Richard Melvin Krueger
#289, 52465 Range Road 213
Ardrossan, Alberta
Canada, T8G 2E7

Date: ____________

University of Alberta

A Polynomial Time Algorithm for Line Segment Diagram Isomorphism

by

Richard Melvin Krueger

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A Polynomial Time Algorithm for Line Segment Diagram Isomorphism** submitted by Richard Melvin Krueger in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Ryan Hayward
Supervisor

Dr. Martin Müller

Dr. Gerald Cliff

Date:

To everyone who has taught me,
intentionally or not,
thank you.

Abstract

In this thesis we examine the problem of determining whether two board positions in the game of Amazons are equivalent. In particular, we present a polynomial time algorithm for solving this problem.

We start with the notion of a line segment diagram, a combinatorial object which captures the essential information about a board position in Amazons (line segment diagrams were introduced independently by Müller and Tegos in their development of a competitive computer program to play Amazons). A line segment diagram extends the notion of a graph by adding a feature allowing the representation of collinear point sequences.

Based on a planar graph isomorphism algorithm of Hopcroft and Tarjan, we develop a polynomial time algorithm for determining isomorphism of line segment diagrams.

Acknowledgements

I would like to thank my supervisor Dr. Ryan Hayward for introducing this problem to me and engaging in numerous battles over notation, nomenclature and many failed ideas. And thank you to my entire committee for careful reading of this thesis and suggestions for improvement. I thank the University of Alberta, the Government of Alberta and NSERC for financially supporting the work on this thesis.

I thank my parents, Felix and Joan, for raising me, challenging me, and pushing me to succeed in everything. Thank you to all my family and friends who have supported me in whatever I chose to do, and putting up with me while doing it. I couldn't have done it without you!

I thank the faculty, staff and students in the Department of Computing Science for a most enjoyable time at the University. The faculty was excellent and provided a comfortable learning environment. The staff did an outstanding job supporting me and providing anything I might need, whether it was a last minute set of transparencies for a talk or rescheduling my defense due to the September 11th, 2001 terrorist attacks.

I must specially acknowledge Dr. Piotr Rudnicki, an inspiration since my first-year logic class. Never one to accept garbage, he insisted that knowing what you do not know is more important than just knowing, a mantra I hold dear to this day. He was an excellent coach for the ACM Programming Contest team, of which I was proud to be a member competing internationally for two years. I thank the department and Faculty of Science for financially supporting our fun!

But my students were perhaps the most enjoyable of my time in Alberta. I thank the department for allowing me to corrupt their minds, fill them with crazy ideas, and once in a while inspire someone to accomplish something of which they never dreamed, and University Teaching Services for helping me to do it more effectively.

Contents

List of Figures

List of Tables

List of Definitions

List of Theorems

List of Algorithms

Chapter 1

Introduction

Suppose you are playing a game with a friend. You play many games: some games you win, others you lose. But being an astute player, you realize that sometimes you reach a position in the game which you have encountered in a previous game. You can improve your playing by remembering that previous game and recalling what moves were later made, and which player won the game.

This scenario is a standard strategy for human learning, particularly when playing games. It is also a valuable tool for teaching computers to play games, for their memory can be vast and perfect, qualities seldom realized biologically.

This thesis is inspired by such a story, that of a player learning the game of Amazons. The player has realized that a game position can reoccur from previous games (as is the case in almost every game) and perhaps even local positions can reoccur multiple times within a single game. Using the experience of these previous encounters and the knowledge thus gained (whether by playing to game end or by computational simulation), a player can improve his game. This strategy leads not only to making better moves, but also enabling more move possibilities to be analyzed in a short period of time.

The key to such a strategy is discovering when we have repetition, be it an exact copy of a known position or a similar position which will lead to the same results. This thesis will answer the question of "are these positions similar?" with an asymptotically efficient computation. Combinatorial structures will be created to encode the essence of a game position in Amazons, followed by the presentation of a new polynomial time algorithm to determine whether two positions are essentially similar, a concept usually known as isomorphism. Incorporating these developments into a system capable of playing Amazons is not discussed in this thesis, and is left to the experts in heuristic search and related areas.

Chapter Two briefly presents some basic concepts in graph theory and discusses previous work establishing the basis for this problem. Chapter Three discusses the advantages and disadvantages of several combinatorial structures with which a game position can be encoded. In Chapter Four we select a particular structure, the line segment diagram, for which we construct a polynomial time isomorphism algorithm, which is the main result of this thesis. We show that this structure leads to an asymptotically efficient method for determining similarity of Amazons positions. This thesis concludes with a survey of its significant contributions and a discussion of future work which may yield better algorithms for this and related problems. We also discuss the practicality of the presented algorithm in the context of simpler yet theoretically intractable approaches.

1.1 Contributions and Related Work

The problem of testing isomorphism of game positions in Amazons was posed by Theodore Tegos and Martin Müller during their efforts to write programs to play the game well. Through my supervisor, Ryan Hayward, the problem was introduced to me. Its graph theoretic nature was analyzed in collaboration between myself and Ryan Hayward, and much of the early work and characterization is significantly due to his efforts.

Translation of this problem into line segment diagrams was a collaboration between Hayward and myself, though it was independently studied previously (under the slightly different name of line segment graph) by Müller and Tegos [MT01]. The representation of such diagrams as coloured graphs as a practical solution to isomorphism was suggested independently by Brendan McKay to Müller and Tegos.

Chapter four describes in detail the main result of this thesis, namely the first polynomial time algorithm for line segment diagrams isomorphism. The outline of this algorithm (including the notion of clump and the planarity based framework) was developed jointly with Hayward; the details were worked out by me. The planar graph isomorphism algorithms of Hopcroft and Tarjan formed the basis of this work, and the new algorithm presented by this thesis is simply theirs with appropriate modifications recognizing our labelings. The clump adjacency graph is a new construction bridging the gap between line segment diagrams and labeled planar graph isomorphism.

Chapter 2

Background

In this chapter we will introduce some basic definitions of graph theory, and then discuss some important graph isomorphism results related to this thesis.

2.1 Graph Terminology

This section will quickly recap important graph terminology. The reader is directed to any of the numerous introductory graph theory textbooks (such as [Wes01]) for more detail.

We begin with definitions.

Definition 2.1.1. A *graph* $G = (V, E)$ is a set V of points (vertices) and a set $E \subseteq V \times V$ of pairs of points (edges). We say the size of the graph is $n = |V|$ and let $m = |E|$.

Definition 2.1.2. A graph is *simple* if there is at most one edge connecting any pair of vertices and no edge connects a vertex to itself.

Definition 2.1.3. A *multigraph* is a graph whose edges form a multiset.

It will be understood throughout this paper that the term "graph" refers to a simple graph, unless explicitly indicated to be a multigraph.

Definition 2.1.4. If the elements of the edge set of a graph are ordered pairs, the graph is said to be *directed* (and forms a *digraph*), u is the head and v the tail of the edge (u, v), the edge is said to be *directed* or *oriented*, and the directed edge is sometimes called an *arc*. If the edges are unordered (where (v, u) refers to the same edge), the graph is said to be *undirected* and the edge is *undirected*. An orientation can be imposed on the edges of an undirected graph to make the graph directed.

Unless we state otherwise, all graphs in this thesis will be undirected. We also mention an abuse of notation common in graph theory where the edge consisting of vertices u, v is written as (u, v), regardless of whether the edge is directed or undirected.

We now discuss the concept of planarity, introducing some related ideas along the way.

Definition 2.1.5. A *path* of length k from v_0 to v_k is a set of edges (v_0, v_1), $(v_1, v_2), \ldots, (v_{k-1}, v_k)$ such that all the vertices $v_0, \ldots, v_k$ are distinct (except possibly $v_0 = v_k$).

Definition 2.1.6. A *cycle* is a path except $v_0 = v_k$, and all cyclic permutations are the same cycle.

Definition 2.1.7. A graph is *connected* if there is a path between any pair of vertices.

Definition 2.1.8. A *tree* is a graph with no cycles.

Definition 2.1.9. A graph is *complete* if an edge exists between every pair of distinct vertices. A complete graph on n vertices is denoted by K_n.

Definition 2.1.10. An (induced) subgraph of a graph G is a *clique* of size k (or a k-clique) if it is isomorphic to the complete graph K_k. A 3-clique is called a *triangle*.

Definition 2.1.11. A *complete bipartite graph* is a graph whose vertex set can be partitioned into two parts, one of size n and the other of size m, such that an edge exists iff the two endpoints are from different parts. Such a graph is abbreviated as $K_{n,m}$.

Definition 2.1.12. A graph $G_2 = (V_2, E_2)$ is a *subgraph* of $G_1 = (V_1, E_1)$ if $V_2 \subseteq V_1$ and $E_2 \subseteq E_1$.

Definition 2.1.13. A graph $G_2 = (V_2, E_2)$ is an *induced subgraph* of $G_1 = (V_1, E_1)$ if $V_2 \subseteq V_1$ and $E_2 = \{(u,v) \in E_1 | u, v \in V_2\}$, the set of all edges from G_1 whose endpoints are in V_2. We say that the subgraph G_2 is induced by the vertex set V_2, and write G_2 as $G_1[V_2]$.

Definition 2.1.14. A graph M is a *minor* of a graph G if M can be obtained from G by a sequence of zero or more edge deletions, edge contractions (merging the two endpoints of an edge) and deletions of isolated (disconnected) vertices. Equivalently, M is a minor of G if some sequence of edge divisions (adding a new vertex in the middle of an edge) of M is a subgraph of G.

Definition 2.1.15. A *crossing-free embedding* of a graph G in the plane is a drawing of G in the plane such that

- vertices are represented by points,
- edges are represented by continuous, smooth curve segments (curves),
- no two points intersect,
- the only points a curve intersects are the edge's end vertices, and the intersection occurs only at the curve's endpoints, and
- two curves intersect only at common end vertices.

Definition 2.1.16. A graph G is said to be *planar* if there is a crossing-free embedding of G in the plane.

Theorem 2.1.17. (Kuratowski [Kur30]) A graph is planar iff it does not contain a K_5 or $K_{3,3}$ as a minor.

Proofs of this theorem are well known in the literature and omitted here. This theorem will be useful later in showing certain graphs not to be planar. We now proceed to some basic connectivity definitions.

Definition 2.1.18. An *n-bond* is a multigraph consisting of a pair of vertices connected by n edges.

Definition 2.1.19. A vertex is a *cut point* or an *articulation point* if its removal increases the number of components in the graph.

Definition 2.1.20. A graph is said to be *biconnected* or *2-connected* if it does not contain a cut point.

We conclude this section with an easy yet important use of depth first search.

Theorem 2.1.21. (Hopcroft and Tarjan [HT73b]) The cut points and biconnected components of a graph can be found in time linear in the size of the graph.

2.2 Graph Isomorphism

A natural question to pose is when two graphs are essentially the same. A graph is an abstract object, and must be represented in some manner. Graphs which are in some sense similar may have different representations. Isomorphism is a mechanism for identifying similarity.

Definition 2.2.1. Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there exists a bijection $\pi : V_1 \to V_2$ such that $(u, v) \in E_1$ iff $(\pi(u), \pi(v)) \in E_2$.

One can discuss how quickly one can determine whether two graphs are isomorphic. Unfortunately, there is no known polynomial time algorithm to provide the answer for general graphs. Despite extensive effort, no one has been able to show graph isomorphism is **NP**-complete. The graph isomorphism decision problem is polynomial time equivalent to its counting problem (counting the number of isomorphisms between two graphs), a phenomenon not found in **NP**-complete problems [Mat79]. Another piece of evidence against **NP**-completeness is that it would collapse the polynomial time hierarchy to the second level [BHZ87, Sch88], a possibility widely disbelieved. There are also other theoretical results also suggesting graph isomorphism should not be **NP**-complete (for example, [KST92]).

Torán [Tor00] surveys facts suggesting graph isomorphism does not contain enough structure or redundancy to be hard. It is also noted that all lower bounds are quite weak, a notable embarrassment to complexity theorists. Fortunately the isomorphism problem has been shown to be polynomial time for several classes of graphs including planar graphs, graphs of bounded degree, and trees. We now discuss algorithms for the first two of these classes; algorithms for the third class are easy to derive, and are omitted.

2.2.1 Planar Graph Isomorphism

Whereas graph isomorphism in general is not known to be polynomial, polynomial time isomorphism algorithms are known for certain special classes of graphs. Two of particular interest to this thesis are planar graphs and graphs of bounded degree. This section will summarize planar graph isomorphism, and the next shall discuss trivalent (and by generalization, all bounded degree) graph isomorphism.

Hopcroft (with Tarjan and later Wong) showed planar graph isomorphism is polynomial time with a series of algorithms with $O(n^2)$ [HT71], $O(n \log n)$ [HT72] and eventually linear $O(n)$ time [HW74]. We briefly summarize the contributions of these algorithms, but make special mention of the $O(n \log n)$ algorithm indicating its usefulness to an isomorphism algorithm derived later in this thesis. Other algorithms exist, many of which are purported to be simpler. Fontet's algorithm [Fon76] and Colbourne and Booth's algorithm [CB81] are not discussed in detail here, since we will be concentrating on Hopcroft and Tarjan's algorithm for much of this thesis.

Hopcroft and Tarjan's $O(n^2)$ planar graph isomorphism algorithm operates by assigning ad hoc numbers (isomorphism codes) to pieces of the graphs such that two pieces have the same number if and only if the two pieces are isomorphic. The algorithm begins with small pieces of the graph (say, for example, the individual vertices) and combines the codes from smaller pieces to compute codes for bigger pieces (and eventually the entire graph). Though this approach is not known to work in polynomial time for general graphs, a graph's planarity permits a clever application of this approach to obtain a polynomial time algorithm.

The algorithm begins by dividing the two graphs into connected components. Clearly once we obtain codes for each component we simply need to compare the list of codes (which are really multisets) from each graph for equality. Each component is subdivided into biconnected components, and a new structure is created, called a *2-tree*. The vertices in a 2-tree are all the biconnected components and articulation points in the component, with edges present between a biconnected component B and articulation point a exactly when $a \in B$. This inclusion relation creates a tree. Assuming we have assigned codes to each vertex in a 2-tree, we simply apply a labeled tree isomorphism algorithm to determine isomorphism of the components. Labeled tree isomorphism can be solved in linear time (using standard algorithms

[HT72]) or sublinear time (using parallel methods [Lin92]).

We assign isomorphism codes to the biconnected components by subdividing each biconnected component into triconnected components (see Section 2.2.2 for a discussion of triconnected components and an important note on requirements for uniqueness). The triconnected components and biarticulation point pairs become vertices in a *3-tree*, whose edges are defined by the inclusion of biarticulation point pairs within a triconnected component (similar to the 2-tree construction). Again a tree is formed to which we can apply tree isomorphism algorithms.

The final step is to determine isomorphism classes of the triconnected components. Indeed this is the true work of the algorithm and is dealt with separately in Section 2.2.2. To summarize, a triconnected component has a unique embedding in the sphere (and hence the plane). The component is tested for planarity and (assuming planar) a planar representation constructed using an $O(n \log n)$ time algorithm. From this planar embedding we can derive face information, which can be used to determine similarity properties of the vertices in the graph. A finite automaton is constructed corresponding to the planar representation. Hopcroft showed that two triconnected components are isomorphic if and only if their automata are equivalent [Hop70a]. Equivalence of the automata is determined with an $O(n \log n)$ algorithm [Hop70b], from which ad hoc integer isomorphism codes are assigned.

General finite automata are in fact more powerful than this problem requires, an observation leading to a simplified partitioning algorithm for assigning isomorphism codes for triconnected components. Hopcroft and Tarjan describe this algorithm in [HT73c]. The partitioning algorithm is also used extensively as the basis for Algorithm 1 later in this thesis, at which time it is more fully explained.

Hopcroft and Wong present a linear time algorithm for isomorphism of planar graphs in [HW74]. Though their algorithm is asymptotically efficient, the hidden constant seems too large for practical use. This algorithm works with a specific labeled planar representation of the graphs, applying a number of reductions of varying priority. The result is a canonical form of the graph: either a regular polyhedra or a single vertex. These can be quickly tested for isomorphism using exhaustive matching.

This linear time algorithm uses several independent reductions, with several special cases. Since we choose the simpler $O(n \log n)$ algorithm for use in this thesis, the details of the linear algorithm are omitted. It remains as future work to modify this (or any other linear time algorithm) to operate on the structures introduced by this thesis below.

2.2.2 Triconnected Components

The planar graph isomorphism problem essentially reduces to the case where the graphs are triconnected. We must define what we mean by a triconnected graph.

Definition 2.2.2. A *cut pair* or *separation pair* (or *biarticulation point pair*) in a biconnected (multi)graph G is a pair of vertices $\{u, v\}$ such that the removal of the pair increases the number of components in G.

Definition 2.2.3. A biconnected (multi)graph is *triconnected* (or *triply connected* or *3-connected*) if it contains no cut pairs.

Small (multi)graphs (on at most three vertices) cannot contain a cut pair. We are usually interested in 3-bonds and triangles K_3 as small "triconnected" (multi)graphs, as we will see below.

Hopcroft and Tarjan give a linear algorithm in [HT73a] for dividing a multigraph into triconnected components. To obtain a unique decomposition, Tutte's definitions [Tut66] are used to specify triconnected components. We now briefly outline their algorithm.

The procedure begins by partitioning into connected components and separating multiple edges into groups of 3-bonds. This step is unnecessary if we begin with a connected graph. Biconnected components are then discovered using depth first search. Each biconnected component is divided by performing a number of biarticulation point pair splits. The last step is to merge adjacent triple bonds into n-bonds, and merge adjacent triangles into polygons. The result is a set of unique triconnected components of the original (multi)graph. This uniqueness is critical to efficient isomorphism algorithms.

To split a biconnected component G, we find a cut pair $\{u, v\}$ in G. We partition $G - \{u, v\}$ into connected components $B_1, \ldots, B_k$ (there must be at least two since $\{u, v\}$ is a cut pair). For each component B_i, a *cleavage graph* is formed by adding a virtual edge (u, v) to the subgraph induced on the vertices $V_i = V(B_i) \cup \{u, v\}$ (the vertices of B_i plus the cut pair). Formally, $G_i = (V_i, E(G[V_i]) \cup (u, v))$. The idea behind the virtual edge is to effectively replace the portion of the graph on the "other" side of the cut pair. This newly constructed cleavage graph remains biconnected since paths are retained (but possibly shortened).

By repeated application of this procedure we eventually obtain a collection of cleavage graphs, each of which is either large and triconnected (no cut pairs) or contains at most three vertices (hence being either a 3-bond or a triangle). The merge step simply makes the set of components unique regardless of the order in which we process the cut pairs. The results are called the triconnected components of G.

Once found, we must answer the isomorphism question on the triconnected components. If the original graph is planar, it is obvious that the triconnected components will also be planar. Hopcroft and Tarjan present an algorithm for determining isomorphism of triconnected planar graphs in [HT73c].

2.2.3 Trivalent and Bounded Degree Graph Isomorphism

The isomorphism problem for trivalent graphs (graphs of vertex degree at most 3) has close ties to group theory. It is well known that graph isomorphism is polynomial time reducible to finding a set of generators for the group $Aut(G)$ of automorphisms of a graph G.

Luks [Luk80, Luk82] shows how finding the generators of $Aut(G)$ is reducible to the colour automorphism problem, where the generators for the subgroup of permutations preserving the colourings of a set are found. Specifically, isomorphism testing of graphs of degree t is reduced to the colour automorphism problem for groups whose composition factors are subgroups of S_{t-1}. For trivalent graphs, we have primitive Sylow 2-groups, which can only have order 2. Generators of primitive 2-groups can be found in polynomial time.

Indeed the property extends: primitive p-groups can only have order p. Though groups resulting from higher valence graphs are not p-groups, they are almost so, and p-subgroups of polynomial index can be found in polynomial time.

Applying divide-and-conquer strategies, Luks derives a polynomial time algorithm for isomorphism of trivalent graphs, and shows the extension to graphs of bounded degree. Though the polynomial bound for Luks' algorithm is elementary to prove, a naive implementation takes $O(n^{10})$ time. Exploiting several tricks, a running time of $O(n^5)$ can be obtained. Several other techniques can improve the bound. Galil et al. [GHL$^+$87] deeply exploit underlying group theory to improve Luks' algorithm to a running time of $O(n^3 \log n)$ for a deterministic algorithm, and $O(n^3)$ as a Las Vegas algorithm (a probabilistic algorithm allowed to flip coins, yet does not make errors — the improvement comes from a procedure showing nonemptiness of a set).

As these results are not central to our result, we will not discuss them here in any further detail. The main importance of these algorithms to this thesis is merely in their existence, for possible application to structures later mentioned. An excellent and entertaining discussion of the details can be found in [Luk82] and [GHL$^+$87], and a monograph by Hoffmann [Hof82] discusses much of the associated group theory and related algorithms.

Chapter 3

Preliminaries

This chapter introduces the concept of grid point sets, and proceeds to motivate further constructions and structures to encode additional knowledge and problem structure. We will see that a grid point set is a very basic representation, whereas more abstract concepts such as Amazon position graphs and line segment diagrams capture more information crucial for proper analysis.

3.1 Grid Point Sets

We first define the basic combinatorial object we will be using throughout this chapter.

Definition 3.1.1. Consider a grid of horizontal and vertical lines and the squares thus formed (as in a chess or checkers board). A *grid point set* (or GPS for short) is a finite subset of squares on the grid. We say two squares in a grid point set are *adjacent* in the GPS if they are adjacent on the grid in any of the eight primary directions (horizontal, vertical or diagonal).

Figure 3.1 illustrates some grid point sets. The shaded squares compose a single grid point set. Throughout this thesis, a *primary direction* will consist of the eight directions of movement possible on a chess board: specifically (using map directions) north, south, east, west, northwest, northeast, southwest and southeast. Observe that squares are considered adjacent if they share an edge or corner. Refer to the arrows in Figure 3.1 for illustration.

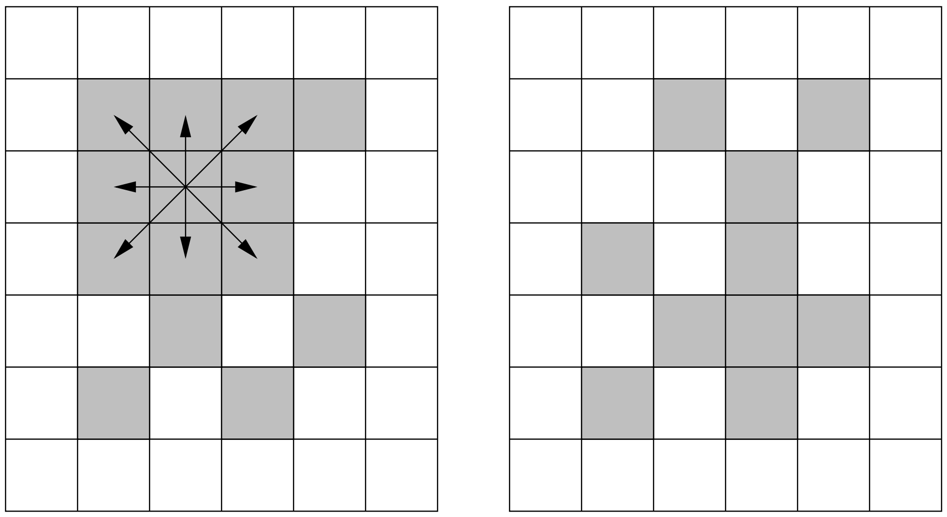


Figure 3.1: Examples of grid point sets.

We say a grid point set is *connected* if a path (a sequence of adjacent squares in the set) exists between any two squares in the set. We usually concentrate on connected grid point sets for convenience, since the extension to disconnected grid point sets tends to be trivial.

Henceforth when illustrating grid point sets, only squares included in the set will be indicated.

One can now examine the structure inherent in a grid point set. It is clear that the physical placement of squares relative to each other is important, but not necessarily the placement of the set on a particular grid. Furthermore, only the squares in the set are relevant: the size of the grid upon which we draw a grid point set is irrelevant (assuming it is large enough to fit the set).

Definition 3.1.2. Two grid point sets G and H are *isomorphic* if there exists a bijection $\pi : G \to H$ such that for all $p_1, p_2 \in G$, either

(I) p_1 and p_2 are adjacent horizontally (respectively vertically) in G iff $\pi(p_1)$ and $\pi(p_2)$ are adjacent horizontally (resp. vertically) in H, or

(II) p_1 and p_2 are adjacent horizontally (respectively vertically) in G iff $\pi(p_1)$ and $\pi(p_2)$ are adjacent verically (resp. horizontally) in H

and either

(III) p_1 and p_2 are adjacent diagonally northwest-southeast (respectively northeast-southwest) in G iff $\pi(p_1)$ and $\pi(p_2)$ are adjacent diagonally northwest-southeast (respectively northeast-southwest) in H, or

(IV) p_1 and p_2 are adjacent diagonally northwest-southeast (respectively northeast-southwest) in G iff $\pi(p_1)$ and $\pi(p_2)$ are adjacent diagonally northeast-southwest (respectively northwest-southeast) in H.

We now describe all such isomorphisms.

Proposition 3.1.3. Two connected grid point sets G and H are isomorphic if and only if G can be transformed into H by a sequence of the following operations on an infinite sized grid:

(i) translations,

(ii) rotations by 90°, and

(iii) flips over a line in one of the primary directions.

Proof. We first label the squares of the grid with an ordered pair of integers in a fashion similar to coordinates in the Euclidean plane: we arbitrarily choose an origin (which is labelled (0,0)) and increase the value of the first label as we move right, and increase the value of the second label as we move up.

Case "$\Leftarrow$": Consider a grid point set G which can be transformed into a grid point set H by a sequence of translations, rotations and flips. For each operation in the sequence, we define a bijection below. To form the isomorphism function we simply compose the bijections according to the sequence. Composition of bijections (and hence isomorphisms) are bijections, hence the composition will be an isomorphism.

Consider a translation operation. It can be expressed as "move n squares right and m squares up." Define $\pi_{T_{n,m}}(p_x, p_y) = (p_x + n, p_y + m)$, where (p_x, p_y) is the label of a point $p \in G$. Horizontal/vertical adjacency of points is preserved since (p_x, p_y) and (p_x+1, p_y) are mapped to (p_x+n, p_y+m) and (p_x+n+1, p_y+m) (and symmetrically for vertically adjacent points). Diagonal adjacency is also preserved since (p_x, p_y) and (p_x+1, p_y+1) is mapped to (p_x+n, p_y+m) and (p_x+n+1, p_y+m+1), and since (p_x, p_y+1) and (p_x+1, p_y) is mapped to (p_x+n, p_y+m+1) and (p_x+n+1, p_y+m). Conditions (I) and (III) are satisfied, so $\pi_{T_{n,m}}$ is an isomorpism.

Consider now a rotation by 90°. We note that any additional rotation (say by 180° or 270°) can be completed by a sequence of rotations by 90°. Without loss of generality we assume rotation counterclockwise around the origin. Define $\pi_R(p_x, p_y) = (-p_y, p_x)$. Again horizontal/vertical adjacency and diagonal adjacency are preserved (by argument similar to above). So π_R is an isomorphism.

Consider now a flip over a line in a primary direction. Without loss of generality we assume the flip is over the horizontal line though the origin (flips over other horizontal lines can be expressed with a sequence of translations before and after the flip, and flips over lines in other primary directions can be expressed with a suitable sequence of rotations and translations). Define $\pi_F(p_x, p_y) = (p_x, -p_y)$. Again adjacency is preserved by a similar argument, so π_F is an isomorphism.

Case "$\Rightarrow$": For the forward direction, assume that G and H are isomorphic and let π be an isomorphism between G and H.

Let us suppose conditions (I) and (III) in Definition 3.1.2 hold, that for $p_1, p_2 \in G$, p_1 and p_2 are adjacent horizontally (resp. vertically) iff $\pi(p_1)$ and $\pi(p_2)$ are adjacent horizontally (resp. vertically) in H, and p_1 and p_2 are adjacent diagonally northwest-southeast (resp. northeast-southwest) iff $\pi(p_1)$ and $\pi(p_2)$ are adjacent diagonally northwest-southeast (resp. northeast-southwest) in H. Then, respecting π, G can be transformed into H by simply applying a translation (if the order of points are preserved from G to H) or by applying a rotation by 180° followed by a rotation (if the order of adjacent points is reversed by π).

Now let us suppose the conditions (I) and (IV) hold. Then G can be transformed into H by applying a flip either over a vertical line or over a horizontal line (depending whether π reverses adjacent points), followed by a translation. We may choose any vertical (resp. horizontal) line to flip over since the result is different only in a translation.

Next suppose that conditions (II) and (III) hold. Then G can be transformed into H by applying a rotation by 90°, then a flip over either a vertical line or a horizonatal line (depending whether π reverses diagonally adjacent points), followed finally by a translation.

Finally, if conditions (II) and (IV) hold, then G can be transformed into H by applying a rotation either by 90° or by 270° (depending whether π reverses adjacent

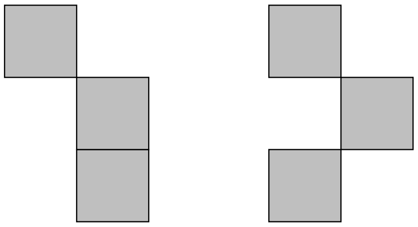


Figure 3.2: "Similar" grid point sets which are non-isomorphic.

points), followed by a translation. □

We now mention the focus of our problem: we wish to represent, in addition to the grid point set, the moves and operations possible in the game of Amazons. Abstractly, a move is a movement of any number of squares in one of the eight primary directions. A move must only pass over squares in the grid point set. An operation is to remove a single square from the grid point set.[1]

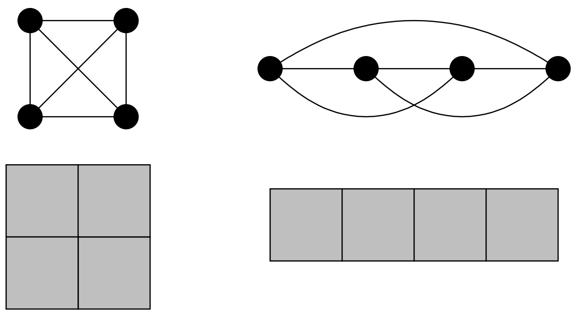
It is clear that isomorphism of grid point sets is insufficient for representing similar configurations when discussing the set of possible moves. Figure 3.2 illustrates this shortcoming. Two non-isomorphic grid point sets have exactly the same sets of moves, and hence should be considered similar. (The fact these grid point sets are non-isomorphic should be immediate considering Proposition 3.1.3.) We will soon introduce a structure sufficient for capturing this similarity.

3.2 Amazon Positions as Graphs

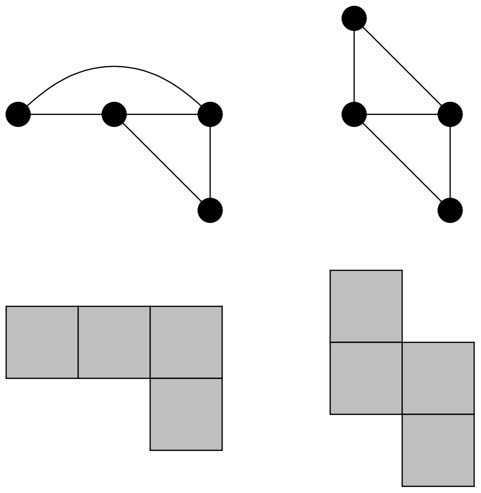
An Amazon position can be represented by a simple graph. By identifying a vertex with each square and including an edge exactly when a piece can move between the two squares in exactly one move, we can represent all the possible movements in the position. Standard graph algorithms can be applied for analysis. In particular, standard graph isomorphism can be used to compare two Amazon positions.

Figure 3.3 illustrates two pairs of positions and their corresponding graph representations. It is clear that both the four-square box and four-square line are both represented by K_4, a clique on four vertices. The question should be raised, are

[1]The rules from the game Amazon are somewhat more precise and restrictive. Refer to Appendix A for the full game rules.



(a)



(b)

Figure 3.3: Two pairs of indistinguishable positions when using graph representations.

these two Amazon positions similar when playing the game? In some regard, the answer should be yes: all squares can be reached within one move regardless of the starting square. But there are some differences we may want to quantify, such as how the move is completed.

This graph representation captures the static relationship between a position and the adjacency of movements upon this position. It completely represents the adjacency of squares, and the paths (the sequence of moves) one can follow to reach a particular square from a starting square. We can talk of paths and translate the concept directly to the position, and path length translates directly to the number of moves required. Hence isomorphic diagrams suggest the range of movements are identical among the corresponding Amazon positions. Consider, for example, the pair of positions in Figure 3.3(a). Each square can be reached in one move from any other square in the position. If this is all that needs to be considered, graph representation is adequate.

3.3 Line Segment Diagrams

We now consider the removal of a square from an Amazon position, in addition to simply moving pieces within the position. We wish to describe and represent the resulting position after applying such an operation.

Consider the position in Figure 3.3(a). After removing one of the squares in the left figure, the representation of the position is merely the original representation with the corresponding point removed (along with any incident edges). If we remove one of the internal squares from the right position, we obtain a disconnected position. Simply removing the corresponding point (and incident edges) from the original representation is no longer sufficient. A finer representation is required to adequately capture the relationship of "moving through squares."

Definition 3.3.1. An *ordered point set* $P = [p_1, p_2, \ldots, p_k], k \geq 1$, is a nonempty sequence of points from a grid point set. We say P is

(i) *contiguous* if p_i is adjacent to p_{i+1} for $i = 1, 2, \ldots, k-1$ (where adjacency is as for a GPS, in the primary directions on the grid),

(ii) *collinear* if p_i is adjacent to p_{i+1}, p_{i+1} is adjacent to p_{i+2}, and both adjacencies are in the same primary direction, for all $i = 1, \ldots, k-2$, and

(iii) *maximal* if P cannot be extended within the grid point set to include another point (on either end).

We can restate the natural meaning of these definitions informally using common chess terminology.

(i) *Contiguous* means a chess king could move from p_i to p_{i+1} in one move, for each i.

(ii) *Collinear* means a chess queen could move from p_i to p_j in one move, for each distinct i and j.

(iii) *Maximal* means a chess queen could not move any farther in this direction within this grid point set.

For brevity, a maximal collinear contiguous ordered point set will be called a *line segment*, or perhaps (slightly incorrectly) referred to as simply a *line*. A collinear contiguous ordered point set (or when maximal, a line segment) $[p_1, \ldots, p_k]$ has length k and will be called a *k-line*. Note that a *k-line* is not necessarily maximal.

Definition 3.3.2. Two ordered point sets (or line segments) $l_1 = [p_1, \ldots, p_k]$ and $l_2 = [q_1, \ldots, q_m]$ are considered equal if $k = m$ and either $p_i = q_i$ for $i = 1, \ldots, k$ or $p_i = q_{m-i}$ for $i = 1, \ldots, k$. We write $l_1 = l_2$.

This definition states that the direction in which we view a line segment is irrelevant. We now describe these objects further.

Corollary 3.3.3. *Line segments* have the following properties.

1. Each line segment can be uniquely defined by its two end points, and may pass through additional points: $l = [p_1, \ldots, p_k], k \geq 1$.

2. Each point appears on a particular line at most once: $p_i = p_j \iff i = j$.

3. Line segments are distinguished simply by the relative order of their points: line segments may be read either forward or reverse (as per the equality condition in Definition 3.3.2).

4. Each line segment corresponds to a possible movement of an amazon or an arrow in Amazons.

We now introduce a combinatorial object which describes all line segments for a particular grid point set.

Definition 3.3.4. A *line segment diagram* (or LSD for short) $D = (P, L)$ consists of a grid point set P (the vertices) and the collection of maximal collinear contiguous ordered point sets L (the lines).

Line segment diagrams have a number of useful properties, including the following. The reader may note that these are conditions inherited from the planarity of the underlying grid point sets.

1. Each point of a grid point set which is in a connected component (of the grid point set) of size at least two is contained on at least one line.
2. Two distinct points have at most one line in common.
3. Two distinct lines have at most one point in common.

Müller and Tegos discuss the relationship of line segment diagrams (graphs) to computer playing of Amazons in [MT01], including inspiration to why efficient isomorphism testing is desirable.

Frequently we are interested in only portions of a line segment diagram.

Definition 3.3.5. A *subdiagram* (sub-line segment diagram) $S = (P', L')$ of a line segment diagram $D = (P, L)$ is *induced* if every line segment $l' \in L'$ is a maximal ordered subset of some line segment $l \in L$ with respect to P'.

We now turn our attention to the isomorphism question concerning line segment diagrams. Due to the fact that the set of line segments of a line segment diagram is determined by the underlying grid point set, we define two variants of line segment diagram isomorphism.

Definition 3.3.6. Two line segment diagrams $D_1 = (P_1, L_1)$ and $D_2 = (P_2, L_2)$ are *isomorphic* if there exists a bijection $\pi : P_1 \to P_2$ such that $[p_1, \ldots, p_k] \in L_1$ if and only if $[\pi(p_1), \ldots, \pi(p_k)] \in L_2$. That is, $\pi(L_1) = L_2$. We write $D_1 \sim D_2$ to indicate D_1 is isomorphic to D_2.

Recall that grid point set isomorphism (Definition 3.1.2) corresponded to flips, rotations and translations.

Definition 3.3.7. Two line segment diagrams (P_1, L_1) and (P_2, L_2) are *strongly isomorphic* if their underlying grid point sets P_1 and P_2 are isomorphic. That is, P_2 can be obtained from P_1 by a finite combination of

(i) flips over a grid line in a primary direction,

(ii) rotations by 90°, and

(iii) translations.

Recall that grid point sets are embedded in the grid, thus the above operations are meaningful. The following corollary immediately justifies our choice of language. We sketch its proof as not to obscure its idea.

Corollary 3.3.8. Two strongly isomorphic line segment diagrams are isomorphic.

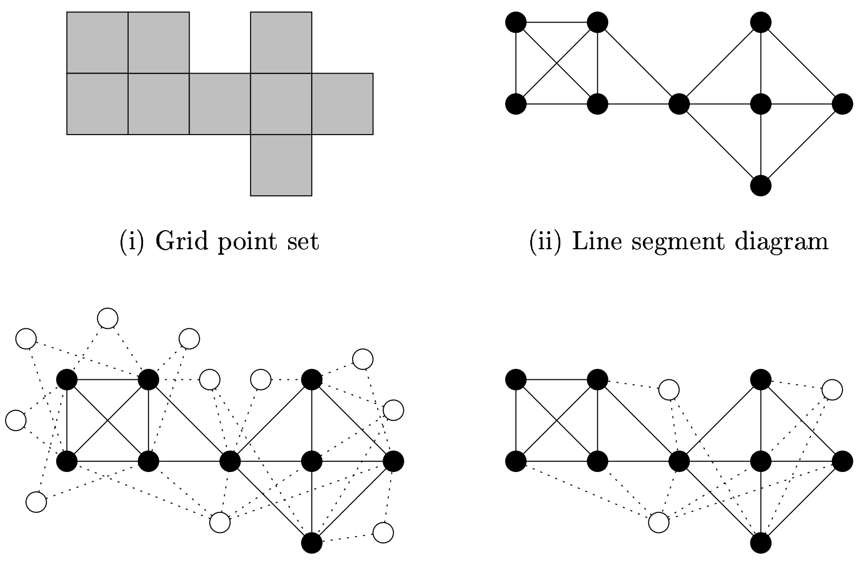
Proof (sketch). Translate, flip, and rotate the two grid point sets so that they are equal. This defines both the grid point set isomorphism and line segment diagram isomorphism. □

3.4 A Representation by Coloured Graphs

The line segment diagrams capture all structural information regarding an Amazons region, but does so by creating a new abstract object. In this section, an equivalent coloured undirected graph is constructed, upon which existing isomorphism algorithms can be applied.

The mapping discussed here was originally proposed by Brendan McKay [MT01], resulting in a format compatible with his *nauty* program [McK81, McK], a practically efficient program for computing canonical labellings and automorphism groups of graphs and digraphs.

To speak of the mapping, we must first introduce some notation. A coloured graph $G = (V, E, col)$ is a graph (V, E) together with a partition col of V, where each



(i) Grid point set (ii) Line segment diagram

(iii) Coloured graph (iv) Simplified coloured graph

Figure 3.4: Mapping line segment diagrams to coloured graphs.

partition set corresponds to a colour. For convenience these sets can be numbered or ordered, such that $col = \{V_1, \ldots, V_k\}$.[2]

The mapping proceeds as follows. Given a line segment diagram $D = (P, L)$, construct a coloured graph $G = (V, E, col)$ where

- $V = P \cup L$, i.e., each point and each line corresponds to a vertex,
- $col = \{P, L\}$, i.e., all point vertices are given the first colour (black) and all line vertices are given the second colour (white), and
- E consists of two types of edges: edges between two point vertices if they are adjacent on some line, and edges between a line vertex and each point on that line (i.e., $E = \{(p_1, p_2) : p_1, p_2 \text{ adjacent on some line } l\} \cup \{(p, l) : p \in l\}$).

An immediate observation is that the mapping may be simplified by omitting any lines of length 2 (the length 2 line vertices are redundant since they are encoded by edges between the point vertices). It should also be clear that G determines a unique line segment diagram up to isomorphism (simply construct the line segment diagram by reversing the construction just described).

[2]The reader should not confuse a coloured graph as used here with the concept of valid colourings of a graph and the graph colouring problem, which is well studied in the literature.

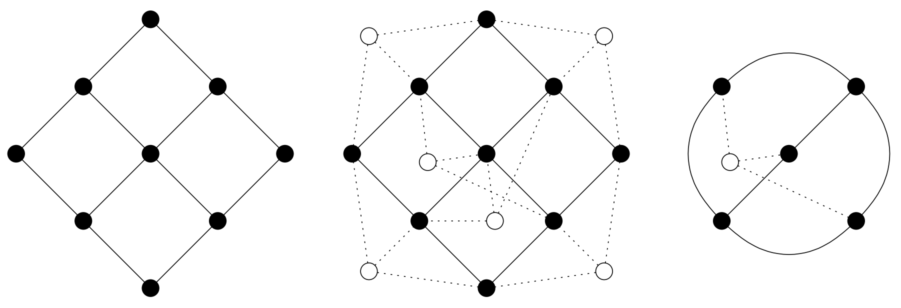


Figure 3.5: Line segment diagram to coloured graph mapping does not preserve planarity.

Figure 3.4 illustrates this mapping. Note that the edges connecting point (black) and line (white) vertices are dotted for the reader's convenience.

Benefits of this mapping centre around the fact that it produces a graph — a well-understood combinatorial object. There has been much prior work to solve assorted problems on graphs. In particular, very effective algorithms and programs exist to test isomorphism (in particular, McKay's aforementioned *nauty*). Unfortunately, isomorphism of coloured graphs is polynomial time equivalent to isomorphism of uncoloured graphs. We remind the reader that we saw in Section 2.2 that no efficient algorithm is known for solving (uncoloured) graph isomorphism in general. We recognize that these coloured graphs have some structure to them, and we may ask whether they are of bounded degree. The point vertices are bounded in their degree, but the lines may be of arbitrarily long length. This yields line vertices of unbounded degree, hence we cannot apply the polynomial algorithm for graphs of bounded degree.

Though we can apply McKay's practical algorithms, we have not (to this point) made any progress towards a polynomial time isomorphism algorithm. The near-planarity of the line segment diagrams and associated coloured graph mapping suggest approaches similar to planar graph isomorphism algorithms (see Section 2.2.1). Unfortunately, even a small "planar" line segment diagram can be mapped to a non-planar graph. For example, in Figure 3.5 a simple 9-vertex line segment diagram is mapped to a graph containing a $K_{3,3}$ minor, one of the forbidden subgraphs for planarity (recall Kuratowski's Theorem 2.1.17). Thus we surmise the coloured graph mapping is only of practical interest. In the next chapter we present a polynomial time algorithm to solve the line segment diagram isomorphism problem using other ideas.

Chapter 4

An Algorithm

We now develop a line segment diagram isomorphism algorithm which does run in polynomial time. The basis of this algorithm is the identification of subdiagrams with unique embeddings (up to strong isomorphism) in the grid. We proceed to encode local isomorphism properties and global structure with a labelled graph, and extend a planar graph isomorphism algorithm to determine isomorphism classes of components.

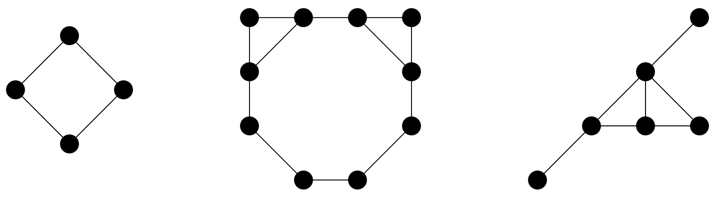
4.1 Clumps

We now introduce the concept of "solid." Given a line segment diagram, its grid point set can be translated, flipped or rotated into other isomorphic grid point sets. If these are the only isomorphic line segment diagrams, our original diagram has an essentially unique embedding the grid. This solidity makes the isomorphism question easy to answer.

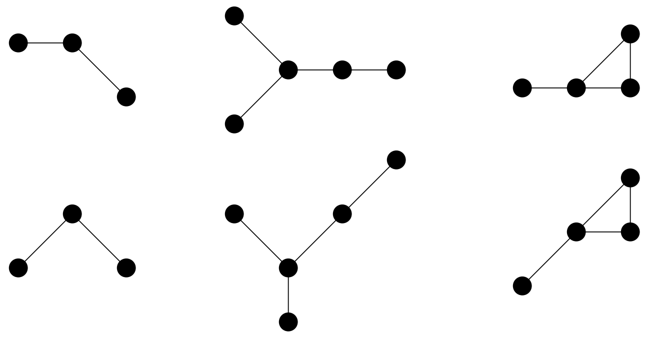
Definition 4.1.1. A line segment diagram D is called *solid* if each line segment diagram isomorphic to D is strongly isomorphic to D.

Figure 4.1 illustrates a selection of line segment diagrams, some solid and some not.

Solid diagrams can appear in many interesting ways, and we know of no characterization for all such diagrams. However, some classes of solid diagrams can be easily recognized, which motivates the next few paragraphs. We introduce one recursively defined class which is particularly useful and easy to recognize.



(a) Solid line segment diagrams.



(b) Nonsolid line segment diagrams.

Figure 4.1: Some solid and nonsolid line segment diagrams. Isomorphic, but not strongly isomorphic, diagrams are presented vertically in (b).

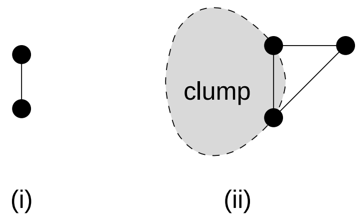


Figure 4.2: Constructing a clump.

Definition 4.1.2. A *subdiagram* S of a line segment diagram D is a *clump* if

- S consists of a pair of adjacent vertices, or
- S is the union of a clump S' and a vertex v such that v and two vertices of S' induces a triangle of D.

Figure 4.2 illustrates this construction. A clump is said to be *maximal* if no additional operations can be applied.

We will be particularly interested in maximal clumps. Unless otherwise stated, we will assume clumps are maximal.

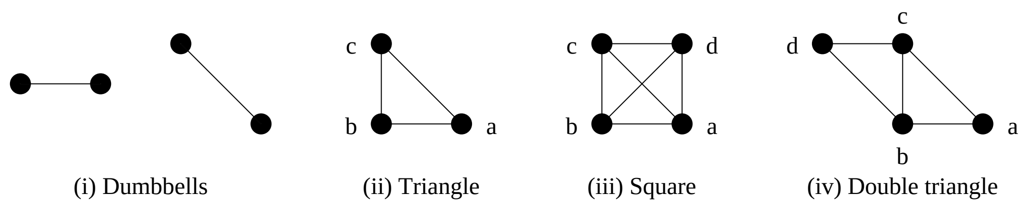


Figure 4.3: Nonsolid clumps.

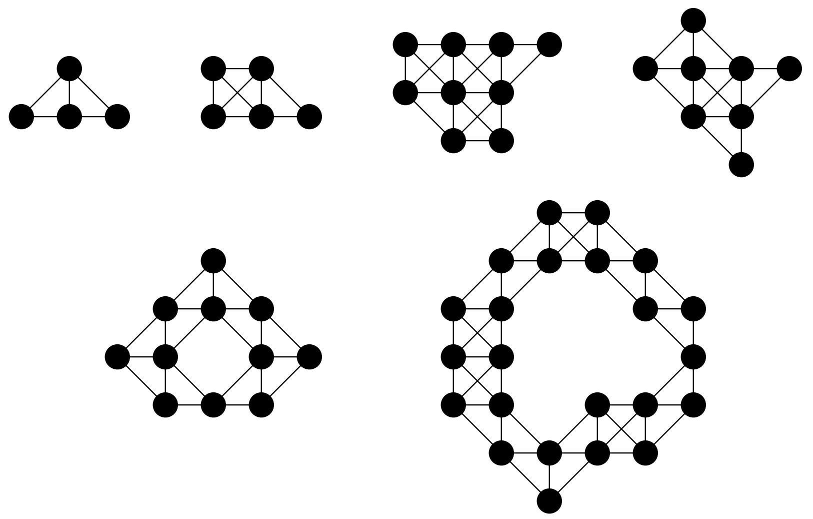


Figure 4.4: Some examples of solid clumps. The bottom two clumps each contain one hole (with one point and six points repectively).

Definition 4.1.3. Consider the grid upon which the grid point set of a line segment diagram lies. A *clump set* is the set of grid points composing a clump. Given a clump, a *hole set* in the clump is a nonempty connected set of grid points completely surrounded by the clump set. A maximal hole set is a *hole* in the clump.

It should be noted that a clump may have more than one hole, and often no holes at all. Some examples of clumps are presented in Figure 4.4, with the bottom two examples each containing one hole.

The usefulness of clumps in isomorphism testing is stated by the following theorem.

Theorem 4.1.4. A clump is *solid* if and only if it is not a dumbbell, triangle, square or double triangle (see Figure 4.3).

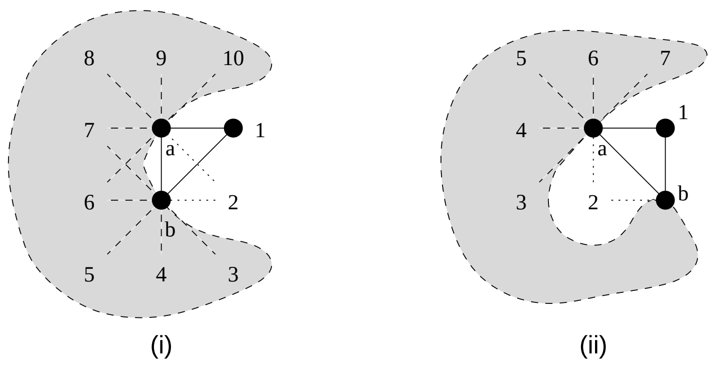


Figure 4.5: Why some clumps are solid.

Proof. We first show the objects described are not solid. It is clear such objects are clumps. Consider the labellings of the following clumps in Figure 4.3.

Dumbbell: The dumbbell can be oriented horizontally or vertically, or it can be oriented diagonally. Diagonal and horizontal/vertical grid point sets are not strongly isomorphic.

Triangle: Define $\pi_\triangle : [a, b, c] \mapsto [a, c, b]$. The line $[a, b]$ is changed from horizontal to diagonal, so the triangle is not strongly isomorphic.

Square: Define $\pi_\square : [a, b, c, d] \mapsto [a, c, b, d]$ and we have the same case as for the triangle. We comment that adding any corner vertex to the perimeter of the square solidifies the subdiagram, and any other vertex merely enlarges the clump.

Double triangle: Again switching b and c causes the same problem as in the triangle.

Consider now any other clump. It is constructed by recursive operations per Definition 4.1.2. We proceed via induction on the construction process.

The base case is a dumbbell, which is covered above. Consider the operation of adding a vertex to a vertical or horizontal line forming a triangle illustrated in Figure 4.5(i). The smaller gray clump C is either solid or one of the smaller non-solid clumps. Adding a vertex to the triangle either forms a 3-line solidifying the clump, or forms the square or double triangle. Adding to the square or the double triangle forms a 3-line, also distinguishing the location of the added point.

Assume now that C is solid in Figure 4.5(i). Then the new clump is solid if we can distinguish between the vertex being added at the only two possible locations, which are labelled 1 and 2. The lines within the clump will be different depending whether the vertex is at 1 or 2 (assuming the vertices within C are not also moved). If C is simply a dumbbell, we have constructed a triangle, which is covered above. Otherwise each of a and b are part of a triangle within the clump, which implies some of 3 through 10 are part of C. If any of 3, 4, 9 or 10 are in C, then 1 and 2 are distinguished and we are done. If either 5 or 7 (or, by symmetry, 6 or 8) are part of the clump, we have a 3-line through 1 (or 2), distinguishing these cases from others.

Consider now the operation of adding a vertex to a diagonal line forming a triangle illustrated in Figure 4.5(ii). Since a must be part of a triangle, at least two of 3 through 7 must be in C. If any of 3, 4, 6 or 7 are included, we are distinguished by adjacency, leaving 5 as our only option. This is not enough to form a triangle, contradicting a being in a solid clump. Hence we can distinguish between 1 and 2, and the new clump is solid. □

We remark that a clump with a hole is solid, which is easily verified considering the above theorem.

Corollary 4.1.5. The maximal clumps in a line segment diagram are unique.

Proof. This follows directly from the recursive definition in 4.1.2. □

One may ask how quickly clumps can be found within a line segment diagram, and once found, how quickly can we determine if two clumps are similar. The following two theorems address these questions. To analyze the theoretical efficiency of our algorithms, we use a uniform cost random access machine as our model of computation. We assume the numbers encountered are bounded by a constant multiple of the size of the line segment diagram, and hence small enough to fit into a single memory location.

Theorem 4.1.6. The set of all maximal clumps of a line segment diagram can be found in time linear in the number of vertices in the diagram.

Proof. Begin by finding a pair of adjacent vertices in the line segment diagram. To extend this clump we must apply the operation of Definition 4.1.2. Create a list of possible grid points and add all points adjacent to both clump vertices. For every vertex we add to the clump, add those grid points adjacent to the new vertex and at least one adjacent old vertex. Since each vertex is considered at most once, and the addition operation is constant time, maximal clumps can be found in time bounded by the number of vertices. □

A string can be constructed to represent a clump. For example, within the grid where we draw a clump we can choose an origin (0,0) such that all grid points in the clump have nonnegative coordinates, and in each of the two coordinates there exist at least one grid point with coordinate value zero. By imposing an ordering on the grid points (such that they are nondecreasing in the first coordinate, and when the first coordinate is constant they are nondecreasing in the second coordinate), we can construct a string representation of the clump by listing each grid point's coordinates. This is by no means the only possible string encoding, nor do we claim it is the best. Once we have chosen an arbitrary string representation scheme, we can speak of lexicographic ordering of the representations of clumps. We use such a strategy in the proof of the following lemma.

Lemma 4.1.7. Isomorphism of solid clumps can be determined in time linear in the number of vertices in the clump.

Proof. Since solid clumps have a unique embedding in the grid, we can easily determine a canonical form: of all the 4 rotations and 4 flip+rotations possible for a clump, determine the lexicographically first according to an arbitrary representation scheme. The representations and choosing the lexicographically first can be computed in time proportional to the number of vertices in the clump. Isomorphism testing is merely checking equality of these canonical embeddings. □

Lemma 4.1.8. Isomorphism of nonsolid clumps can be determined in constant time.

Proof. By Theorem 4.1.4 there are four types of nonsolid clumps, each of constant size. □

Theorem 4.1.9. Isomorphism of clumps can be determined in linear time.

Proof. By Lemmas 4.1.7 and 4.1.8. □

Clumps, into which any line segment diagram can be decomposed, can be efficiently classified into isomorphism classes. For example, hashing or data trees can provide a mapping from isomorphism classes to ad-hoc integers, which can be assigned to the clumps. Equality of clump labels would thus be equivalent to isomorphism of the clumps.

4.2 Clump Adjacency Graphs

We have shown a technique for determining the "similarity" of certain pieces of line segment diagrams. We now proceed to describe strategies for determining similarity of larger pieces of a line segment diagram. First we describe a labelled graph embedding describing the interaction of clumps.

Informally, a clump may be attached to other clumps. Formally we call a grid point shared between two clumps an *attachment point*. Each clump may have several attachment points (bounded by the number of points in the clump), and each attachment point may be a part of up to four clumps (an obvious bound considering Definition 4.1.2).

Definition 4.2.1. A *clump adjacency graph* (or CAG for short) $A(D)$ of a line segment diagram D is defined by the following.

- The vertices of $A(G)$ are the set of clumps and the set of attachment points.
- An edge (c, a) exists in $A(G)$ exactly when the attachment point a is contained within clump c.

Figure 4.6 illustrates the conversion process, from grid point set, via line segment diagram, to the clump adjacency graph. It should be noted that a clump adjacency graph is bipartite: the clump vertices are indicated by shaded dashed circles containing the clumps, while attachment vertices are indicated by solid circles.

Definition 4.2.2. The *canonical representation* of a clump adjacency graph $A(D)$ in the plane is as follows:

- If a clump has no holes, the clockwise order of its attachment points is preserved in $A(D)$.

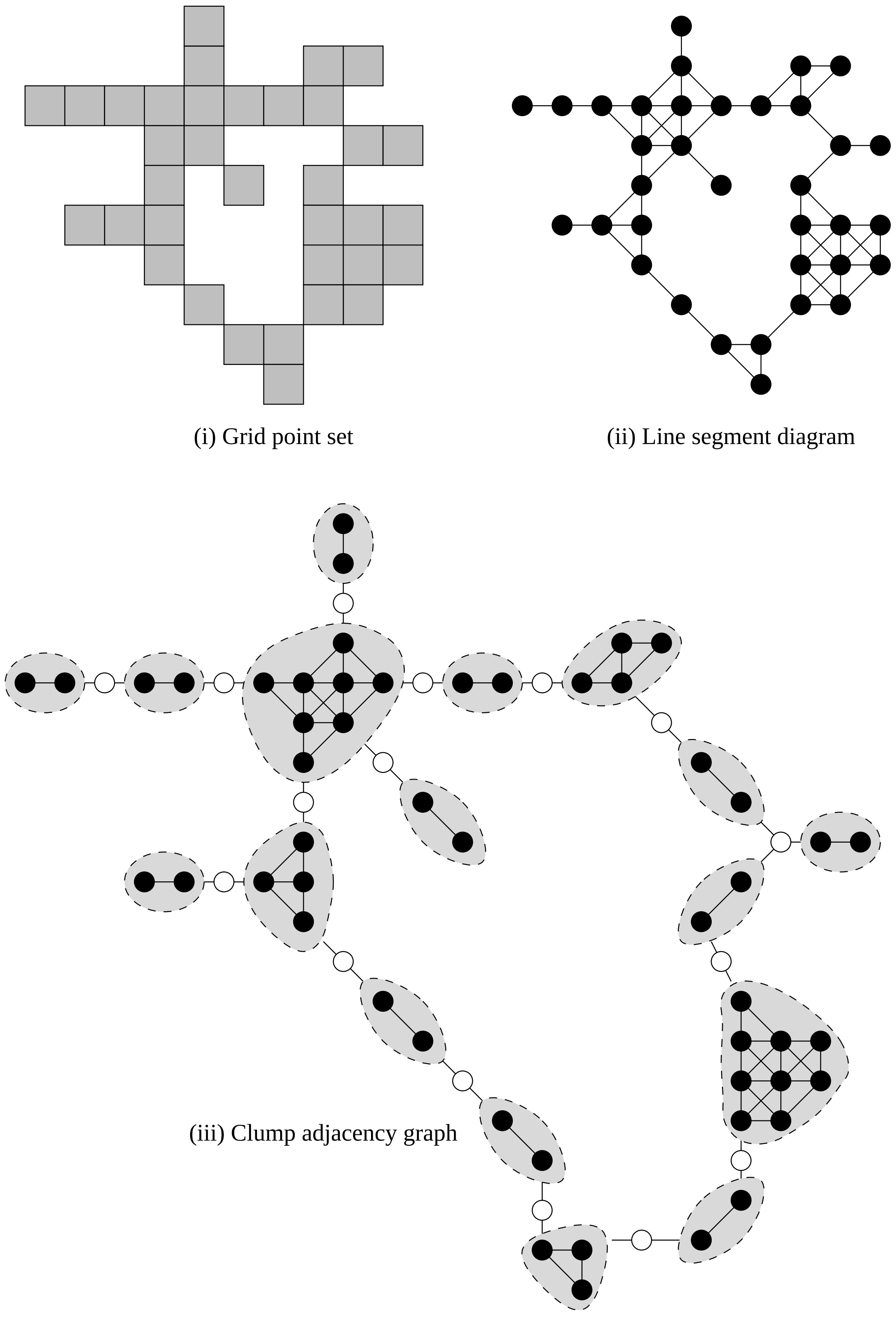


Figure 4.6: Creating a clump adjacency graph.

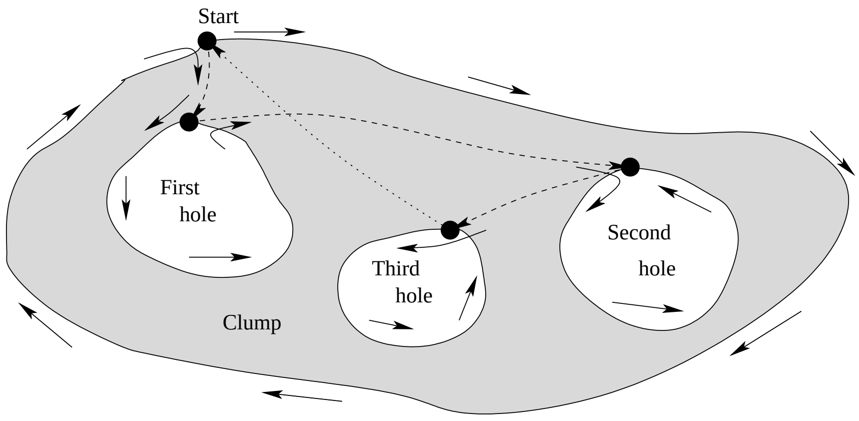


Figure 4.7: Ordering attachment points around a clump with holes.

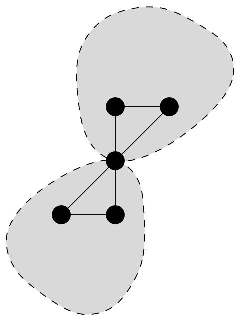


Figure 4.8: Double linked clumps.

- If a clump has holes, consider its lexicographically first grid embedding (as according to some arbitrary encoding scheme). Order the attachment points beginning along the outside edge starting from the top-right-most grid point and proceeding clockwise. Determine the top-right-most grid point belonging to the perimeter of a hole and order the attachment points proceeding counter-clockwise. Continue ordering attachment vertices in other holes in the same manner. (See Figure 4.7.)

Consider two adjacent clumps joined by at least two lines, with portions of each line being within both clumps (for example, Figure 4.8). The orientation and composition of one clump defines the orientation of the other. One clump cannot be changed without suitably changing the other (where "changes" include flips, rotations, etc.). Hence joining such double-linked clumps and treating them as one object reflects this dependence better than simple clump adjacency. We will incorporate this feature into our algorithm presented below.

It is clear that the clump adjacency graph can be simplified by joining adjacent collinear dumbbells into sub-lines, but this improvement does not affect asymptotic running time. We will not apply this simplification in our examples.

One additional detail is needed for a true equivalence between line segment diagrams and clump adjacency graphs: the inherent structure of the attachments must be preserved. Collinearity of clumps is an important invariant which is destroyed when simply considering clumps. Attachment points are thus encoded with the local attachment structure, both where they occur in the clump and how they are connected to other clumps.

Definition 4.2.3. An *attachment environment* is an attachment point together with the orientation of attachments indicated (in other words, the neighborhood of the attachment point).

A *clump environment* is a clump (or group of double-linked clumps) with the attachment points distinguished and the orientation of attachments indicated (the neighborhoods of the attachment points).

Figure 4.9 lists the different types of attachment environments possible. Note that collinearity and shape of the clumps are the only importance here, which for example is why a 'V' shape is not included (it is the same as the second environment listed, two single noncolinear lines). Notice that attachments of double-linked clumps as per Figure 4.8 are not included in the list since we join them into a single object prior to considering attachment environments.

Maintaining the attachment point mapping within each clump and including the attachment structure within each attachment point vertex allows us to determine isomorphism of line segment diagrams using suitably modified graph isomorphism algorithms. Using labels we store the clump and attachment environments with the clump adjacency graph, and maintain the mapping of attachment point vertices to attachment points within the clumps.

Figure 4.10 illustrates the full labelled graph encoding of the diagram from Figure 4.6. The numbers and letters within the graph vertices indicate isomorphic

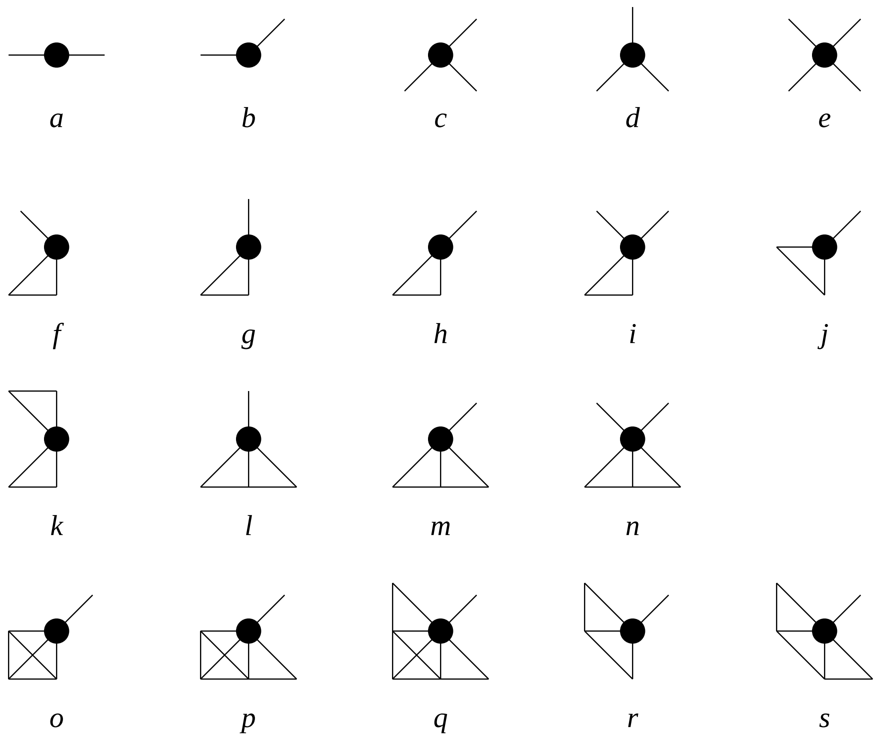


Figure 4.9: List of attachment environments.

clump environments or attachment environments. White points within the clumps distinguish their attachment points.

From now on we assume a clump adjacency graph is labelled with clump environments and attachment environments as described above.

Theorem 4.2.4. Isomorphism of clump adjacency graphs is equivalent to isomorphism of line segment diagrams.

Proof. Let $D_1 = (P_1, L_1)$ and $D_2 = (P_2, L_2)$ be two line segment diagrams, and let $A_1 = A(D_1)$ and $A_2 = A(D_2)$ be the associated labelled clump adjacency diagrams. To simplify discussion, let us name some mappings:

$$\begin{tikzcd} D_1 \arrow[r, "\pi"] \arrow[d, "\alpha"'] & D_2 \arrow[d, "\beta"] \\ A_1 \arrow[r, "\rho"] & A_2 \end{tikzcd}$$

Let α (β) map a point of D_1 (D_2) into the vertices of A_1 (A_2) containing the point, and α_A (β_A) map only into attachment environment vertices. Analogously, let α^{-1} (β^{-1}) map a vertex of A_1 (A_2) into the set of points of D_1 (D_2) mapping into that

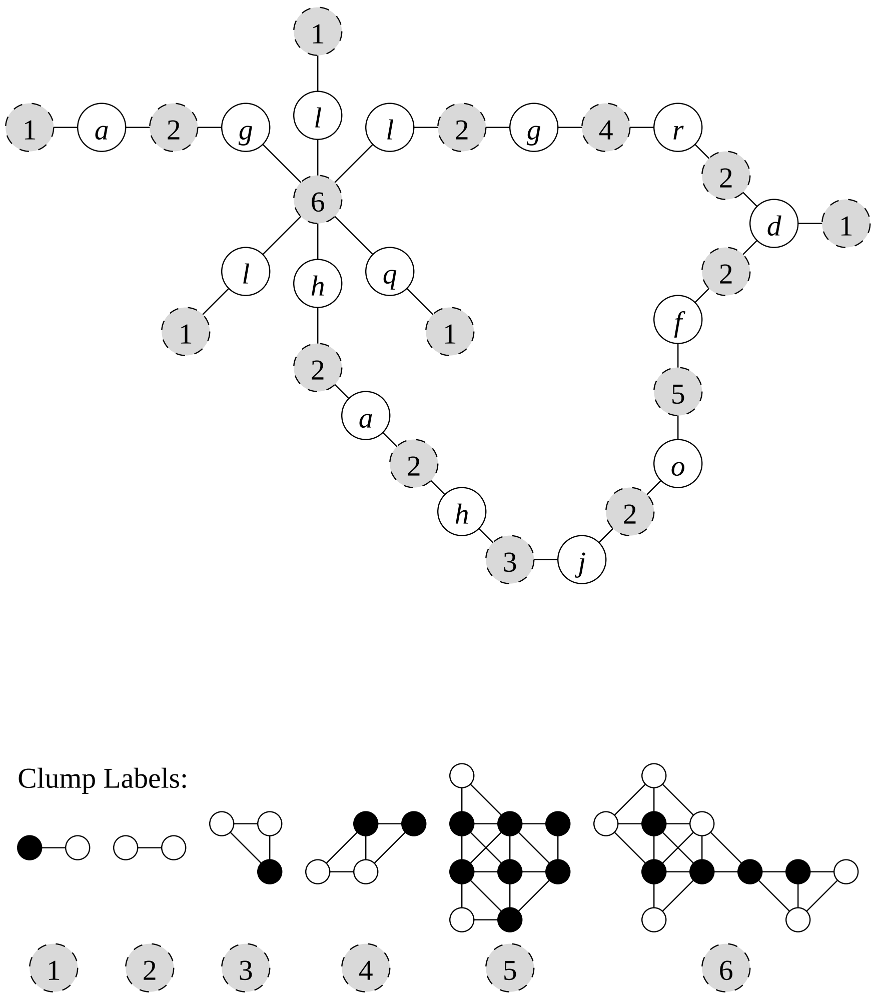


Figure 4.10: The labelled clump adjacency graph encoding.

vertex by α (β). The mappings π and ρ will be constructed or assumed below as appropriate.

We begin with the reverse direction. Assume D_1 and D_2 are isomorphic, call the isomorphism $\pi : P_1 \to P_2$. Since the lines of each diagram are in correspondence by π, Corollary 4.1.5 tells us that the clumps of D_1 are mapped to clumps of D_2. Hence, attachment points are also mapped to attachment points.

We construct the function $\rho : V(A_1) \to V(A_2)$ as follows. Let $v \in V(A_1)$. If v is an attachment environment, it contains a single grid point $p = \alpha^{-1}(v)$. Let $\rho(v) = \beta_A(\pi(p))$, the attachment environment containing $\pi(p)$. On the other hand, if v is a clump environment, it contains at least one 2-line (p_1, p_2). This is mapped to a 2-line $(\pi(p_1), \pi(p_2))$ in D_2, which appears in a unique clump environment; let $\rho(v)$ be this clump environment.

We claim that ρ is an isomorphism between A_1 and A_2. Consider some edge $(u, v) \in E(A_1)$. Assume without loss of generality that u is a clump environment vertex and v is an attachment enviroment vertex. There is some grid point shared between u and v (namely the attachment point $\alpha^{-1}(v)$), call it p. Then $\pi(p)$ is an attachment point in D_2 which corresponds to attachment vertex $\beta(\pi(p))$, which is exactly $\rho(v)$. But $\rho(u)$ contains the 2-lines of u according to the isomorphism π, hence $\pi(p) \in \rho(u)$. Since $\pi(p)$ is in both $\rho(u)$ and $\rho(v)$, we have $(\rho(u), \rho(v)) \in E(A_2)$. The reverse direction is analogous, thus A_1 and A_2 are isomorphic.

For the other direction, assume that A_1 and A_2 are isomorphic and call this isomorphism $\rho : V(A_1) \to V(A_2)$. Considering the partition of D_1 and D_2 into clumps, we construct the mapping $\pi : P_1 \to P_2$ sending attachment points in D_1 to attachment points in D_2. Specifically, if p is an attachment point in D_1, it appears in exactly one attachment vertex $v = \alpha_A(p)$ in A_1. Similarly, $\rho(v)$ corresponds to exactly one attachment point q in D_2. Hence we define $\pi(p) = q = \beta^{-1}(\rho(\alpha_A(p)))$ according to these inclusions.

Consider now the points of D_1 which are not attachment points. They are in clumps. Suppose $p \in P_1$ is in a solid clump c. Then $\alpha(p)$ is a clump environment vertex mapped to $\rho(\alpha(p))$ in the other diagram. So $c' = \beta^{-1}(\rho(\alpha(p)))$ represents an equivalent clump in D_2, which is also solid. From the attachment points mapping π defined above and solidity of the clumps, we can extend π into a mapping for the entire clump c into clump c'. For non-solid clumps, the same idea can be used, except lines must explicitly be maintained (using the mapping pi and the attachment/clump environments).

We claim π is an isomorphism. Consider a line $l \in L_1$. If l is entirely within a clump of D_1, it is mapped to a line within a clump of D_2. If l passes through more than one clump, it will pass through an attachment point p. This attachment point is mapped to an attachment point in D_2 by the isomorphism ρ. The line on either side of p is within a clump and respected by π. Since the attachment environment is respected by ρ, the line will be respected through the clump, hence $\pi(l)$ is a line in L_2. The reverse mapping is analogous. □

Recalling that polynomial time algorithms exist for determining isomorphism of planar graphs, the following theorem suggests the utility of the clump adjacency graph object.

Theorem 4.2.5. Clump adjacency graphs of a line segment diagram are planar.

Proof. Consider the edges in a clump adjacency graph: they join an adjacency point with a clump. For each adjacency point, consider the set of 2-lines (not necessarily maximal) including this grid point. The set of edges incident to this adjacency point in the CAG is a subset of this set of 2-lines, with the restriction that only one 2-line per clump can be chosen. So the edges in the CAG is a particular subset (with possible repetition) of 2-lines from the line segment diagram.

The grid point set of a line segment diagram is embedded in the plane. Consider the associated line segment diagram embedding in the plane and the 2-line crossings (i.e., those line intersections not occurring at a grid point). The only such crossing is diagrammed in Figure 4.11. The two solid lines cross away from a grid point.

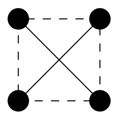


Figure 4.11: Line crossings in a line segment diagram not at a grid point.

It is clear that the dashed lines are forced in such a situation. Hence all four grid points belong to the same maximal clump. So no such crossings need appear in the clump adjacency graph: if we chose one of the solid lines in our subset of 2-lines, we can choose one of the dashed lines instead. Thus a crossing-free CAG embedding in the plane can be constructed from the planar embedding of the grid point set. □

4.3 The Isomorphism Algorithm

Minding Theorem 4.2.5, we now adapt a planar graph isomorphism algorithm to complete our LSD isomorphism algorithm. We reference the results of Section 2.2.1 for background on planar graph isomorphism.

We draw attention to the algorithm of Hopcroft and Tarjan as documented in [HT72]. It is not the asymptotically best algorithm — it runs in $O(|V| \log |V|)$ time — but we found that it served well for adaptation. We outline the Hopcroft and Tarjan algorithm and indicate our modifications.

Roughly the algorithm divides a line segment diagram into connected components, and finds clumps (storing information about their adjacency environment). Clumps are useful because they are easy to process for assignment of isomorphism codes. We form the clump adjacency graph, which happens to be planar. We can apply the ideas of known algorithms to efficiently test clump adjacency graphs for isomorphism. Our algorithm divides the clump adjacency graph into biconnected components, which are related by inclusion of articulation points. The relation defines a tree, so when the nodes are labelled with isomorphism codes, we can move through the tree from the leaves revising our labelling until we obtain a single number for the graph. The question is how to assign a unique isomorphism code to a biconnected component. The answer is to divide it into triconnected components (using a suitable definition to make such components unique) and do the same thing. This leaves the question of how to code the triconnected components, which is the only true difficulty in the process. Fortunately, triconnected components of planar graphs have a unique embedding in the sphere (up to looking from the inside or outside), which provides us an efficient technique to assign the codes.

Since isomorphism of triconnected components is the only difficult portion of our algorithm, we devote special mention and encapsulate our discussions as Algorithm 1. We return to the main process as Algorithm 3.

4.3.1 Dividing Up the Clump Adjacency Graph

We now describe the details of the algorithm. Given a connected clump adjacency graph A, it is subdivided into biconnected components, then further subdivided into triconnected components. We remind the reader that the biconnected components of a graph are unique, and that (when using Tutte's definition [Tut66]) the triconnected components are also unique.

The graph A is represented by a tree T'_A, with one vertex for each biconnected component B and one vertex for each articulation point a. The vertex v_B is adjacent to the vertex v_a in T'_A if B contains a. The leaves of this tree are called *2-leaves*.

Similarly, each biconnected component B is itself represented by a tree T''_B, with one vertex for each triconnected component C within B and one vertex for each cut pair (biarticulation point pair) (a, b). The vertex v_C is adjacent to the vertex v_{ab} in T''_B if both a and b are in C. The leaves of this tree are called *3-leaves*.

Consider now the 2-leaves, and in particular the 3-leaves contained therein. The next step is to assign numbers to these 3-leaves such that two 3-leaves are isomorphic if and only if their codes are equal.

A 3-leaf has an orientation with respect to its cut pair: we can look either from the front or back of the embedding plane, which is equivalent to switching the points in the cut pair. Hence each 3-leaf is assigned a pair of integers (which are equal exactly when the 3-leaf is symmetric regarding the biarticulation points). Whereas the Hopcroft and Tarjan algorithm tests the 3-leaf for planarity and constructs a planar representation, we are already working with a planar representation, saving some work. It is noted that since the 3-leaf is triconnected, its planar representation is unique (up to viewing from inside or outside). A $O(|V| \log |V|)$ algorithm is applied to partition the isomorphic 3-leaves into equivalence classes.

4.3.2 Assigning Isomorphism Codes to 3-Leaves

We refer to Hopcroft and Tarjan's algorithm for triconnected planar graph isomorphism in [HT73c] and [HT72], and proceed to describe our version, noting the differences. To allow us to speak of the left and right faces to an edge, we now consider an arbitrary orientation of the edge, distinguishing its head and tail.

Hopcroft and Tarjan let λ be a mapping of edges to (adhoc) integers where $\lambda(e_1) = \lambda(e_2)$ if and only if the number of edges on the face to the right of e_1 is the same as the number of edges on the face to the right of e_2 and the degrees of the heads of e_1 and e_2 are the same. We require additional information to properly process clump adjacency graphs, and thus also partition edges based on the isomorphism codes of the clump environments and attachment environments of the heads and tails. We call the new mapping with all these restrictions ρ, which is computed in the first three steps of Algorithm 1.

For each edge e in a 3-leaf, let $f(e, R)$ and $f(e, L)$ be the edges adjacent to the head of e which are to the immediate right and left respectively. Clearly this depends on a planar embedding, which we constructed earlier. We partition the edges into equivalence blocks $B(1), \dots, B(k)$ based on their mapping by ρ. A list $PROCESS$ of block index-direction pairs is created by combining all block indices with the symbols R and L.

While possible, we select a pair (i, D) from $PROCESS$ for processing, where i is a block number and D is either the symbol R or L. Each block $B(j)$ is split into $B(j)$ and $B(j')$ such that no edge in $B(j)$ is to the immediate right of an edge in $B(i)$ and all edges in $B(j')$ are to the immediate right of an edge in $B(i)$ (if one of the new blocks is empty, it is discarded). We update the work list by adding (j', D) to $PROCESS$ if $(j, D) \in PROCESS$, and if $(j, D) \notin PROCESS$ we add only one of (j', D) or (j, D), choosing the index of the smaller cardinality block.

The partitioning algorithm, similar to that presented in [HT73c], is summarized in Algorithm 1 below.

The result is a partitioning of the edges into indistinguishability classes. Following [HT72, HT73c], two edges e_1 and e_2 are distinguishable if there exist edges e_3 and e_4, a primary path p_1 from e_1 to e_3 and p_2 and a corresponding primary path p_2 from e_2 to e_4 such that $\lambda(e_3) \neq \lambda(e_4)$, and indistinguishable otherwise. We extend the function λ to ρ for clump adjacency graphs by including clump attachment code and attachment location information.

Isomorphism of the 3-leaves is simply checking if there is an edge in one graph which is indistinguishable from an edge in the other graph, i.e., some block contains an edge from each graph. As noted by Hopcroft and Tarjan [HT73c], both an

Algorithm 1 Efficient edge partitioning by indistinguishability in a triconnected component (adapted from [HT73c])

```
Input: oriented triconnected clump adjacency graph(s)
Output: partitioning of edges into indistinguishability classes
 1: partition edges by clump environment isomorphism codes
 2: partition edges by attachment environment isomorphism codes
 3: partition edges by $\lambda$ values into blocks $B(1), \dots, B(k)$
                                        {completed the partitioning by $\rho$}
 4: $PROCESS \leftarrow \{1, \dots, k\} \times \{\mathrm{L}, \mathrm{R}\}$      {process left and right for all blocks}
 5: while $PROCESS \neq \emptyset$ do
 6:   choose and remove $(i, D)$ from $PROCESS$
 7:   for each edge $e \in B(i)$ do
 8:     $e_M \leftarrow f(e, D)$, the edge to the immediate left/right of $e$
 9:     $j \leftarrow$ index of block containing $e_M$
10:     move $e_M$ from $B(j)$ to $B(j')$
11:   end for
                              {now only add small groups for reprocessing}
12:   for each $B(j')$ just created do
13:     if $(j, D) \in PROCESS$ then
14:       add $(j', D)$ to $PROCESS$
15:     else
16:       if $|B(j')| \leq |B(j)|$ then
17:         add $(j', D)$ to $PROCESS$
18:       else
19:         add $(j, D)$ to $PROCESS$
20:       end if
21:     end if
22:   end for
23: end while
```

orientation of the second graph and its reverse orientation (where the orientation of each arc is reversed) must be tested. It may be that two indistinguishable edges are oriented in opposite directions, meaning the partitioning algorithm cannot match them. By reversing the orientation on all edges for one graph, we can now ensure matching of indistinguishable edges.

This completes the algorithm for determining isomorphism between triconnected clump adjacency graph components, and is summarized in Algorithm 2 below.

Algorithm 2 Efficient isomorphism of triconnected components

```
Input: two triconnected clump adjacency graphs, $G$ and $H$
Output: whether the two graphs are isomorphic
1: $\overrightarrow{G}$, $\overrightarrow{H}$ ← arbitrary orientation on edges of graphs $G$, $H$
2: $\overleftarrow{H}$ ← reverse orientation of $\overrightarrow{H}$
3: partition edges of $\overrightarrow{G}$, $\overrightarrow{H}$ and $\overleftarrow{H}$ by indistinguishability using Algorithm 1
4: if there is a block containing edges from $\overrightarrow{G}$ and either $\overrightarrow{H}$ or $\overleftarrow{H}$ then
5:   return "yes"
6: else
7:   return "no"
8: end if
```

4.3.3 Combining the Isomorphism Codes

We now assign pairs of integers to the 3-leaves' cut pair vertices such that equality of codes is equivalent to isomorphism. Each 3-leaf is then replaced with a edge joining its biarticulation points in the graph. This creates new 3-leaves within the 2-leaves, and the process is repeated.

Eventually every 2-leaf will be reduced to a single edge. These edges are removed from the graph, with their codes being assigned to the corresponding articulation points. The new 2-leaves are found and the process is repeated. Eventually the graph will be reduced to a single vertex with an attached isomorphism code.

Applying the above algorithm to all clump adjacency graphs from two line segment diagrams can answer whether they are isomorphic: sort the isomorphism codes for each line segment diagram and check for equality.

The entire algorithm is sketched in Algorithm 3 below. We emphasize the use of clump and attachment environments in lines 16 and 20. Knowledge of the environments is critical to ensure that the isomorphism coding (equal codes iff isomorphic) is consistent during graph reductions: if two clumps have equal codes, not only must their respective sets of neighboring clumps have equal codes, but the neighboring clumps must be attached at corresponding locations, in corresponding ways.

Algorithm 3 Isomorphism of line segment diagrams (adapted from [HT72])

```
Input: two line segment diagrams
Output: whether the diagrams are isomorphic
 1: partition line segment diagrams into connected components
 2: for each connected LSD component do
 3:   partition component into maximal clumps, storing clump environments and
      attachment environments
 4:   join adjacent double-linked clumps
 5:   assign isomorphism codes to clump environments
 6:   construct clump adjacency graph
 7:   repeat
 8:     partition graph into biconnected components
 9:     construct tree $T'$ of biconnected components and articulation points
10:     for each 2-leaf do
11:       repeat
12:         partition into triconnected components
13:         construct tree $T''$ of triconnected components and cut pairs
14:         partition 3-leaves into labelled isomorphism classes using Algorithm 2
15:         for each 3-leaf do
16:           add its isomorphism codes to its biarticulation vertex (the parent)
17:           replace leaf in graph with edge
18:         end for
19:       until the 2-leaf is reduced to a single edge
20:       add its isomorphism code to its articulation point (the parent)
21:       remove the 2-leaf from the graph
22:     end for
23:   until $T'$ reduced to single labelled vertex
24: end for
25: sort vectors of isomorphism codes for each graph and compare for equality
```

4.4 Examples

We illustrate the isomorphism algorithm presented above with some examples. Figures 4.6 and 4.10 above exemplify the preliminaries: the process of encoding a grid point set as a line segment diagram, and then a clump adjacency graph. We now illustrate the process of comparing two line segment diagrams for isomorphism.

Figure 4.12 lists two line segment diagrams. The left diagram, labelled (a), is quite similar to the right diagram, labelled (b). We apply our isomorphism algorithm

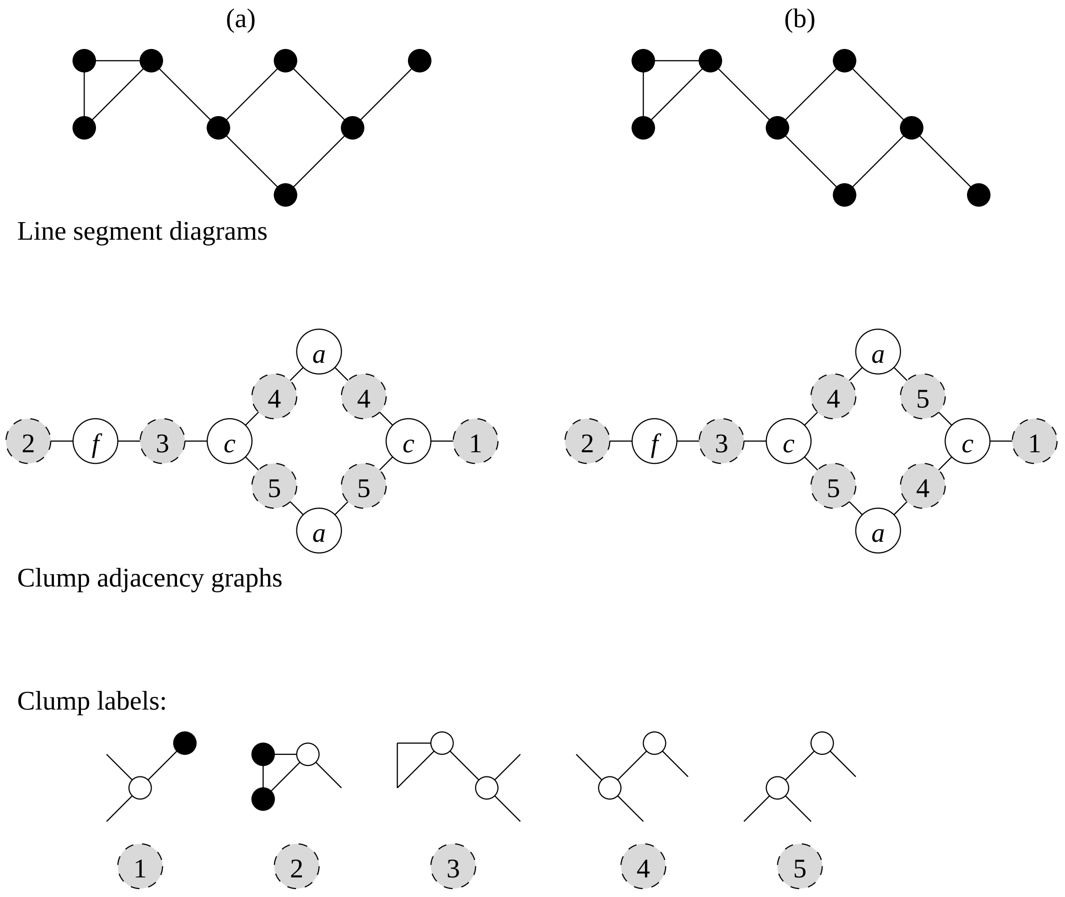


Figure 4.12: Preparing the isomorphism test: an example.

(Algorithm 3) to test whether these diagrams are the same.

Since each diagram is connected, we proceed to identify maximal clumps. Each diagram contains two types of clumps, the dumbell (the 2-line) and a triangle. There are no double-linked clumps, so we identify attachment vertices and construct attachment environments. When constructing the clump adjacency graphs, as shown in the second part of Figure 4.12, simply storing clumps is insufficient to determine isomorphism. Thus isomorphism codes are assigned to the five clump environments, as shown in the third part of Figure 4.12, and the clump adjacency graphs are constructed. These labelled clump adjacency graphs are now tested for isomorphism.

Let us now concentrate on the left clump adjacency graph. Its biconnected components consist of the 8-cycle (the diamond) and each of the four edges not part of the cycle. The tree T' is simply a path, as illustrated in Figure 4.13. There are no nontrivial triconnected components within any biconnected component, so T'' trivially consists of a single vertex for each node of T'. The 2-leaves are the leaves of the tree T', and the algorithm simply adds arbitrary isomorphism codes to each biconnected component.

This process is illustrated in Figure 4.13. Note that the isomorphism codes (indicated by uppercase letters in the example) are actually generated by Algorithm 2 when processing the 3-leaves. Since this example does not contain any interesting triconnected components, we do not show this process in detail. Instead we present another example to illustrate this assignment.

Consider the line segment diagram in Figure 4.14. Its clump adjacency graph is biconnected (the clump environment labelling is obvious and thus omitted), so its tree T' trivially consists of a single node representing the entire graph.

Partitioning the graph into triconnected components results in nine components, as illustrated. The dashed lines indicate virtual edges added to the components to preserve triconnectedness (roughly, the dashed edge represents a path between the endpoints passing through the "rest" of the graph; this is as Tutte's notation in [Tut66]). Though the vertices are not uniquely labelled, the mapping should be apparent from the figure.

The tree T'' is constructed and contains each triconnected component and each biarticulation pair. Note that each triconnected component has a unique embedding

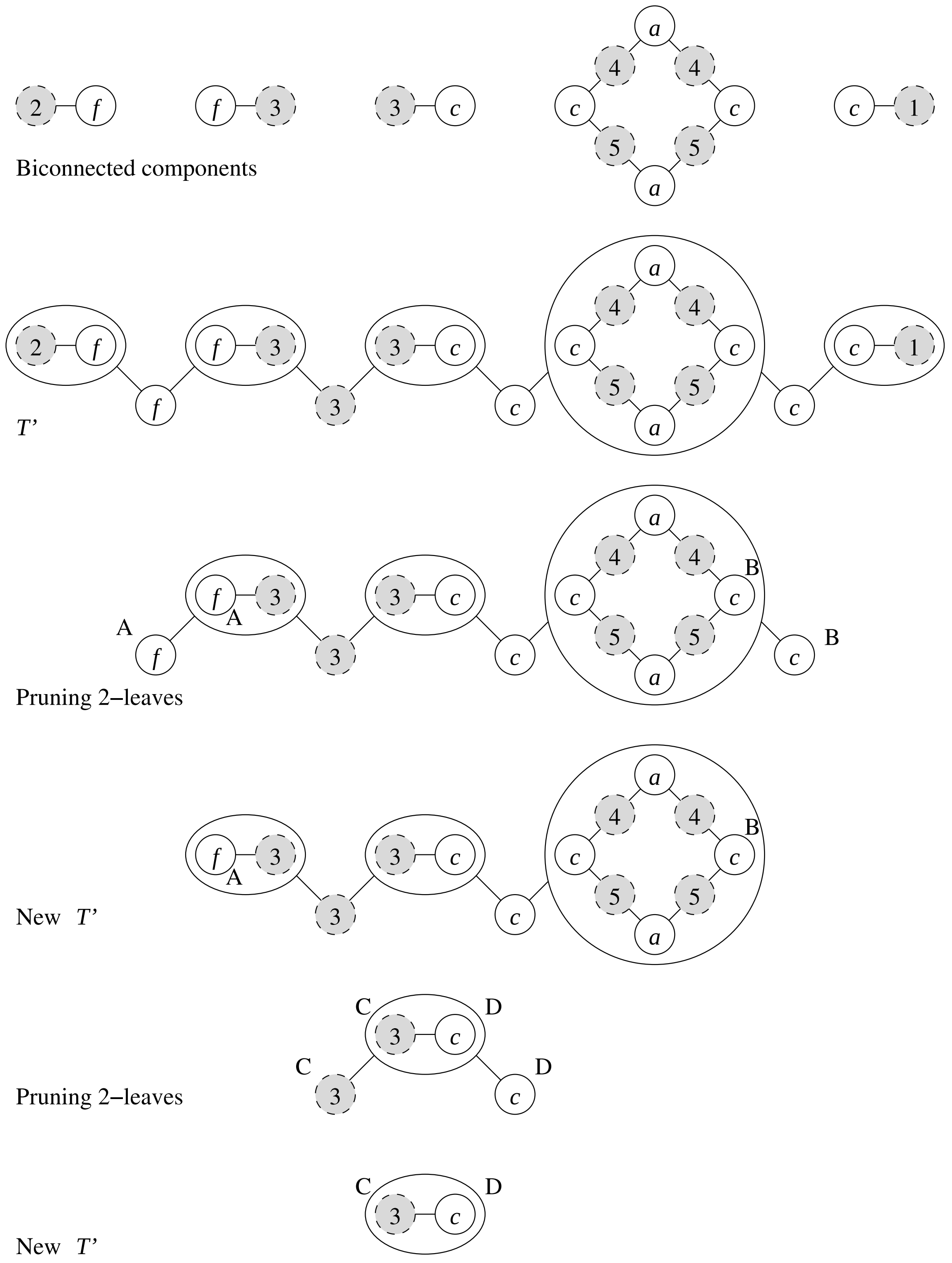


Figure 4.13: Building T' from the biconnected components, using the clump adjacency graph in Figure 4.12(a) as an example.

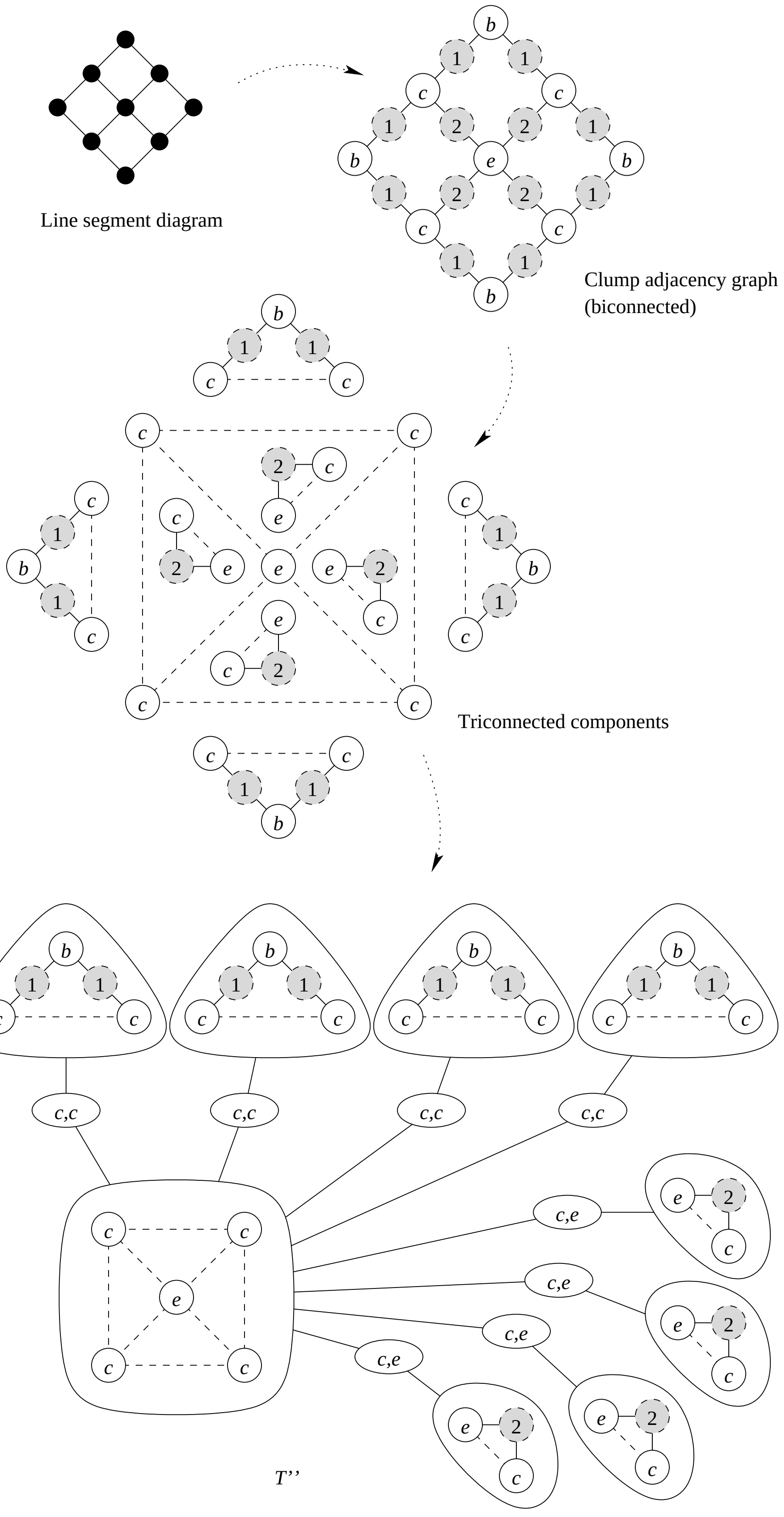


Figure 4.14: Building T'': an example.

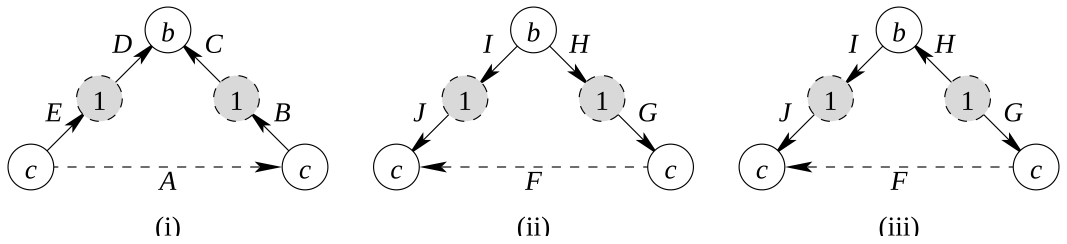


Figure 4.15: Orienting triconnected components.

in the sphere (the reader may study the component c, c, c, c, e to convince himself). The next step of the isomorphism algorithm partitions the 3-leaves into isomorphism classes using the modified Hopcroft and Tarjan partitioning.

Let us for example consider two copies of the 5-cycle from Figure 4.14. We arbitrarily orient the edges, let us say as Figure 4.15(i) and (ii), and we name the edges with uppercase letters as shown. The first step is to initialize the blocks based on ρ. Recall that the ρ partitioning is by clump and attachment codes and the λ function expressing the number of edges defining the face to the right. The first line of Table 4.1 shows this partitioning and initialization of the $PROCESS$ list.

Each following line chooses the first member of the $PROCESS$ list and processes each edge in the block. Any edge to the immediate left or right of the edge being processed is removed from its old block $B(j)$ and placed in a new block $B(j')$. The $PROCESS$ list is updated and any empty blocks are purged (for simplicity).

Since the graphs are cycles, the left and right edges are the same — this is not true in general. It is important to note that at the (4,L) step, no block contains edges from different graphs, so we know that we have not found an isomorphism. Following the process to completion verifies this observation. This is not to say no isomorphism exists, simply that we have chosen a bad orientation on all the edges. It is for this reason both the oriented graph and its reverse orientation are tested. Only if both orientations fail can we conclude they are not isomorphic.

To illustrate this, consider the orientation of the second graph in Figure 4.15(iii). This differs only by the orientation of one edge, and we now find two indistinguishable edges. We follow the same process in Table 4.2. Examining the partitioning into blocks after the final step one notices $B(2)$ contains edge E from (i) and edge G from (iii). Since edges from each graph are in the same indistinguishability class,

Table 4.1: Edge indistinguishability partitioning with bad edge orientation.

(i, D)	Block index B(i)									
	1	2	3	4	5	6	7	8	9	10
—	A, F	B, E	C, D	G, J	H, I					
	$PROCESS = \{$(1,L), (1,R), (2,L), (2,R), (3,L), (3,R), (4,L), (4,R), (5,L), (5,R)$\}$									
(1,L)	A, F	E	C, D	G	H, I	B	J			
	$PROCESS = \{$(1,R), (2,L), (2,R), (3,L), (3,R), (4,L), (4,R), (5,L), (5,R), (6,L), (7,L)$\}$									
(1,R)	A, F	E	C, D	G	H, I	B	J			
	$PROCESS = \{$(2,L), (2,R), (3,L), (3,R), (4,L), (4,R), (5,L), (5,R), (6,L), (7,L)$\}$									
(2,L)	A, F	E	C	G	H, I	B	J	D		
	$PROCESS = \{$(2,R), (3,L), (3,R), (4,L), (4,R), (5,L), (5,R), (6,L), (7,L), (8,L)$\}$									
(2,R)	A, F	E	C	G	H, I	B	J	D		
	$PROCESS = \{$(3,L), (3,R), (4,L), (4,R), (5,L), (5,R), (6,L), (7,L), (8,L)$\}$									
(3,L)	A, F	E	C	G	H, I	B	J	D		
	$PROCESS = \{$(3,R), (4,L), (4,R), (5,L), (5,R), (6,L), (7,L), (8,L)$\}$									
(3,R)	A, F	E	C	G	H, I	B	J	D		
	$PROCESS = \{$(4,L), (4,R), (5,L), (5,R), (6,L), (7,L), (8,L)$\}$									
(4,L)	A	E	C	G	H, I	B	J	D	F	
	$PROCESS = \{$(4,R), (5,L), (5,R), (6,L), (7,L), (8,L), (9,L)$\}$									
(4,R)	A	E	C	G	H, I	B	J	D	F	
	$PROCESS = \{$(5,L), (5,R), (6,L), (7,L), (8,L), (9,L)$\}$									
(5,L)	A	E	C	G	H	B	J	D	F	I
	$PROCESS = \{$(5,R), (6,L), (7,L), (8,L), (9,L), (10,L)$\}$									
	$\vdots$									
(10,L)	A	E	C	G	H	B	J	D	F	I
	$PROCESS = \emptyset$									

Table 4.2: Edge indistinguishability partitioning with better edge orientation.

(i, D)	Block index B(i)								
	1	2	3	4	5	6	7	8	9
—	A, F	B, E, G, J	C, D, H, I						
	$PROCESS = \{(1, L), (1, R), (2, L), (2, R), (3, L), (3, R)\}$								
(1,L)	A, F	E, G	C, D, H, I	B, J					
	$PROCESS = \{(1, R), (2, L), (2, R), (3, L), (3, R), (4, L)\}$								
(1,R)	A, F	E, G	C, D, H, I	B, J					
	$PROCESS = \{(2, L), (2, R), (3, L), (3, R), (4, L)\}$								
(2,L)	A	E, G	C, H, I	B, J	D	F			
	$PROCESS = \{(2, R), (3, L), (3, R), (4, L), (5, L), (6, L)\}$								
(2,R)	A	E, G	C, H, I	B, J	D	F			
	$PROCESS = \{(3, L), (3, R), (4, L), (5, L), (6, L)\}$								
(3,L)	A	E, G	C, H	B	D	F	I	J	
	$PROCESS = \{(3, R), (4, L), (5, L), (6, L), (7, L), (8, L)\}$								
(3,R)	A	E, G	C, H	B	D	F	I	J	
	$PROCESS = \{(4, L), (5, L), (6, L), (7, L), (8, L)\}$								
(4,L)	A	E, G	H	B	D	F	I	J	C
	$PROCESS = \{(5, L), (6, L), (7, L), (8, L), (9, L)\}$								
(5,L)	A	E, G	H	B	D	F	I	J	C
	$PROCESS = \{(6, L), (7, L), (8, L), (9, L)\}$								
	$\vdots$								
(9,L)	A	E, G	H	B	D	F	I	J	C
	$PROCESS = \emptyset$								

the two original triconnected graphs are isomorphic. Hence they would receive the same isomorphism code within T''.

4.5 Correctness

Correctness of Algorithm 3 is now argued using results from the previous sections. The arguments establishing those results will generally not be repeated here.

Consider connected line segment diagrams. Steps 3 through 5 partitions into clumps and joins any adjacent double linked clumps. Solid clumps can only be isomorphic to (a rigid transformation of) themselves, so isomorphism codes suffice. Non-solid clumps are of constant size and are treated with special consideration, namely the clump environment. The clump environment imposes solidity onto double linked clumps, which is easily included into the algorithm by their joining. Hence this assignment of isomorphism codes respects any possible isomorphism.

The algorithm proceeds by checking isomorphism of the clump adjacency diagrams, as per Theorem 4.2.4. Steps 7 through 23 are a straight-forward adaptation of Hopcroft and Tarjan's algorithm: the clump adjacency graphs are converted into trees T' of biconnected components and each biconnected component is assigned an isomorphism code. Well known labeled-tree isomorphism is applied to generate an isomorphism code for the entire clump adjacency graph. Step 20 must respect the clump and attachment environments to properly propogate isomorphism codes up the tree. The isomorphism code must be added respecting where the attachment point is located. Step 25 simply combines the results for connected components in the obvious way.

To assign isomorphism codes to the biconnected components, it is partitioned into unique triconnected components, as described in [HT73a]. Such a unique partitioning is always possible by [Tut66]. The triconnected components are organized into a tree T'', allowing application of labeled-tree isomorphism to determine isomorphism of the biconnected components. Step 16 must respect the clump and attachment environments to properly propogate isomorphism codes up the tree.

To assign isomorphism codes to the triconnected components (step 14), one must be more careful. The partitioning is performed using the test in Algorithm 2, which in turn simply uses the edge partitioning procedure in Algorithm 1. Since this is only a minor variation of Hopcroft and Tarjan's partitioning algorithm from [HT73c], we repeat here only those portions of the proof affected by our modifications. The reader is directed to [HT73c, HT72, Hop70a] for the lemmas and theorems relating indistinguishability and isomorphism.

The only notable modification to Algorithm 1 from Hopcroft and Tarjan's is the original partitioning function for the edges. Whereas Hopcroft and Tarjan partition only on λ, we partition both on λ and the isomorphism codes assigned to the vertices. This starting partitioning is finer than Hopcroft and Tarjan's, so we cannot incorrectly answer in the affirmative (claim to find an pair of indistiguishable edges when none exists). We can, however, answer negatively when an indistinguishable pair does exist. But for such an answer to be generated, the pair must be placed in distinct blocks by our algorithm. This can only happen if the edges are distinguished by some other edge (hence distinguishable), or if they were originally placed in different blocks. Thus the only way this can happen is if ρ distinguished them, which means the isomorphism codes did not match. This is precisely the distinction we desire.

Because of the arbitrary orientation imposed on edges of the graphs in step 1 of Algorithm 2, we cannot guarantee the process will find an indistinguishable pair if such a pair exists. The only way this can happen is if the orientation is reversed for each such pair of edges. Hence we also apply the procedure to the graph with reversed edge orientations (steps 2 and 3), guaranteeing indistiguishability will be recognized.

4.6 Complexity

We now discuss the run time complexity of the above algorithm. We will derive an asymptotic upper bound showing this problem is in the class **P**.

We begin with Algorithm 1. We assume sets and blocks can be represented such that addition, selection, deletion elements can be performed in time independent of the size of the set.

Lemma 4.6.1. Algorithm 1 can be computed in $O(|V| \log |V|)$ steps on a oriented triconnected clump adjacenct graph $G = (V, E)$.

Proof. We refer the reader to [HT73c] for a proof that the Hopcroft and Tarjan algorithm is in $O(|E| \log |E|)$. Since the graph is planar, $|E| \leq 3|V| - 6$, showing the algorithm is in $O(|V| \log |V|)$. We simply show our modification cannot increase this complexity.

The only modification we applied in Algorithm 1 is the original partitioning into blocks: we add additional constraints that if e_1 and e_2 are in the same block, the isomorphism codes for the endpoints of e_1 and e_2 are equal and the attachment point locations are the same. This simply means we are using a labelled graph, or indeed we start with a partition of the edges by isomorphism codes and attachment locations and then refine it based on the size of faces and degrees of vertices (its λ value). This operation, depending whether we consider such partition part of input or to be computed, is clearly either free or takes a linear number of steps — well within the $O(|V| \log |V|)$ bound. □

We now turn to the entire algorithm to show its complexity.

Theorem 4.6.2. Algorithm 3 can be computed in $O(n^2)$ steps, where n is the number of grid points in a line segment diagram.

Proof. Partitioning a line segment diagram into connected components takes only a linear number of steps (for example, by a graph traversal algorithm such as breadth-first search or depth-first search). Partitioning components into maximal clumps is linear by Theorem 4.1.6. Finding clump protrusion information is linear in the size of each clump, which aggregates into linear in the size of the LSD. Finding double-linked clumps is a byproduct of finding protrusion information, so joining adjacent double-linked clumps is free.

Assigning isomorphism codes to maximal clumps requires the isomorphism question to be answered, and some method for assigning the codes. Theorem 4.1.9 states isomorphism is linear in the size of the clump, and trivial partitioning algorithms can operate within n^2 steps. Summing over all clumps, partitioning clumps into isomorphism classes is $O(n^2)$. Constructing the clump attachment graph simply uses information already obtained, thus linear in the size of the line segment diagram.

The outer loop is dominated by the triconnected component partitioning process, which is basically Algorithm 1. This partitioning takes $O(|V| \log |V|)$ steps, which aggregated over all triconnected components is $O(n \log n)$ steps. Finding biconnected and triconnected components each take $O(n)$ steps. We refer to [HT72] for the analysis of the original algorithm.

We end with sorting the two vectors of isomorphism codes. There are $O(n)$ components, thus the sort takes at most $O(n \log n)$ steps. Checking equality of vectors requires at most $O(n)$ steps, completing the algorithm.

Thus, the number of steps required is dominated by partitioning maximal clumps and partitioning triconnected components into isomorphism classes. The former can be performed in $O(n^2)$ steps, dominating the latter's $O(n \log n)$ steps. □

4.7 Enumeration

The ability to test for isomorphism naturally suggests examination of the enumeration problem. Can we generate all non-isomorphic line segment diagrams efficiently?

It is not clear how to adapt the algorithm from the previous section into a generation algorithm. Yet, using elementary techniques, one can systematically generate each possible line segment diagram, but with many duplicates. The duplicate line segment diagrams can be recognized and discarded using any isomorphism algorithm: either brute-force backtracking, McKay's *nauty*, or the above presented algorithm.

To illustrate the relative growth in number of line segment diagrams as we increase the number of grid points, we completed the above process using a simple brute-force backtracking isomorphism algorithm. This process is far from efficient, and is only of interest to the curious.

Table 4.3 illustrates the relative growth of grid point sets and line segment diagrams. The first data column indicates the total number of different grid point sets on n grid points. The second column indicates the number of these grid point sets which are distinct given our concept of isomorphism. The last column gives the number of distinct line segment diagrams using n grid points. The representation we used only permitted generation up to eight vertices due to space limitations, though other researchers have examined similar problems up to ten vertices. We present their results for 9 and 10 vertices to further illustrate the rate of growth.

Table 4.3: Number of grid point sets and nonisomorphic line segment diagrams.

n	All GPS's	Nonisomorphic GPS's	Nonisomorphic LSD's
1	1	1	1
2	4	2	1
3	20	5	3
4	110	22	11
5	638	94	42
6	3832	524	199
7	23592	3031	960
8	147841	18770	4945
9	—	118133*	25786**
10	—	758381*	137988**

* These values are sequence A030222 from [Slo].
** These values are from [MT01].

We comment that grid point sets are equivalent to n-polyplets (polyominoes connected at edges or corners) permitted to contain holes. There is no other combinatorial object known to be similar to line segment diagrams.

We also comment that the results reported by Tegos [MT01] are only correct up to 7 vertices: his encoding only supported grid point set generation on boards containing up to 56 elements (for example, a 7×8 board size). The results he presents in [MT01] beyond 7 vertices are incorrect for number of nonisomorphic grid point sets, but are probably correct for line segment diagrams.

Chapter 5

Conclusions

The goal of this thesis is to capture the information contained within a position during the game of Amazons. A player may want to know when two positions are essentially the same, a necessary step in using knowledge from the past to gain insight towards the next move.

This thesis discussed the progression from a grid point set, a primitive structure, towards more useful encodings such as the line segment diagram. The main problem discussed in this thesis is determining when two line segment diagrams are isomorphic. Deriving an efficient algorithm for solving line segment diagram isomorphism required that the representation graphs be planar, a property realized except for the representation of the moves available within a two by two square (Figure 4.11).

A clump allows all such planarity violations to be grouped within a solid set. A clump, once large enough, is not malleable, and hence can be easily tested for isomorphism. Clumps can be reduced to labeled vertices based on isomorphism classes, and the relationship between clumps can be represented with a planar clump adjacency graph. The significant contribution of this thesis is an efficient algorithm to test isomorphism of clump adjacency graphs, which by trivial extension, answers our original isomorphism question.

By efficient we mean polynomial time: the work performed by this algorithm grows polynomially with the size of the line segment diagrams. If the size of the diagram (in terms of number of points) is n, the time required by this algorithm is bounded by some constant factor of n^2. No claim is made as to the size of this factor in relation to other algorithms, and this factor may be large enough as to make this algorithm impractical in practice. The contribution made by this algorithm is showing that isomorphism of line segment diagrams is polynomial, leading the way to building more efficient algorithms.

For practical use, an encoding using coloured graphs compatible with McKay's *nauty* program is discussed. *Nauty* is one of the fastest isomorphism testing and enumerating programs in the world. It is highly tuned to work quickly in most cases. It is well known that hard instances seldom occur in graph isomorphism, and we see no evidence suggesting our problem is any different. In most cases (especially those encountered on a standard Amazons board) an exhaustive matching algorithm should be sufficient to efficiently answer the isomorphism problem.

5.1 Further Work

We have stressed that the isomorphism algorithm for line segment diagrams derived in this thesis is of theoretical interest, though perhaps not of practical interest. An open question is whether the presented algorithm can be simplified sufficiently to encourage implementation. It would be encouraging to obtain a clean, simple algorithm. As the author has no knowledge of a cleaner isomorphism algorithm for planar graphs, such a development seems unlikely. But there is some recent work concerning planar graphs (for example that of Boyer [BM01]) which may be applicable.

Another direction for future work is to study the time bottlenecks in the algorithm. Can these ideas be applied to modify either Hopcroft and Wong's [HW74] or Fontet's [Fon76] linear time planar graph isomorphism algorithms? Or can any other structural properties be applied to improve the running time or analysis to linear time? The author believes both these approaches to be promising. Considering the structure of planar graphs and their isomorphism algorithms, there is strong evidence suggesting line segment diagram isomorphism is equivalent. Showing this equivalence seems possible.

To build a database of positions, it is necessary to generate all line segment diagrams of a certain size. Is it possible to modify this algorithm to perform enumeration and generation? What is the complexity of generation, and how can a line segment diagram be efficiently encoded for generation?

The graph isomorphism problem is equivalent to the automorphism counting problem. It seems reasonable to assume that the same can be shown for line segment diagrams. If true, does it shed light on the enumeration problem?

Obviously there is still much work to be done in this area. This thesis merely opens the door to a number of new questions. It is hoped that the structures and definitions are sufficient to explore these questions. What other games and fields can be encoded with similar structures? Our notion of clump depends on the points being arranged in a square grid. Eric Mendelsohn (personal communication) made the observation that similar notions likely exist for triangular or hexagonal grids (which are used for example in Chinese Checkers or Hex). With this observation, adaptation of our algorithm to such grids should be possible.

More generally, a line segment diagram is a graph with the notion of line added. When considering the application to other games without the strict point embedding in a grid, our algorithm no longer applies as presented. What can be said with the general line segment diagram? Does it give us more than graphs? Clearly, the notion of a graph (where collinearity is only between adjacent vertices) is subsumed in the notion of a line segment diagram (when the base set is not restricted to lie in a grid). It would be interesting to determine whether the two structures are in some sense equivalent. If so, algorithms for isomorphism, generation and other properties could be translated, possibly yielding better, more understandable algorithms.

Bibliography

[BHZ87] Ravi Boppana, Johan Hastad, and Stathis Zachos. Does co-**NP** have short interactive proofs? *Information Processing Letters*, 25:127–132, May 1987.

[BM01] John M. Boyer and Wendy Myrvold. Simplified $O(n)$ planarity algorithms, 2001. Submitted to Journal of Algorithms.

[CB81] C. J. Colbourn and K. S. Booth. Linear time automorphism algorithms for trees, interval graphs, and planar graphs. *SIAM J. Computing*, 10:203–225, 1981.

[Fon76] M. Fontet. A linear algorithm for testing isomorphism of planar graphs. In *Proceedings of the 3rd Internation Conference on Automata, Languages and Programming*, pages 411–423. Springer-Verlag, 1976.

[GHL$^+$87] Zvi Galil, Christoph M. Hoffmann, Eugene M. Luks, Claus P. Schnorr, and Andreas Weber. An $O(n^3 \log n)$ deterministic and an $O(n^3)$ Las Vegas isomorphism test for trivalent graphs. *Journal of the ACM*, 34(3):513–531, July 1987.

[Hof82] Christoph M. Hoffmann. *Group-Theoretic Algorithms and Graph Isomorphism.* Springer-Verlag, Berlin, 1982.

[Hop70a] John E. Hopcroft. An $n \log n$ algorithm for isomorphism of planar triply connected graphs. Technical Report CS-192, Stanford University, Stanford, California, 1970.

[Hop70b] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. Technical Report CS-190, Stanford University, Stanford, California, 1970.

[HT71] John E. Hopcroft and Robert E. Tarjan. A V^2 algorithm for determining isomorphism of planar graphs. *Information Processing Letters*, 1:32–34, 1971.

[HT72] John E. Hopcroft and Robert E. Tarjan. Isomorphism of planar graphs (working paper). In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 131–152. Plenum Press, 1972.

[HT73a] John E. Hopcroft and Robert E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.

[HT73b] John E. Hopcroft and Robert E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[HT73c] John E. Hopcroft and Robert E. Tarjan. A $V \log V$ algorithm for isomorphism of triconnected planar graphs. *Journal of Computer and System Sciences*, 7:323–331, 1973.

[HW74] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Conference Record of Sixth Annual ACM Symposium on Theory of Computing*, pages 172–184, Seattle, Washington, 30 April–2 May 1974.

[KST92] J. Köbler, U. Schöning, and J. Torán. Graph isomorphism is low for **PP**. *Journal of Computational Complexity*, 2:301–310, 1992.

[Kur30] Casimir Kuratowski. Sur les problèmes des courbes gauches en Topologie. *Fundamenta Mathematicae*, 15:271–283, 1930.

[Lin92] Steven Lindell. A logspace algorithm for tree canonization (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 400–404, Victoria, British Columbia, Canada, 4–6 May 1992.

[Luk80] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. In *Proceedings of the 21st IEEE Symposium on Foundations of Computer Science*, pages 42–49, New York, 1980.

[Luk82] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25:42–65, 1982.

[Mat79] Rudolf Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8(3):131–132, 1979.

[McK] Brendan McKay. The nauty page. `http://cs.anu.edu.au/people/bdm/nauty/`.

[McK81] Brendan McKay. Practical graph isomorphism. In *Congressus Numerantium 30*, pages 45–87, 1981.

[MT01] M. Müller and T. Tegos. Experiments in computer amazons. In R. Nowakowski, editor, *More Games of No Chance.* Cambridge University Press, 2001. To appear.

[Sch88] Uwe Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37:312–323, 1988.

[Slo] Neil J. A. Sloane. The on-line encyclopedia of integer sequences. `http://www.research.att.com/~njas/sequences/index.html`.

[Tor00] Jacobo Torán. On the hardness of graph isomorphism. In *FOCS*, 2000.

[Tut66] W. T. Tutte. *Connectivity in Graphs.* University of Toronto Press, Toronto, 1966.

[Wes01] Douglas B. West. *Introduction to Graph Theory.* Prentice-Hall, Upper Saddle River, NJ, 2nd edition, 2001.

[Zam94] Walter Zamkauskas. Amazons — the rules. Usenet posting, December 1994.

Appendix A

The Rules of Amazons

Amazons, invented by Walter Zamkauskas [Zam94], is a simple game to explain, but a complicated game to play.

The game is played on a 10x10 grid, with two players (red and blue) each having 4 tokens representing amazons. Black tokens are used to indicate landing spots for arrows.

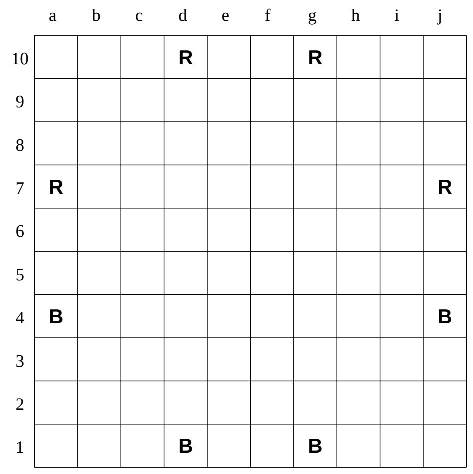


Figure A.1: Initial configuration of the Amazons board.

Each player (red and blue) has 4 amazons initially placed on the board (Figure A.1). Red starts and players alternate turns.

Each turn consists of moving one amazon (like a queen in chess) and firing an arrow (again like a queen in chess) from that stopping location. Both parts are mandatory. The location the arrow is fired is marked with a black token.

No amazon or arrow can move into or move over an occupied location, occupied either by an amazon or a previous arrow. An arrow's landing place becomes a permanent hindrance to movement.

A player who cannot make a legal move loses. The winner is the last player able to make a move.