the first ordering is also considered greater than the second one.
   (b) If the first is lesser than the second, the first orderings is also considered lesser than the second one.
   (c) If they are the same vertex, shift both iterators in their respective direction and return to step 3.
4. If all vertices are equal the arrangements belong to the same equivalence class.

Time Complexity: Making a single comparison between two ordering is linear on the number of vertices n. Therefore, a binary search in an ordered container of orderings requires $O(n\ log(n))$.

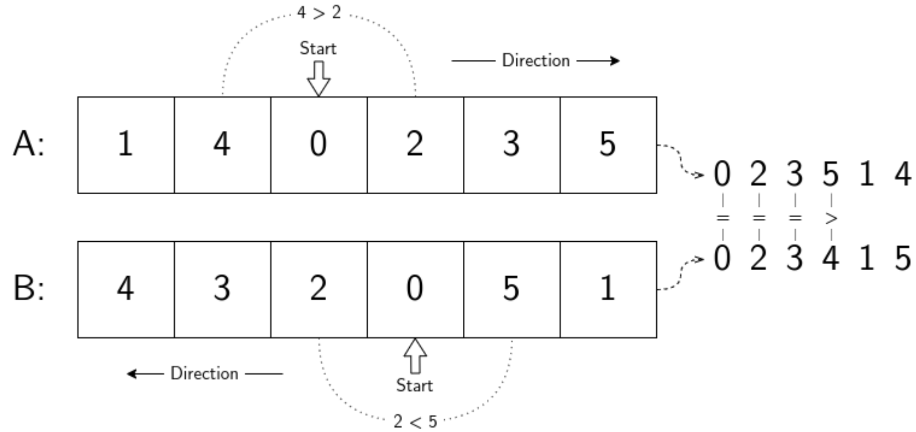


Figure 5.3: Comparison of arrangements A (2,5,1,3,4,6,2) and B (5,4,3,1,6,2,5), where $A > B$.

Methods

Previous vertex of another. Returns the previous adjacent vertex on the arrangement of vertex v in clockwise direction.

Notation: *previous_of()*.

Arguments: Vertex v.

Given by the expression:

$$vertices[(n + indices[v] - 1) \mod n]$$

Time Complexity: Constant.

Subsequent vertex of another. Returns the next adjacent vertex on the arrangement of vertex v in clockwise direction.

Notation: *next_of()*.

Arguments: Vertex v.

Given by the expression:

$$vertices[(indices[v] + 1) \mod n]$$

Time Complexity: Constant.

Distance between two vertices. Returns the number of vertices on the arrangement between vertices v_0 and v_1, non inclusive and in clockwise direction.

Notation: *distance()*.

Arguments: Vertices v_0 and v_1.

Given by the expression:

$$(n + indices[v_0] - indices[v_1] - 1) \mod n$$

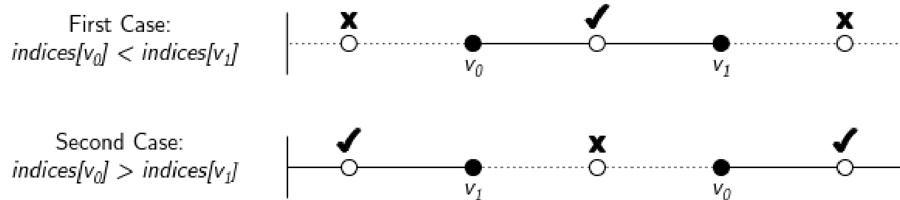
Time Complexity: Constant.

Check if a vertex is between other two. Return whether or not vertex v is between vertices v_0 and v_1 in the arrangement in clockwise direction.

Notation: *is_between()*.

Arguments: Vertices v, v_0 and v_0.

Two possible cases are recognized depending on the position of initial and final vertices, v_0 and v_1 respectively. The considered vertex v has to be contained in the segment $(next_of(v_0), ..., previous_of(v_1))$, which is contiguous in the *vertices* array if the index of v_0 is less than the index of v_1. Otherwise, the segment will be divided, occupying both ends of the *vertices* array.



The following Boolean expression is proposed:

$$\underbrace{(A \land C \land D)}_{\text{First Case}} \lor \underbrace{(B \land (C \lor D))}_{\text{Second Case}}$$

Where:

- $A : indices[v_0] < indices[v_1]$
- $B : indices[v_0] > indices[v_1]$
- $C : indices[v] < indices[v_0]$
- $D : indices[v] > indices[v_1]$

Time Complexity: Constant.

Next gap given a position. Returns the next gap in the arrangement in clockwise direction starting from a position given by an iterator. If no iterator is passed the position is assumed to be the start of the arrangement. The purpose of the iterator is not only to indicate the position, but also to store it between this method's calls to efficiently iterate over gaps.

Notation: *next_gap()*.

Arguments: Optional iterator.

The following implementation is proposed:

1. Store the vertex pointed by the iterator.
2. Shift the iterator in clockwise direction.
3. If the vertex currently pointed by the iterator and the previously stored are not connected in the graph G, return the gap containing these two vertices, go back to step 1 otherwise.
4. If the end of the iterator is reached there is no gap.

Time Complexity: Linear on the number of vertices n.

Number of gaps. Returns the amount of gaps in the ordering.

Notation: *number_of_gaps()*.

The following implementation is proposed:

1. Initialize a counter to 0.
2. Initialize an iterator at the start of the arrangement.
3. Call the method *next_gap()* providing the iterator as an argument. If a gap is returned, increment the the counter by 1 and repeat this step.
4. Return the counter.

Time complexity: Linear on the number of vertices n.

Reverse Segment. The given segment of the arrangement is reversed. This segment is defined by its initial and final vertices, v_0 and v_1 respectively, in clockwise direction.

Notation: *reverse_segment()*.

Arguments: Vertices v_0 and v_1.

The conventional method of gradually swapping the elements at both extremes of the collection until they meet at the middle is used. However, an ordering is cyclic so the segment could be split in the *vertices* array and, in addition, the *indices* array also needs to be modified so it reflects the changes made to the ordering. Therefore, the following version is proposed:

1. Create two iterators, the first moving in clockwise direction and positioned at v_0, the second in counterclockwise direction and at v_1.
2. Repeat the number of times given by the expression:

$$\lfloor distance(v_0, v_1)/2 \rfloor + 1$$

   (a) Swap the positions of the vertices pointed by the iterators in the *indices* array.

   (b) Swap the vertices pointed by the iterators in the *vertices* array.

   (c) Shift both iterators in their respective directions.

Time Complexity: Linear on the number of vertices n.

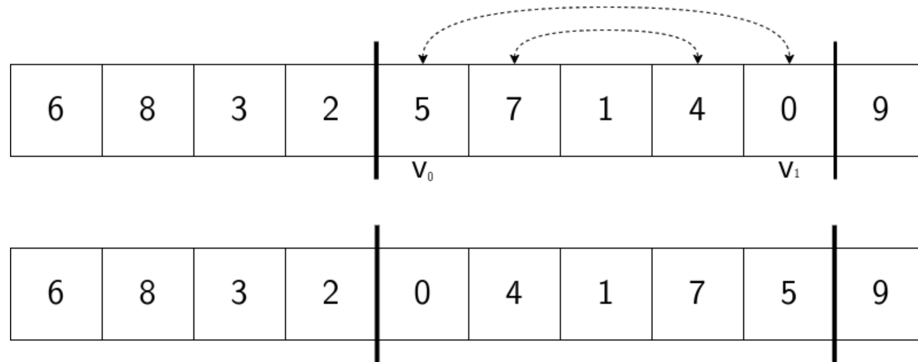


Figure 5.4: Effect of a segment reversal on the *vertices* array.

Swap Segments. Exchange the placement of two non overlapping segments in the arrangement. The segments are defined by their initial and final vertices in clockwise direction.

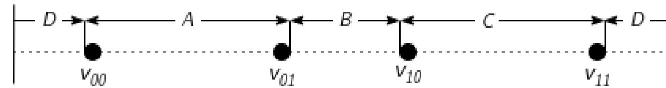
Notation: *swap_segments()*.

Arguments: Initial vertex of the first segment v_{00}, final vertex of the first segment v_{01}, initial vertex of the second vertex v_{10} and final vertex of the second segment v_{11}.

Precondition: The proposed implementation requires the segments not to intersect. However, the functions making use of this method, discussed in Section 5.3, intrinsically assure that the provided vertices are correctly positioned.

The ordering is decomposed in four segments:

- First Segment A: $(v_{00}, ..., v_{01})$
- Central Segment B: $(next_of(v_{01}), ..., previous_of(v_{10}))$
- Second Segment C: $(v_{10}, ..., v_{11})$
- Final Segment D: $(next_of(v_{11}), ..., previous_of(v_{00}))$



Then, the ordering is rearranged as (C, B, A, D), keeping in mind that most likely one of the segments will not be sequentially stored in the *vertices* array.

Time Complexity: Linear on the number of vertices n.

Algorithm 1 Swap Segments

```
 1: function SWAP_SEGMENTS(v_00, v_01, v_10, v_11)
 2:     new_vertices ← ARRAY[0 : n − 1]
 3:     i ← 0
                                                        ▷ Second Segment
 4:     itr ← iterator(indices[v_10])
 5:     end ← iterator(indices[next_of(v_11)])
 6:     WRITE_TO_NEW_VERTICES
                                                        ▷ Central Segment
 7:     itr ← iterator(indices[next_of(v_01)])
 8:     end ← iterator(indices[v_10])
 9:     WRITE_TO_NEW_VERTICES
                                                        ▷ First Segment
10:     itr ← iterator(indices[v_00])
11:     end ← iterator(indices[next_of(v_01)])
12:     WRITE_TO_NEW_VERTICES
                                                        ▷ Final Segment
13:     itr ← iterator(indices[next_of(v_11)])
14:     end ← iterator(indices[v_00])
15:     WRITE_TO_NEW_VERTICES

16:     vertices ← new_vertices                 ▷ Update the vertices array
                                                ▷ Update the indices array
17:     for (i ← 0; i < n; i ← i + 1) do
18:         indices[vertices[i]] ← i

19: procedure WRITE_TO_NEW_VERTICES
20:     while itr ≠ end do
21:         new_vertices[i] ← itr.pointed_vertex
22:         itr ← itr + 1
23:         i ← i + 1
```

5.1.3 Gap

Representation of two adjacent vertices in an ordering, which are not connected in the graph G.

Attributes

V_0. First vertex.

V_1. Second vertex.

Properties

Equality. Two gaps are said to be the same if they contain the same pair of vertices, whatever the order. To test the equality of gaps g_0 and g_1 the following expression is proposed:

$$(g_0.v_0 = g_1.v_0 \wedge g_0.v_1 = g_1.v_1) \vee (g_0.v_0 = g_1.v_1 \wedge g_0.v_1 = g_1.g_0)$$

Time Complexity: Constant.

Hashable. Enables the fast search and retrieval in hash based containers. The *Cantor* pairing function is used [11]:

$$Cantor : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$Cantor(x, y) = \frac{1}{2}(x + y)(x + y + 1) + y$$

Applied to a gap, the following criterion is used:

- $x = max(v_0, v_1)$
- $y = min(v_0, v_1)$

Since the *Cantor* pairing function is bijective, it is also a perfect hash function with no possible hash collision.

Time Complexity: Constant for both computing the pairing function and searching a gap in a hash based container.

5.1.4 Solver

Attributes

Graph. Instance of class *Graph*, representing graph G.

Ordering Stack. LIFO container of instances of class *Ordering*.

Gap Set. Hash based container of unique instances of class *Gap*.

Number of Gaps. Unsigned two byte integer storing the minimum number of gaps found in an ordering.

Properties

Singleton. Restricted instantiation of the class to a single instance for easier access.

Methods

Clear. Removes every single ordering from the ordering stack except the one at the top and empties the gap set.

Notation: *clear()*.

Solve. Runs the SLH algorithm controlling the logic flow between the different stages.

Time Complexity: $O(n^6\, log(n)\, k^4)$, where n is the number of vertices and k is the maximum degree of graph G. However, this is worst case scenario and it is highly unlikely. $O(n^5\, log(n)\, k^4)$ is given as the more reasonable time complexity, or even $O(n^4\, k^4)$ if the graph solved at the early stages of the algorithm. See Section 5.4 for a detailed explanation.

5.2 Isomorphisms

Representative of the generative transformations [see 4, page 4], these are the only processes that perform transformations over orderings. Implemented as blind functions which do not check any condition, they just change the configuration of the vertices in an ordering, making direct use of the *Ordering* class methods *reverse_segments()* and *swap_segments()*. As the actual isomorphisms, these functions can be applied sequentially to form complex transformations.

Isomorphism γ

Given an ordering C and two of its vertices x and b, the segment $(x, ..., b)$ is reversed. In other words, considering the vertices $y \leftarrow previous_of(x)$ and $a \leftarrow next_of(b)$ and the segments $A \leftarrow (x, ..., b)$ and $B \leftarrow (a, ..., y)$, the ordering $C = (x, ..., b, a, ...y, x)$ or (A, B) is transformed into $(b, ..., x, a, ..., y, b)$ or (A^R, B), see Figure 4.3.

Arguments: Ordering C and vertices x and b.

Algorithm 2 Isomorphism γ

```
1: function ISOMORPHISM_γ(C, x, b)
2:     C.reverse_segment(x, b)
```

The function internally calls *Ordering* class method *reverse_segment()*, therefore, keep in mind that the reversed segment is defined in clockwise direction. However, if the ordering is being traversed in counterclockwise direction the

correct transformation can be achieved by simply interchanging vertices x and b at the time of calling the function.

Time Complexity: Linear in the number of vertices n.

Isomorphism $\aleph$

Given ordering C and its vertices x, e, c, a, b, f, d and y, the following segments are considered:

- $A \leftarrow (x, ..., e)$
- $B \leftarrow (c, ..., a)$
- $C \leftarrow (b, ..., f)$
- $D \leftarrow (d, ..., y)$

The ordering $C = (x, ..., e, c, ..., a, b, ..., f, d, ..., y)$ or (A, B, C, D) is transformed into the ordering $(e, ..., x, a, ..., c, d, ..., y, f, ..., b, e)$ or (A^R, B^R, D, C^R), see Figure 4.4

Arguments: Ordering C and vertices x, e, c, a, b, f, d and y.

Algorithm 3 Isomorphism $\aleph$

```
1: function ISOMORPHISM_ℵ(C, x, e, c, a, b, f, d, y)
2:     C.swap_segments(b, f, d, y)
3:     C.reverse_segment(x, e)
4:     C.reverse_segment(c, a)
5:     C.reverse_segment(b, f)
```

Since multiple of the used vertices are adjacent, it can be argued that there is redundant information passed into the function. On the other hand, because of the way it has been implemented, the processes making use of this function have already acquired the values of all the vertices and it has been considered that there is no need to compute them again.

Just as isomorphism γ function, the vertices must be given in clockwise direction to achieve the desired transformation. If C is being traversed in counterclockwise direction instead, the vertices defining segments A, B, C and D have to be passed swapped. Keeping the values of the vertices as they are, the function would be called in the following manner:

$$\text{ISOMORPHISM_}\aleph(C, e, x, a, c, f, b, y, d)$$

Time Complexity: Linear on the number of vertices n.

5.3 Transformations

A special combination of isomorphisms γ and $\aleph$, each of these complex transformations requires a different set occurrences involving snakes, ladders and gaps to be present in an ordering so it can be transformed. They are implemented as searching algorithms which look for specific conditions and leave the actual conversion of the ordering to the isomorphism functions discussed in the previous section.

The implementations of these transformations, even if they search for a distinct set of conditions, all follow the same structure and utilize a defined set of techniques to navigate the orderings and check that the conditions are met. Consequently, just Algorithm 4 containing the description for the floating transformation "4-flo type 1" is included, see Figure 5.5. This algorithm is an excellent reference for the rest of the transformations since it makes use of all the aforementioned techniques.

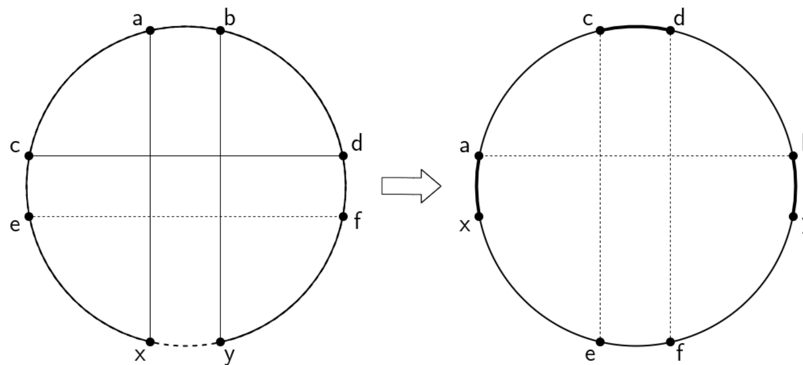


Figure 5.5: Floating 4-flo type 1 transformation.

However, before going any further, an important distinction has to be made. All of the transformation functions are provided with an ordering C and one of its gaps g. C is the ordering to be searched and g sets the point from where the algorithm will start looking, but given that C can be traversed in two different directions, also two types of transformations are identified:

- **Symmetric**: Those transformations that thanks to the conditions they require, if there is a set of vertices which fulfills them in C, then it will be found no matter the direction. The floating transformation "4-flo type 1" falls into this category.
- **Asymmetric**: Contrary to the symmetric transformations, C has to be searched in both directions in order to check if there is any set of vertices which meets the imposed conditions.

Taking this into account, the algorithm sequentially tries to search for a series of vertices, using the already established vertices and the given conditions as

Algorithm 4 Floating 4-flo type 1

```
1: function FLOATING_4FLO_TYPE1(C, g)
2:     G ← Solver.instance.graph
3:     gS ← Solver.instance.gap_set

4:     x ← g.v1
5:     y ← g.v0

6:     for a in G.get_neighbours(x) do
7:         b ← next_of(a)

8:         if G[y][b] ∧ C.distance(x, a) ≥ 2 ∧ C.distance(b, y) ≥ 2 then
9:             itr ← Iterator(C.indices[x]) + 2
10:            end ← Iterator(C.indices[a])

11:            for (; itr ≠ end; itr ← itr + 1) do
12:                c ← itr.pointed_vertex

13:                for d in G.get_neighbours(c) do
14:                    if C.is_between(d, b, y) then
15:                        e ← C.previous_of(c)
16:                        f ← C.next_of(d)

17:                        if y ≠ f ∧ ¬(Gap(e, f) in gS) then
18:                            opt ← 0
19:                            if G[e][f] then opt ← opt + 1
20:                            if ¬G[c][e] then opt ← opt + 1
21:                            if ¬G[a][b] then opt ← opt + 1
22:                            if ¬G[d][f] then opt ← opt + 1

23:                            C′ ← SHALLOW_COPY(C)
24:                            ISOMORPHISM_γ(C′, x, e)
25:                            ISOMORPHISM_γ(C′, c, a)
26:                            ISOMORPHISM_γ(C′, b, e)
27:                            ISOMORPHISM_γ(C′, f, y)

28:                            return opt, C′

29:    return 0, NULL
```

guides to find the next vertex. The “anchor points”, as they will be called, are just the vertices which for a given state of the algorithm have already been set. These anchor points, when possible, are used in conjunction with the required snakes, ladders and gaps to obtain a new anchor point. If all anchor points have been positioned, then a set of vertices fulfilling the conditions exists and the transformation can proceed.

The techniques or methods used to navigate the orderings in either direction fall into these three categories:

1. **Adjacency in graph *G***: Provides the information regarding the existence or absence of edges between vertices.

   (a) Adjacency List: Allows to loop trough all the ladders of an anchor point. This can be seen in Algorithm 4 line 6, since there is a ladder $(x-a)$ and vertex x, being part of g, is an anchor point, the algorithm iterates over all possible ladders of x in order to set vertex a.

   (b) Adjacency Matrix: Returns in constant time whether or not two vertices are connected in G in order to check the existence of snakes, ladders and gaps.

   Having both representations of graph G implies that there is redundant information stored. On the other hand, not having available one of the two means a suboptimal search. In the lack of the adjacency list, all n vertices would have to be considered as possible ladders, and in the absence of the adjacency matrix, testing if two vertex are connected in G would cost $O(log(n))$ at best if the neighbours of each vertex are stored in orderly manner.

2. **Ordering Iterators**: Used as a last resort, they are necessary when the next vertex to be found has no related anchor point. This situation can be seen at Algorithm 4, lines 9-11, where vertices x, y, a and b have already been set but not one of vertices c, d, e and f has a snake, ladder or gap with any of the anchor points. Consequently, the vertices of an entire segment have to be observed, in this case segment $(x, .., a)$ in search for vertex c.

3. **Relative position in ordering *C***: Provide early stopping conditions and most importantly assure that the anchor points are correctly positioned between each other, leaving no room to contradictions.

   (a) Previous and next of a vertex: They easily obtain the adjacent vertex of an anchor point in C. See Algorithm 4, line 7, where vertex b is just the next of a, the previous in the case of the direction being counterclockwise.

   (b) Distance between vertices: Considering the anchor points, it is used to determine if the conditions are still feasible. See Algorithm 4, line 8, once vertices a and b have been set there is no reason to continue if

in the segments (x,...,a) and (b,...,y) contain less than four vertices, since vertices e, c and d, f respectively, also have to be included.

(c) Vertex contained within a segment: Used in conjunction with technique 1.a when the vertex at the end of a ladder starting from an anchor point has to be contained in a certain segment. See Algorithm 4, line 14, where considering the ladder $(c-d)$ and the anchor point c the algorithm is trying to set vertex d. However, vertex d has to be included in segment $(b, ..., y)$ or otherwise it will be discarded.

Depending on the minimum number of gaps potentially closed there are three types of transformations: "closing", "floating" and "opening" transformations. Furthermore, the "stages" defined at Section 5.4 do not ask for a particular transformation, these functions request for any applicable transformation of one given type instead, just specifying the ordering to transform. This means that for all gaps in the ordering and for all transformations falling in the given type, any combination could be potentially applicable.

It is not specified the priority in which the transformations are tested, nor if all the transformations are tested before continuing to the next gap or the other way around. The approach taken in this implementation is based on one of the principles adopted in [3, page 38], which gives priority to the transformations of lower computational cost. For that matter, the implementation iterates over the transformations of the specified type, ordered from lower to higher computational cost, and tries to apply the selected transformation with all the gaps in the ordering until an applicable combination is found, or going to the next transformation otherwise. Notice that this approach is just one of the possible among others and it could not be the same as the authors used.

5.3.1 Closing Transformations

These transformations guarantee that at least one gap is closed after the conversion of the ordering. Used in "Stage 0", see Section 5.4.1, this process does not maintain a collection of visited orderings, therefore, all transformations are applied over the same *Ordering* object instance C.

Arguments: Ordering C and gap g.

Returns: Boolean type indicating whether or not the transformation has been applied.

Transformation	**Symmetric**	**Asymptotic Notation**
2-opt type 1	No	$O(n+k)$
2-opt type 2	Yes	$O(n+k)$
3-opt	Yes	$O(n+k^2)$

Table 5.1: SLH closing transformations in order of priority. Where k refers to the maximum degree of graph G.

5.3.2 Floating Transformations

If one of these transformations is applicable to an ordering C, the resulting ordering C' will have the same number of gaps or less than C.

Any transformation can be seen as an exchange of gaps, for example taking a look at floating transformation "4-flo type 1" in Figure 5.5, gap $(x|y)$ is exchanged for a possible gap (e, f). Floating transformations require one additional condition, being that the exchanged gap has not been already subject of a previous transformation, see Algorithm 4, line 17. For that matter, if the transformation is still applicable, gap g will be added to a hashed container so the transformations to come do not produce it again.

There is a variation of the floating transformations used in "Stage 3", see Section 5.4.4, which requires at least one gap to be closed, just as the closing transformations. This is easily implemented imposing variable *opt* in Algorithm 4, line 18, to be greater that 0.

Compared to the original floating transformations, even if they no longer impose restrictions in the exchanged gap, the resulting ordering C' cannot have been previously visited, adding a time complexity of $O(n\ log(n))$, result of a binary search in an ordered container, which needs to be given as a parameter, plus the time it takes to compare two orderings.

Arguments: Ordering C, gap g and, optionally, an ordered container of visited orderings.

Returns: The number of gaps closed and ordering C'.

Transformation	**Symmetric**	**Asymptotic Notation**	
		flo	opt
2-flo	No	$O(n+k)$	$O(n\ log(n)\ k)$
3-flo	-	$O(n+k^2)$	$O(n\ log(n)\ k^2)$
4-flo type 1	Yes	$O(n+k^2\ s)$	$O(n\ log(n)\ k^2\ s)$
4-flo type 2	No	$O(n+k^3)$	$O(n\ log(n)\ k^3)$
5-flo	No	$O(n+k^4)$	$O(n\ log(n)\ k^4)$

Table 5.2: SLH floating transformations in order of priority. Where k is the maximum degree of gap graph G and s is the length of segment $(x, ..., a)$, see Figure 5.5 and Algorithm 4, lines 9-11.

Floating transformation "3-flo" is a special transformation in the sense that it can be symmetric and asymmetric depending on the considered ladder, $(y-b)$ or $(c-d)$ respectively [see 4, page 8].

5.3.3 Opening Transformations

Transformations that potentially increase the number of gaps by one. Aside from ordering C and one of its gaps g, they also require an ordered container of orderings to be passed as an argument, with the intention of not repeating already visited orderings. The collection can be just the orderings product of other opening transformations as it is used in “Stage 2”, see Section 5.4.3, or the entire list of visited orderings as used in “Stage 3”, see Section 5.4.4.

Arguments: Ordering C, gap g and an ordered container of orderings.

Returns: The number of gaps closed and the resulting ordering C'.

Transformation	**Symmetric**	**Asymptotic Notation**
4-flo	No	$O(n\ log(n)\ k^2\ s)$

Table 5.3: SLH opening transformation. Where k is the maximum degree of graph G and s is the length of segment $(x, ..., a)$ [see 4, page 9].

5.4 Stages

The SLH algorithm is formed by four main processes called “Stages” which try to solve the HCP using different approaches. They transition between each other until ultimately an ordering containing 0 gaps is discovered or n^3 orderings have been considered. In either case, the ordering with the least amount of gaps found is returned. The State Diagram explaining the transition between the different stages is shown in Figure 5.6 and the detailed explanation is available at [4, page 10].

The worst time complexity of the complete process is $O(n^6\ log(n)\ k^4)$, given by “Stage 3”, the stage with the highest computational cost $O(n^5\ log(n)\ k^4)$, and the n times “Stage 1” though “Stage 3” can be repeated, the latter requiring to find an ordering with less gaps than any previous ordering found before in order to return to the first.

However, tests show that “Stage 3” is repeated few times, since by the moment the algorithm reaches this stage most gaps have been already closed, therefore a more reasonable time complexity would be $O(n^5\ log(n)\ k^4)$. Furthermore, most instances are solved without ever reaching “Stage 2”, just requiring $O(n^4\ k^4)$, the time complexity of “Stage 1”.

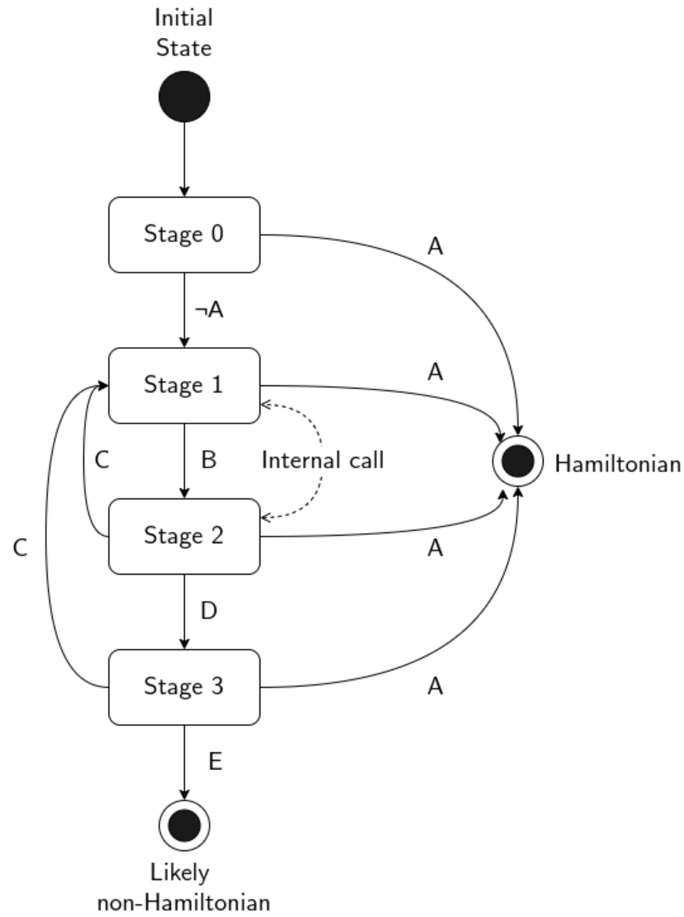


Figure 5.6: State Diagram showing the transition between the different stages of SLH. The conditions being as follows: A) An ordering with 0 gaps found. B) The number of orderings in the stack is greater than n^2 or no more floating transformations can be performed. C) An ordering with a new minimum number of gaps found. D) All opening transformations considering the ordering at the top of the stack and one of its gaps have been considered. E) The number of orderings in the stack is greater that n^3 or the ordering at top of the stack has no possible opening transformation.

5.4.1 Stage 0

As many gaps as possible are closed just using closing transformations over the same *Ordering* object instance C. Once no more closing transformations can be performed, the number of gaps is updated.

A maximum of n closing transformations will be performed, since they close at least one gap, and ordering C at any given time can have as many as n gaps from where the transformations can be applied. The closing transformation "3-opt" requires two anchor points to be established, and if each vertex has a maximum of k neighbours in G then there are k^2 potential combinations. Therefore, this stage has a worst case time complexity of $O(n^2\ k^2)$.

Algorithm 5 Stage 0 of SLH

```
1: function STAGE0()
2:     S ← Solver.instance
3:     C ← S.ordering_stack.top()
4:     transformation_made ← closing_transformation(C)

5:     while transformation_made do
6:         transformation_made ← closing_transformation(C)

7:     S.number_of_gaps ← C.number_of_gaps()
```

5.4.2 Stage 1

Just using floating transformations this stage tries to close gaps using a lateral search. The number of ordering containing the same number of gaps is limited to n^2, but each time an ordering with less gaps is found, the ordering stack is emptied. This can happen a maximum of n times. The floating transformation "5-flo" requires 4 anchor points to be set, if each vertex has as many as k neighbours in G and if each ordering has up to n gaps from where a transformation can be applied, there are nk^4 potential combinations. Therefore, this stage has a worst case time complexity of $O(n^4\ k^4)$.

5.4.3 Stage 2

This process takes the ordering with the least amount of gaps, performs opening transformations using a single gap and delegates the control over again to "Stage 1". This can be interpreted as the algorithm trying to find new pathways in order to continue the search.

If "Stage 1" can create up to n^2 new orderings, the amount of performed opening transformations is limited to n or otherwise more than n^3 orderings would be created. Therefore, if "Stage 1" with $O(n^4\ k^4)$ can only be repeated a maximum of n times, this stage has a worst case time complexity of $O(n^5\ k^4)$.

Algorithm 6 Stage 1 of SLH

```
1: function STAGE1()
2:     S ← Solver.instance
3:     CS ← S.ordering_stack
4:     auxiliary_CS ← SHALLOW_COPY(CS)

5:     while CS.size() ≤ n^2 ∧ auxiliary_CS.size() > 0 do
6:         opt, C ← FLOATING_TRANSFORMATION(auxiliary_CS.top())

7:         if C ≠ NULL then
8:             CS.push(C)

9:             if opt > 0 then
10:                S.number_of_gaps ← S.number_of_gaps − opt
11:                if S.number_of_gaps = 0 then return
12:                S.clear()
13:                auxiliary_CS.clear()

14:            auxiliary_CS.push(C)

15:        else auxiliary_CS.pop()
```

5.4.4 Stage 3

An opening transformation on the ordering at the top of the ordering stack is followed by a sequence of all possible floating transformations which close gaps. This is repeated until an ordering with a new minimum number of gaps has been found or n^3 orderings have been considered. Each new ordering is required not to be already visited, requiring $O(log(n))$ to perform a binary search in an ordered container and $O(n)$ to compare two orderings. The floating transformation "5-flo" requires 4 anchor points to be set, if each vertex has as many as k neighbours in G and if each ordering has up to n gaps from where a transformation can be applied, there are nk^4 potential combinations. Therefore, this stage has a worst case time complexity of $O(n^5 \; log(n) \; k^4)$.

Algorithm 7 Stage 2 of SLH

```
1: function STAGE2()
2:     S ← Solver.instance
3:     C ← S.ordering_stack.pop()
4:     g_C ← S.number_of_gaps
5:     g ← C.next_gap()
6:     visited_C ← SORTED_SET

7:     opt, C′ ← OPENING_4FLO_TRANSFORMATION(C, g, visited_C)
8:     while C′ ≠ NULL do
9:         S.ordering_stack.push(C′)
10:        S.clear()
11:        S.gap_set.insert(g)
12:        S.number_of_gaps ← S.number_of_gaps − opt
13:        STAGE1()

14:        if S.number_of_gaps < g_C then
15:            if S.number_of_gaps = 0 then return FALSE
16:            S.clear()
17:            return TRUE

18:        visited_C.insert(C′)
19:        opt, C′ ← OPENING_4FLO_TRANSFORMATION(C, g, visited_C)

20:    S.ordering_stack.push(C)
21:    S.number_of_gaps ← g_C
22:    return FALSE
```

Algorithm 8 Stage 3 of SLH

```
1: function STAGE3()
2:     S ← Solver.instance
3:     CS ← S.ordering_stack
4:     visited_C ← SORTED_SET(CS)

5:     while CS.size() ≤ n^3 do
6:         C ← OPENING_TRANSFORMATION(CS.top(), visited_C)
7:         if C = NULL then break

8:         auxiliary_CS ← STACK
9:         auxiliary_g_CS ← STACK

10:        CS.push(C)
11:        visited_C.insert(C)
12:        auxiliary_CS.push(C)
13:        auxiliary_g_CS.push(C.number_of_gaps())

14:        while CS.size() ≤ n^3 ∧ auxiliary_CS.size() > 0 do
15:            opt, C ← OPT_TRANSFORMATION(auxiliary_CS.top(), visited_C)

16:            if C ≠ NULL then
17:                CS.push(C)
18:                visited_C.insert(C)
19:                auxiliary_CS.push(C)
20:                auxiliary_g_CS.push(auxiliary_g_CS.top() − opt)

21:            else if auxiliary_g_CS.top() < S.number_of_gaps then
22:                S.number_of_gaps ← auxiliary_g_CS.top()
23:                if auxiliary_g_CS.top() = 0 then return FALSE
24:                S.clear()
25:                return TRUE

26:            else
27:                auxiliary_CS.pop()
28:                auxiliary_g_CS.pop()

29:    return FALSE
```

5.5 Run Time

At the time of executing the program it is required to provide a file with ".hcp" extension containing the representation of the graph to be solved as a list of edges. Once the graph has been specified the program's life cycle is formed by a simple sequence of subprocesses:

1. **Parsing:** The ".hcp" file is parsed as an instance of class *Graph*.
2. **Initialization:** The singleton instance of class *Solver* is created with the instance of class *Graph* generated by subprocess 1, the number of vertices for all the objects of class *Ordering* is set and the ordering containing the initial assignment of the vertices is constructed and pushed into the ordering stack.
3. **SLH Execution:** Using the implemented version of the SLH, the specified graph is searched with the intention of finding a Hamiltonian cycle.
4. **Output:** The series of vertices contained in the ordering with the least amount of gaps found are returned.
5. **Clean up:** The created objects are destructed if necessary.

5.6 Possible Improvements

During the development of the implementation several improvements were considered, which in the end, were not added to the latest version of the program since they had a trade off, they were inconsistent with original design of SLH or simply, there was a lack of information to support them.

5.6.1 Ordering fingerprint

In [3, page 41, Algorithm 1] the term "fingerprint" is used as an alternative to store the visited orderings in an ordered list. It is inferred that a fingerprint is the minimal expression of an ordering instance, which still maintains the identity of the ordering and it is comparable to another fingerprint.

An example of such element could be achieved based on the *vertices* array attribute and the comparable property of the *Ordering* class in Section 5.1.2:

- Array containing $n-1$ vertices, since it would always start from vertex 1.
- The listing of the vertices would be done using the same criterion as the comparable property of the ordering class, following the direction imposed by the vertex with smaller identifier adjacent to and starting from vertex 1.
- Two fingerprint instances would be compared vertex by vertex, from start to finish, until a pair of vertices returns an inequality or all comparisons are equal.

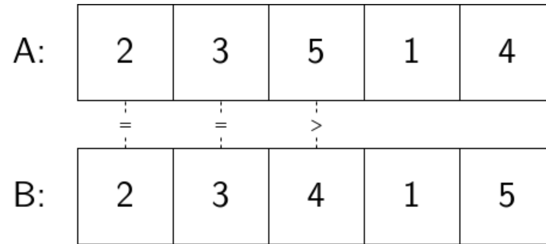


Figure 5.7: Comparison of fingerprints of orderings A (2,5,1,3,4,6,2) and B (5,4,3,1,6,2,5), where $A > B$. Same as Figure 5.3.

Applied to the proposed implementation, it would mean that each time an ordering was recovered it would have to be reconstructed from the fingerprint in $O(n)$ time, and that each time a transformation was made the produced ordering had to be created following the fingerprint listing order.

On the other hand, the space in memory required to run the algorithm would be approximately reduced by half in difficult instances, since just a compressed representation of the orderings would have to be stored instead of the entire instance.

5.6.2 Blind Improvement Space

Explained in detail in [3, page 23], a “Blind Improvement Space” (BIS) is a term used by “k-opt” transformation algorithms referring to a map that associates a non-optimal tour of a graph to a set of transformations which produce at least one improved tour, closer to the optimal. The BIS of SLH is conformed by the set of stages and the order the transformations of each category are applied, having a lack of information regarding the latter.

It is probable that the BIS created by the proposed implementation is not the same as the authors originally intended, which would explain the unreliable results on the last two stages of the algorithm shown in Section 6.2. This would also mean that optimizing the BIS of the implementation could significantly increase its performance in difficult graphs.

5.6.3 Multithreading

The current version of the algorithm is implemented as a single threaded program, however, some segments of the code could be ran in multiple threads as means to increase the computing power. The most notable segments being the searches of applicable transformations for the different categories. If the CPU would allow it, multiple threads could ran in parallel looking for new orderings using different combinations of gaps and transformations.

Notice that in this case, if the first eligible transformation found would be performed, the algorithm would no longer be deterministic, since the output could vary between different executions depending on the state of the CPU and the operating system. Therefore a priority system should be also implemented, one that would make a thread wait if it has found an applicable transformation until the threads considering higher priority gaps and transformations had finished, which would nevertheless speed up the process in some situations.

Chapter 6

Experimental Validation

The goal of this chapter is to evaluate the proposed implementation using different problem benchmarks.

All the times listed in this section where achieved running the C++ version of the implemented SLH running in a custom desktop computer with a Ryzen 7 2700X at 4.3GHz overclocked with Precision Boost Overdrive and 16GB of RAM under Linux Debian 10.

6.1 Benchmarks

The graphs used to test the implementation are obtained from the TSPLIB website's specific HCP section [12], which provides 9 graph of up to 5000 vertices, a large set of 797 symmetric cubic graphs of up to 2048 vertices, accessible at [5] and lastly, from the Flinders HCP Challenge Set [7], a collection of 1001 difficult graphs. All mentioned graphs contain Hamiltonian Cycles with the exception of two symmetric cubic graphs, C10.1 (the Petersen graph) and C28.1 (the Coxeter graph).

6.2 Results

The obtained results obtained in the aforementioned benchmark graph sets are gathered using a series of tables containing the following fields:

- **Graph:** Name or identifier of the graph.
- **Number:** Number of graphs in the benchmark set.
- **Size Range:** Minimum and maximum number of vertices in the graphs of the benchmark set.
- **Time (t):** Time required to solve the graph, *s* of second and *ms* for milliseconds.
- **Stage (stg.):** Stage in which the graph was solved. Even if the algorithm returns to previous stages the furthest reached is considered.
- **Success rate:** Percentage of successfully solved graphs in the benchmark set.

Graphs	Number	Size Range	Time (s)	Success Rate
TSPLIB	9	1000-5000	1.95	100
Symmetric cubic	795	4-2048	24.32	100

Table 6.1: Combined time required to solve all graphs in the TSPLIB challenge set and the 795 symmetric cubic graphs containing Hamiltonian cycles.

After a series of versions and optimizations the proposed SLH implementation has exceeded the expectations regarding the time required to solve graphs with large amount of vertices, solving all symmetric cubic graph in less than half a minute and all the graphs proposed by TSPLIB in less than two second.

Due to its deterministic nature, if the algorithm is initialized with the same initial configuration of vertices, it will always follow the same series of actions and return the same outcome. Therefore, to further test the consistency of the implementation, each graph has been tested ten times using random initial orderings. In Table 6.2 and Table 6.3 the particular results for the graphs with higher count of vertices are listed.

Graph	Time (ms)	Stage	Graph	Time (ms)	Stage
alb1000	23	1	alb3000d	216	1
alb2000	69	1	alb3000e	211	1
alb3000a	155	1	alb4000	353	1
alb3000b	191	1	alb5000	561	1
alb3000c	158	1			

Table 6.2: Average time required for 10 different tests using random initial orderings for all TSPLIB graphs.

Graph	Time (ms)	Stage	Graph	Time (ms)	Stage
C2048.1	46	1	C2048.14	45	1
C2048.2	67	1	C2048.15	60	1
C2048.3	53	1	C2048.16	41	1
C2048.4	57	1	C2048.17	56	1
C2048.5	58	1	C2048.18	54	1
C2048.6	93	1	C2048.19	54	1
C2048.7	52	1	C2048.20	50	1
C2048.8	91	1	C2048.21	62	1
C2048.9	55	1	C2048.22	55	1
C2048.10	47	1	C2048.23	47	1
C2048.11	62	1	C2048.24	55	1
C2048.12	67	1	C2048.25	74	1
C2048.13	62	1			

Table 6.3: Average time required for 10 different tests using random initial orderings for all the symmetric cubic graphs of 2048 vertices.

The results are promising, solving the graph of 5000 nodes in an average time of half a second, however note that all tests have been solved between stages 0-1. In order to evaluate the performance of the remaining stages, graphs in the Flinders HCP Challenge Set are used, these being difficult instances which reliably reach the latter stages of the algorithm.

Graph	Run 1		Run 2		Run 3		Run 4		Run 5	
	t (s)	stg.	t (s)	stg.	t (s)	stg.	t (s)	stg.	t (s)	stg.
1	0.294	2	0.274	2	0.473	2	0.838	2	0.0	1
2	0.0	1	0.0	1	0.0	1	0.0	1	0.07	2
3	2.26	3	1.47	3	1.15	2	0.1	2	2.68	3
4	0.2	2	0.18	2	0.0	1	0.27	2	0.0	1
5	7.29	3	4.56	3	2.04	3	1.64	2	3.79	3
6	0.15	2	0.0	1	1.19	3	0.3	2	0.0	1
7	14.08	3	5.21	3	7.52	3	17.79	3	4.75	3
8	8.5	2	0.0	1	0.0	1	5.61	2	11.85	2
10	0.6	2	0.0	1	4.94	2	3.52	3	0.0	1

Table 6.4: Five distinct runs of the first ten graphs in the Flinders HCP Challenge set using random initial orderings. All instances were successfully solved.

As shown in Table 6.4 the implementation is not reliable solving difficult instances. The algorithms shines in stages 0-1, however, when it transitions to stages 2 or 3 the time required to solve the graph rapidly increases, even more so as the number of vertices grows. In addition, the state reached is highly dependant on the initial configuration of the vertices considered, an obvious example being graph 8 which alternates between nearly instantly being solved and requiring one of the longest time observed.

One of the observations made during multiple runs is that the algorithm often falls to stage 3 when it has just gap left to be closed, meaning that the problem is stuck in this stage until it is being solved, since stage 3 requires at least one gap to be closed in order to transition. This can be, due to multiple reasons, an error in the implementation, a misinterpreted guideline of the official algorithm, a suboptimal search space, *etc.*

Chapter 7

Web Application

As a means to show the internal behaviour of the SLH applied to real and complex graphs, a simple web application with a user interface has been developed. The application allows to drag and drop a ".hcp" file containing the representation of the graph as a list of edges. Afterwards, the given graph is solved using the implementation made of the SLH. The algorithm, modified to periodically reserve the best ordering found at that given time, allows the web application to render the stored ordering using the same visual representation the algorithm was based upon, arranging the sequence of vertices contained in the ordering along the perimeter of a circle and subsequently drawing the snakes, ladders and gaps.

The ladders are painted in orange color, as chords in the circle between two non adjacent vertices in the ordering. The snakes and gaps on the other side, are painted as arcs between all the adjacent vertices, in green and red color respectively, which in the end constitutes the entirety of the circle's perimeter. All in all, the contrast between the colors combined with geometrical properties of some graphs and the patterns produced by the SLH transformations can ultimately produce striking images. Furthermore, the user can go back and forth between distinct visual representation of the orderings, using an image gallery, allowing to see how, over the course of the algorithm, the number of gaps decreases as snakes increase.

The Web application has been built using Flask, HTML, JavaScript and the Bootstrap framework for the CSS styles. The python implementation of the SLH, with the variations already mentioned, is used to solve the HCP instances provided by the ".hcp" files, and the visual representation of the graphs are rendered using the Pillow library.

Unfortunately, due to the space and computational requirements of the SLH implementation, the web application has to remain as a local demonstrative interface. Moreover, the security measures required, given the user input file, and the concurrency of multiple users, distance even further a publicly accessible web site from this project's scope.

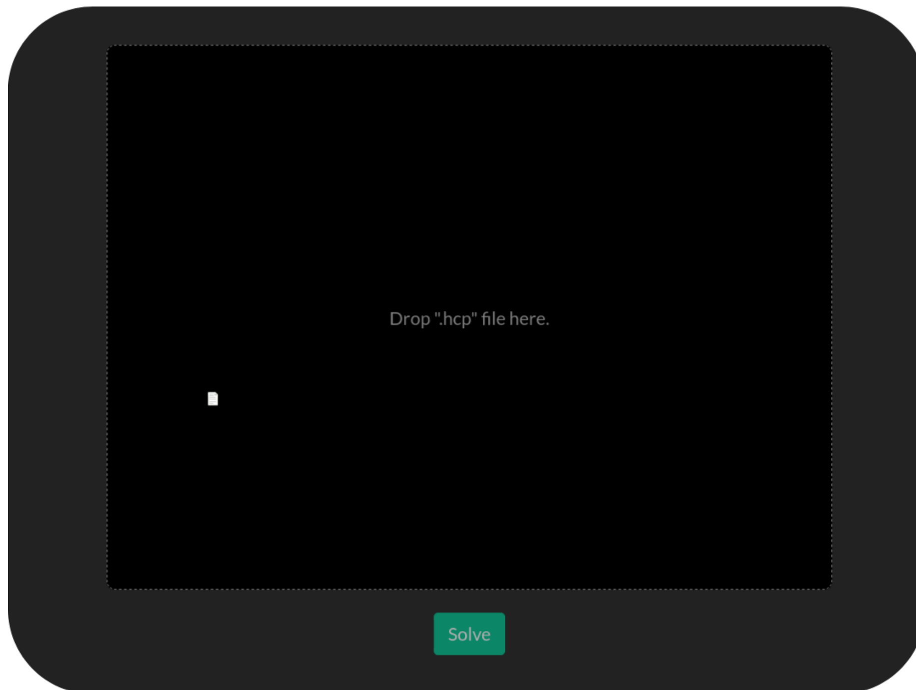


Figure 7.1: SLH Web Application input window.

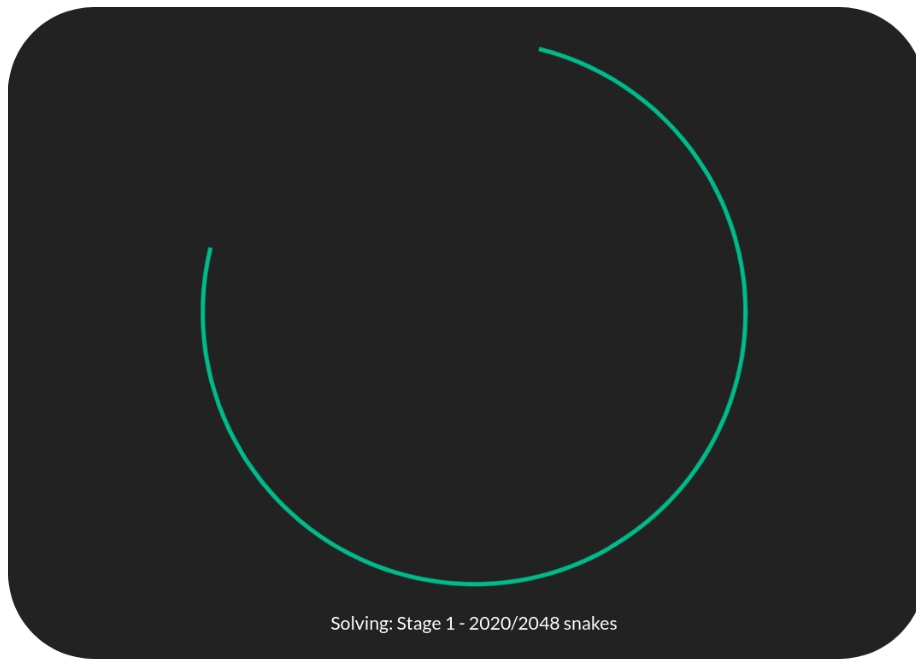


Figure 7.2: SLH Web Application loading screen.

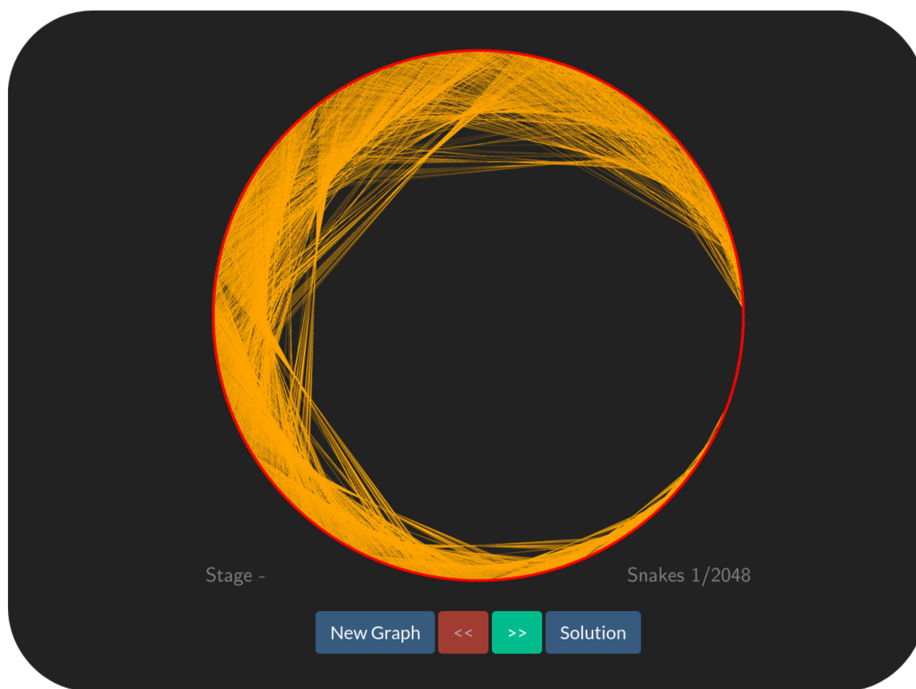


Figure 7.3: SLH Web Application image gallery showing the initial ordering for the symmetric cubic graph C2048.25.

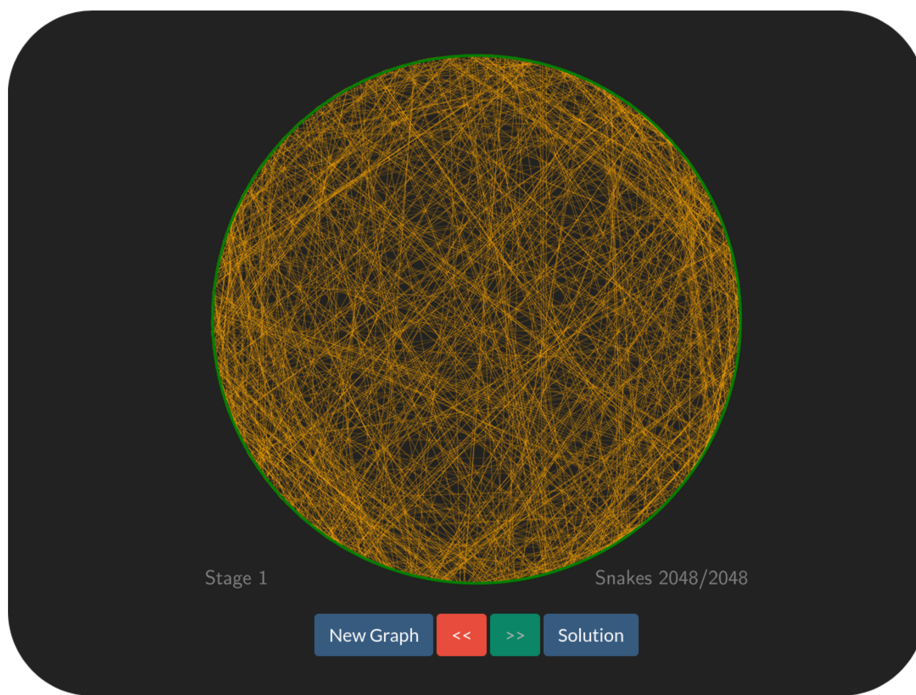


Figure 7.4: SLH Web Application image gallery showing the final ordering of the symmetric cubic graph C2048.25, which contains a Hamiltonian cycle.

Chapter 8

Conclusions

The course of this project has been at all times carried out under the context of the "Snakes and Ladders" Heuristic, a state of the art algorithm for the Hamiltonian Cycle Problem. The main goal was to design and develop a functional implementation of the heuristic, which had to be built from the ground up just with the help of the description provided by the authors.

After meticulous study and a series of optimizations a version of the algorithm with excellent results in graphs of large number of vertices was conceived, which corroborated the incredible potential of the "Snakes and Ladders" Heuristic.

On the other side, there is no denying that the implementation proposed does not reliable solve difficult instances and is highly dependant on the initial configuration of the vertices. However, this is not discouraging, but rather reassures that there are improvements to be made.

The products derived from this project are two implementations, in python and C++, a web application, showing the internal behaviour of the algorithm, and most importantly, a detailed guideline of the designed program, allowing anyone who desires to build it on their own, or even better, improve the given implementation.

Bibliography

[1] David Applegate, William Cook, and André Rohe. "Chained Lin-Kernighan for large traveling salesman problems". In: *INFORMS Journal on Computing* 15.1 (2003), pp. 82–92.

[2] David Applegate et al. *Concorde TSP Solver*. `http://www.math.uwaterloo.ca/tsp/concorde.html`. 2003.

[3] Pouya Baniasadi. "Algorithms for Solving Variations of the Traveling Salesman Problem". PhD thesis. Flinders University, College of Science and Engineering, 2019.

[4] Pouya Baniasadi et al. "Deterministic "Snakes and Ladders" Heuristic for the Hamiltonian cycle problem". In: *Mathematical Programming Computation* 6.1 (2014), pp. 55–75.

[5] M. Conder. *Trivalent (cubic) Symmetric Graphs on Up to 2048 Vertices*. `https://www.math.auckland.ac.nz/~conder/symmcubic2048list.txt`.

[6] David Eppstein. "The traveling salesman problem for cubic graphs". In: (2003), pp. 307–318.

[7] Michael Haythorpe. "FHCP Challenge Set: The first set of structurally difficult instances of the Hamiltonian cycle problem". In: *arXiv preprint arXiv:1902.10352* (2019).

[8] Michael Haythorpe. "Finding Hamiltonian cycles using an interior point method". PhD thesis. Australian Mathematical Society, 2010.

[9] Keld Helsgaun. "An effective implementation of the Lin–Kernighan traveling salesman heuristic". In: *European journal of operational research* 126.1 (2000), pp. 106–130.

[10] Shen Lin and Brian W Kernighan. "An effective heuristic algorithm for the traveling-salesman problem". In: *Operations research* 21.2 (1973), pp. 498–516.

[11] Meri Lisi. "Some remarks on the Cantor pairing function". In: *Le Matematiche* 62.1 (2007), pp. 55–65.

[12] TSPLIB. *Hamiltonian Cycle Problem (HCP)*. `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/hcp/`.