A Greedy Heuristic for Axial Line Placement in Collections of Convex Polygons

Leonard Hagger

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science

Johannesburg, 2004

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

Leonard Hagger
19 th day of August 2004

Abstract

Axial line placement is one step in a method known as space syntax which is used in town planning to analyse architectural structures. This is becoming increasingly important in the quickly growing urban world of today. The field of axial line placement is an area of space syntax that has previously been done manually which is becoming increasingly impractical. Research is underway to automate the process and this research forms a large part of the automation.

The general problem of axial line placement has been shown to be NP-complete. For this reason, previous research in this field has been focused on finding special cases where this is not the case or finding heuristics that approximate a solution.

The majority of the research conducted has been on the simpler case of axial line placement in configurations of orthogonal rectangles and the only work done with convex polygons has been in the restricted case of deformed urban grids. This document presents research that finds two non-trivial special cases of convex polygons that have polynomial solutions and finds the first heuristic for general configurations of convex polygons.

Acknowledgements

I thank Ian Sanders for his help. I would also like to thank Joseph O'Rourke for inadvertantly giving me many ideas. Pravesh Ranchod also deserves credit for help with the document and helping me with the data structure.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Space syntax [Hillier *et al.* 1983] is used in town planning to describe and analyse architectural structures. There are various steps in this process but the one that is of concern in this document is that of axial line placement (ALP). ALP is one of the harder steps in space syntax so it warrants research. Previously, the process was done manually and with growing problem sizes it is becoming increasingly important to automate axial line placement.

The general problem of ALP has been shown to be NP-complete [Sanders 2002] so it is important to develop heuristics and find special cases that have polynomial solutions. This document presents the first heuristic for ALP in configurations of convex polygons which is the most general version of the problem.

The purpose of this chapter is to give the reader an idea of what axial line placement involves as well as to give an overview of the rest of the document and the research contained therein. The chapter begins with a brief description of the problem of axial line placement and follows with a discussion on the importance of the problem and the importance of this research. The next part of the chapter gives a description of the actual heuristic. This begins with a description of some problems that need to be solved for preprocessing and concludes with a description of the main part of the heuristic. The chapter ends with an overview of the structure of the rest of the document.

1.2 Axial line placement in convex polygons

A configuration of convex polygons is given in a two-dimensional plane, where each polygon in this configuration may not overlap. However each polygon in the configuration may share a its edges with other polygons in the set. The shared line segment where two polygons share an edge is called an *adjacency*, i.e. the part where the polygons edges touch. The problem of axial line placement is to find the minimum number of axial lines that cross the adjacencies in a configuration, where all points on the axial lines must lie inside the polygons.

The lines placed across these adjacencies should be *maximal* i.e. each line should cross as many adjacencies as possible. However, this does not mean that the solution with the longest lines should be chosen. Figure 1.1 shows an example of axial lines being placed across the adjacencies between convex polygons. The polygons in the figure are shaded so there is no confusion between the polygons and the spaces between the polygons. It can be seen in the diagram that no lines cross any unshaded area. If any lines did then it would not be a valid placement.

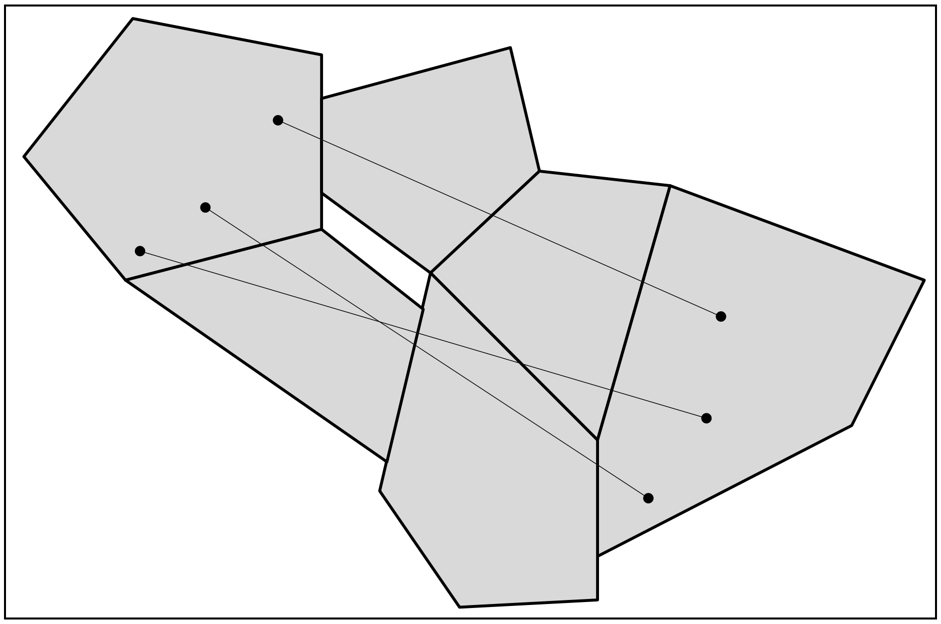


Figure 1.1: Example of optimal axial line placement.

1.3 The importance of axial line placement in convex polygons

Axial line placement has applications in VLSI design and town planning. However, this section focuses on the importance of furthering the field of axial line placement to include convex polygons rather than the practical importance of axial line placement. Sanders [2002] gives a more thorough treatment of the practical applications of ALP.

The problem of axail line placement has been shown to be NP-complete [Sanders 2002], making the development of heuristics imperative. Previously, a large portion of the work done on axial line placement has been in orthogonal rectangles and the application area considers many cases which may not be represented well by rectangles. The research done with rectangles tries to use the geometry of rectangles to get a good solution instead of looking for problems that are similar to the problem of axial line placement in general or focusing on the characteristics of the problem itself. The heuristic presented in this document takes a more general view of the problem.

No heuristic exists for the general problem and the only work done on convex polygons has been on deformed urban grids which is a special configuration of convex polygons that represents networks of streets in a city or town [Wilkins and Sanders 2004; Konidaris and Sanders 2002]. The next logical step for the area of axial line placement is to investigate the problem in its most general form; collections of convex polygons. Finding a good approximation algorithm for the general problem is really important because an algorithm is needed that can be applied to any configuration. This document presents such an algorithm.

1.4 Preprocessing for the heuristic

The heuristic presented in this document can be applied to any configuration of polygons that will be considered in the area of axial line placement. It works by recursively choosing a line based on the current set of lines already chosen. This means that some configuration of lines is required before the algorithm may start. This document suggests the following three methods for choosing a starting configuration.

1. Choose a random polygon and generate lines from it.
2. Choose a polygon based on its number of adjacencies and position in the configuration, and generate lines from it.
3. Detect special configurations of convex polygons embedded in the configuration and use their respective algorithms to place lines.

Methods 1 and 2 are discussed further in chapter 6. Method 3 is desirable because it may cover most of the configuration before the heuristic starts. However, in order for it to work well, the special cases detected should have polynomial time algorithms to place the minimum number of lines in them. Unfortunately, the only special configuration of convex polygons studied previously is deformed urban grids, as mentioned in section 1.3. Deformed urban grids are difficult to detect in general cases so other special cases are required.

The special configurations used by this document for method 3 are as follows.

- Chains of convex polygons
- Stars of convex polygons
- Networks of stars

Networks of stars are made up of stars and the stars are made up of chains. These special cases are discussed in the following three subsections.

1.4.1 Chains of convex polygons

Figure 1.2 shows an example of a chain of convex polygons. Chains of convex polygons are the simplest non trivial configurations for the problem of axial line placement in *arbitrary* convex polygons and form the basis for the other special cases that are discussed in this

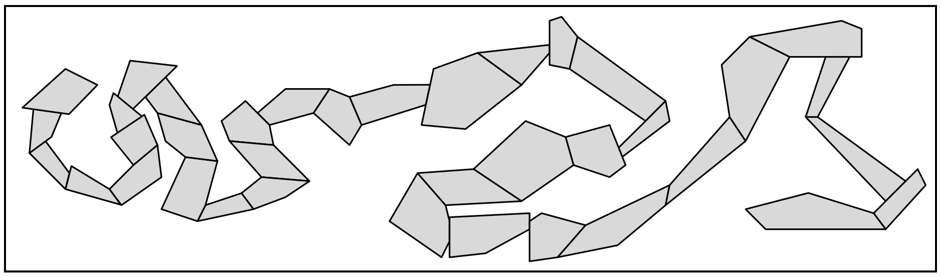


Figure 1.2: A chain of convex polygons

document. Chains of convex polygons are a natural extension of work done previously on chains of rectangles [Phillips 2001; Sanders *et al.* 2000b].

A quadratic time algorithm was developed by transforming the problem of ALP in chains of convex polygons to *interval point cover*.

In addition to their application to the heuristic developed in this document, chains of convex polygons can be applied to other heuristics for the general case that use path finding algorithms.

1.4.2 Stars of convex polygons

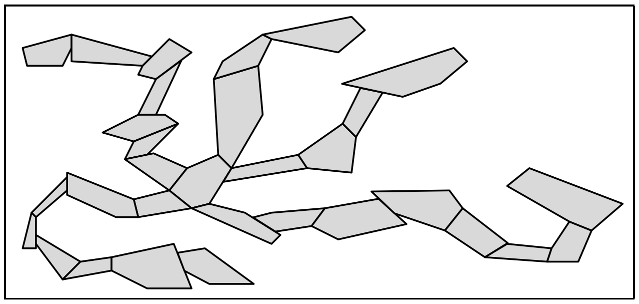


Figure 1.3: A star of convex polygons

Figure 1.3 shows an example of the next special case developed: stars of convex polygons, which are made up of chains. This configuration does not extend previous research done on ALP in rectangles like chains of convex polygons does, and is a completely new special case.

The chain algorithm is used to transform the problem to a well known graph theory problem known as *maximum cardinality matching*. This creates a quadratic time algorithm that solves ALP in stars of convex polygons.

1.4.3 Networks of stars

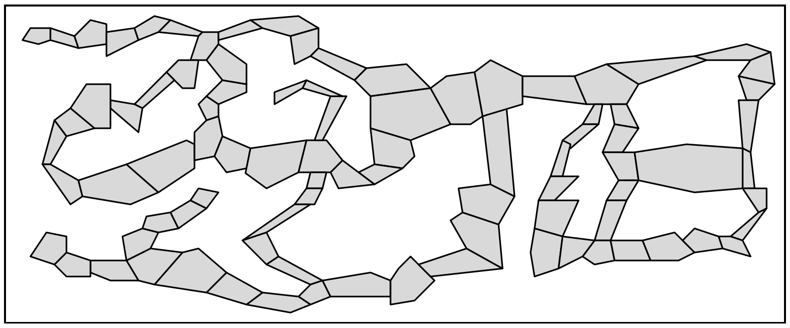


Figure 1.4: A network of stars

The final special case considered by the research presented in this document is networks of stars which are formed by joining stars of convex polygons. An example is shown in figure 1.4. The algorithm developed for networks of stars is a heuristic, unlike the algorithms developed for chains and stars. This special case is considered because of conflicts that can happen when stars are detected in a configuration. This document presents a cubic time algorithm for networks of stars.

The network of stars special case can be useful to areas outside of this research because of its resemblance to a town plan, much like the deformed urban grid mentioned earlier. However, it is more general than a deformed urban grid which may increase its usefulness.

1.5 Development of the heuristic

The heuristic presented in this document first uses some method to find a starting set of axial lines that cross some adjacencies in a general configuration of convex polygons. Then it recursively chooses exactly one line that extends as far as possible from the current set of lines. This process terminates when all of the adjacencies have been crossed. This method is a generalised version of the algorithm developed for ALP in chains of convex polygons.

The line that extends the furthest is chosen from a set of lines generated using a depth first search. This is a major overhead of the algorithm so a data structure is presented in this document to make it more efficient. Efficiency is also increased by limiting the number of lines found by the depth first search but results in a worse solution.

The algorithm developed is $O(ln^3)$ where n is the number of polygons and l is a parameter passed to the algorithm that limits the number of lines found by the search. This research only considers values of l no greater than $O(n^2)$. This is acceptable because the problem is NP-complete and this is the first heuristic developed.

1.6 Evaluation of the heuristic

The solutions found by the heuristic for the general case of axial line placement were evaluated by implementing the heuritic and testing it on randomly generated configurations of convex polygons.

The tests done using the heuristic fall into the following three categories.

- Comparing the various methods for finding a starting set of lines.
- Comparing limits placed upon the heuristic itself.
- Comparing the solutions found by the heuristic to solutions with the smallest number of lines.

Ideally, an approximation ratio would have been found and used to evaluate the heuristic but doing so is outside the scope of this dissertation because of time constraints and the amount of work done in other areas such as the special cases.

1.7 Structure of the dissertation

Chapter 2 is the next chapter in this document and it gives some background to the problem of axial line placement. It discusses the work done on rectangles and one special case using convex polygons, and shows the need for the heuristic developed in this document. Additionally, chapter 2 contains discussions of algorithms that are needed for axial line placement to take place, as well as some algorithms that are used to solve the special cases that were discovered by this research. This chapter forms the basis for the research that is described in the rest of the document.

Chapters 3, 4 and 5 each present a special case and the algorithm that is used to solve it. These special cases are integrated into the heuristic so they are discussed before the heuristic is described in any detail. The first of these chapters discusses chains of convex polygons which is the most basic of the special cases considered. It describes how the problem of placing the smallest number of axial lines in a chain of convex polygons is transformed into interval point cover.

The second chapter dealing with special cases discusses stars of convex polygons and describes how chains are joined to form stars and how the chain algorithm is combined with the algorithm for maximum cardinality matching to find an algorithm.

The third chapter dealing with special cases discusses the network of stars. This chapter describes the conflict that can occur between stars when they are being detected in a general configuration of convex polygons and how the network of stars special case is used to resolve this conflict.

Chapter 6 is the most important chapter in this document as it describes the heuristic produced by this research to place axial lines across the adjacencies in a general configuration of convex polygons. This chapter describes the three methods of finding a starting set of lines and discusses the use of the algorithms for the special cases. The data structure that is used to make the heuristic more efficient is described along with a description of the heuristic itself.

The empirical tests that were done using an implementation of the algorithm are described in chapter 7 where the results of these tests are presented and evaluated. This chapter finds the best configuration of the heuristic and gives reasons for some configurations working badly and suggests ways to improve those configurations.

This is the first attempt at developing a heuristic for the general case of axial line placement, so naturally there will be some areas that could be developed further. These areas are described in chapter 8 where other special cases that have polynomial solutions are suggested as well as some improvements to the heuristic itself.

Chapter 9 concludes this dissertation by summarising the significant contribution that this research has made to the area of axial line placement.

Chapter 2

Background and related work

2.1 Introduction

Axial line placement is a relatively new problem. It originated from a problem known as space syntax, which is a method used to determine how accessible areas in a town are to each other. Axial line placement (ALP) is a step in this process. The problem was brought into a computer science context by Sanders [2002] (see also Sanders [1999] and Sanders and Kenny [2001]). While Sanders [2002] was being written more research was done in this area in the form of honours research reports. These form the bulk of the work done on axial line placement and are reviewed in this chapter. Some terms used in this chapter and in the rest of the document that might be unfamiliar to the reader are defined in appendix A.

This chapter covers the following topics.

- A definition of axial line placement
- A brief history of axial line placement
- A review of research directly related to axial line placement
- A review of work that is directly related to this research
- A review of research that is similar to axial line placement

The history of axial line placement reviews the work done with rectangles where axial lines are placed with arbitrary orientation and the lines are orthogonal. This includes discussions on heuristics and special cases where a polynomial solution exists. Additionally, a special case of axial line placement in convex polygons is discussed.

The review of research directly related to axial line placement contains discussions of work done in the areas of visibility and adjacency detection. These areas contain algorithms that are required before axial line placement can take place.

The review of work that is directly related to this research discusses *maximum cardinality matching*. This area includes an algorithm that is used to solve one of the problems in this document.

Finally, the review of research that is similar to axial line placement includes discussions on the *Art Gallery* problem and the *Stabbing* problem. These areas are considered similar to axial line placement.

2.2 Axial line placement defined

Axial line placement arose from a method called space syntax which was initially proposed by Hillier *et al.* [1983], which is an architectural method used to analyse urban and structural layouts. They define an axial line as the following:

A one dimensional extension of the sight lines from particular spaces.

For the problem of axial line placement, the spaces referred to in the above definition are convex polygons, and the one dimensional extension of the sight lines are represented by any lines that cross the shared edges of the convex polygons but are wholly contained inside the convex polygons. This leads to the following definition of axial line placement (ALP) from Sanders [2002]:

Given a collection of adjacent polygons, find the minimum number of maximum length straight line segments contained wholly inside the convex polygons (axial lines) that will cross every adjacency (shared edge) between the polygons. Each adjacency must be crossed by at least *one axial line.*

A variation on this problem exists where each adjacency can be crossed by *only one* axial line. However, previous work has been done on the variant with multiple crossings and so is chosen to be the focus of this research.

In this document, an axial line is defined by the adjacencies it crosses and its length is defined by the number of these adjacencies. Therefore, the terminology "maximum length" means that each line should cross as many adjacencies as possible. Figure 2.1 gives an example of a collection of convex polygons with axial lines placed upon it. This is an optimal configuration of axial lines that cross the adjacencies in this configuration.

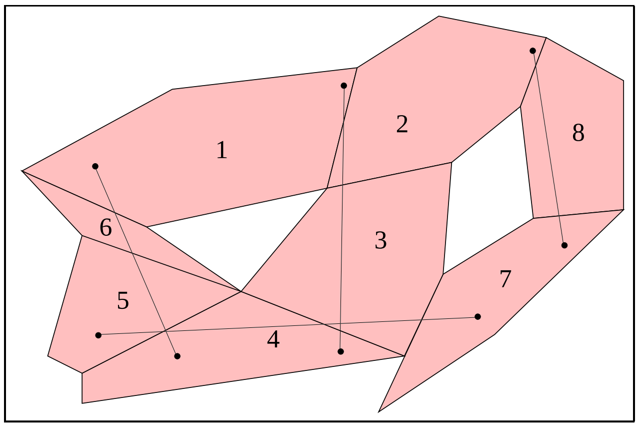


Figure 2.1: Example of axial line placement

Note that the line that crosses the adjacencies between polygons 5, 4, 3 and 7 crosses adjacencies that have already been crossed by other lines. This is necessary because of the maximal lines condition in the definition. If the *single crossing* variant was used then only the adjacency between polygons 3 and 7 would be crossed by that line.

In order to determine if it is possible to place an axial line on the two adjacencies between polygons 2, 8 and 7 it must be known if these adjacencies can "see" each other. This is done using a visibility algorithm which is discussed in section 2.4.2. It may not be necessary to determine the position of an actual line given the application.

Here are three types of axial lines are discussed in this document, particularly in the next section. These definitions are mutually exclusive.

- Redundant – all the adjacencies crossed by this line are already crossed by another line.
- Essential lines – at least one adjacency crossed by this type of line cannot be crossed by another line that is not redundant.
- Choice lines – these are lines that are not essential or redundant.

If a line is redundant then there is a longer line that crosses the redundant line's adjacencies. Therefore, a line is not maximal if it is redundant. By the definition, essential lines cannot be redundant themselves. Furthermore, essential lines are axial lines that *have* to appear in any solution because the lines must be maximal. From the definition of choice lines, it can be seen that they *only* cross adjacencies that are crossed by essential lines or other choice lines.

Determining if a line falls into any of the three categories above can be done by finding all the axial lines that cross all of the adjacencies in a collection. First, redundant lines must be identified and removed, then the essential lines can be identified. The choice lines are the lines that remain. This method is used in several heuristics that are discussed in the following section.

2.3 History of axial line placement

Presently, space syntax [Hillier *et al.* 1983] is carried out manually but there are many aspects that could be automated on a computer such as axial line placement. Axial line placement was originally called *ray guarding* because they seem similar. ALP has been proven to be computationally difficult so it would be difficult for a human to obtain good solutions for large cases and so computers would be better suited for the task.

This section gives an overview of previous work done on axial line placement beginning with its origins. The history of axial line placement started with orthogonal lines in orthogonal rectangles (ALP-OLOR). The next progression was to consider lines with arbitrary orientation in orthogonal rectangles (ALP-ALOR). This research deals with arbitrary lines in convex polygons (ALP-ALCP). One special case of ALP-ALCP has been considered and will be discussed in this section.

2.3.1 Placing orthogonal axial lines in orthogonal rectangles (ALP-OLOR)

In ALP-OLOR (axial line placement - orthogonal lines in orthogonal rectangles), all the lines placed upon the adjacencies are orthogonal to either Euclidean axis. Figure 2.2 shows an example of how axial lines are placed for ALP-OLOR.

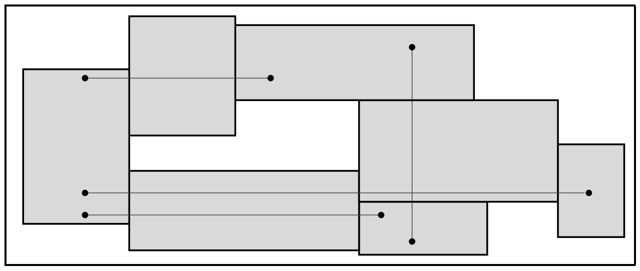


Figure 2.2: Example of axial line placement with orthogonal lines in orthogonal rectangles (ALP-OLOR).

Sanders [2002] shows that ALP-OLOR is NP-complete through a transformation from vertex cover for a planar graph [Lichtenstein 1982]. The transformation from planar vertex cover is done by mapping the vertices in a planar graph to choice axial lines in the problem being considered. Edges in the planar graph are mapped to adjacencies that are crossed by the choice axial lines.

This transformation is done in two steps. First, a planar graph is transformed to the problem of a "stick diagram". Sanders [2002] defines an instance of a "stick diagram" as a collection H of horizontal lines and U of vertical lines such that each vertical line is cut by exactly two horizontal lines, and a positive integer $S \leq |H|$. The question posed is whether there is a set of horizontal lines, $H' \subseteq H$, such that every vertical line in U is cut at least once and $|H'| \leq S$

In this "stick diagram" each vertex in the original graph is mapped to a horizontal line representing a choice axial line and each edge in the original graph is mapped to a vertical line that is cut by the two horizontal lines that represent the two vertices to which the edge is incident. Second the "stick diagram" is transformed into an instance of ALP-OLOR.

A heuristic algorithm

Sanders [2002] also gives an $O(n^2)$ heuristic algorithm to solve ALP-OLOR. The algorithm starts by determining the adjacencies between the rectangles and storing the information in an array, where each element denotes a rectangle and has a linked list attached to it containing the rectangles that are adjacent to it on the right.

The next step is to generate the non-redundant set of axial lines to cross all the adjacencies in a collection of adjacent orthogonal rectangles. Next the redundant, choice and essential lines defined above at the beginning of this section are identified.

First, all the possible orthogonal axial lines which cross the adjacencies between rectangles are generated to form the set of candidate lines. Sanders [2002] provides a polynomial time algorithm to do this. The redundant lines are removed and the essential lines are identified. The essential lines are removed from the candidate set and placed into the final set. The set of candidate lines is now the set of choice lines.

The final step is to choose a subset C of the choice lines such that each of the lines in C crosses at least one adjacency that is not crossed by any other line in C or by an essential line. Sanders [2002] calls this the choice conflict and is the crux of the algorithm and is where the heuristic comes in. The heuristic resolves this conflict by choosing the candidate line that crosses the most uncrossed adjacencies[1]. This line is then made an essential line (actually added to the final set of lines) and is removed from the candidate set. If other lines in the candidate set are made redundant by making this an essential line then the redundant line is removed from the candidate set. This process is repeated until no lines remain in the candidate set.

Special cases

Also considered in Sanders [2002] are special cases where the minimum number of lines can be found in polynomial time. These cases are chains and trees of rectangles and were solved by Sanders *et al.* [2000a]. Figure 2.3 shows an example of a chain and a tree of rectangles.

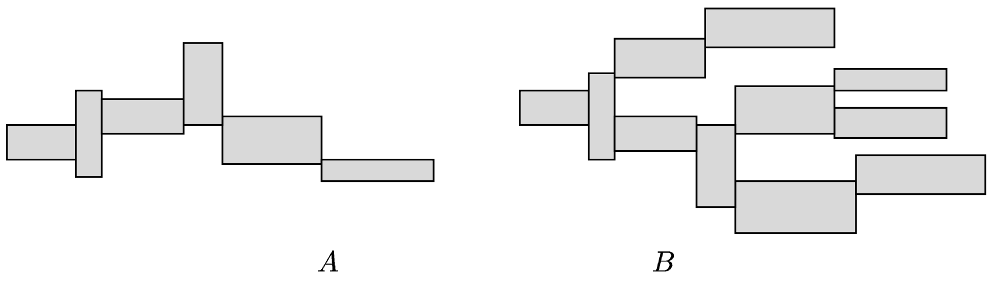


Figure 2.3: Special cases of ALP-OLOR. Example of a chain A and a tree B of rectangles for ALP-OLOR.

Sanders [2002] defined a chain of orthogonal rectangles as *"any collection of orthogonal rectangles where every rectangle is horizontally (vertically) adjacent to at most one other rectangle at each end."*

The research contained in this document generalises this definition by allowing convex polygons in the place of rectangles. The algorithm that places axial lines in chains of rectangles works by first finding the set of "forward" lines and the set of "backward" lines. Then these two sets are merged to form the final set of lines which is the result. The forward lines are generated by considering the left most adjacency in the chain. An axial line is placed upon it and extended to cross as many adjacencies to the right as possible. This line

[1]An uncrossed adjacency is an adjacency that has not been crossed by a line in the solution set

is then placed into the set of forward lines. The process is repeated starting with the left most uncrossed adjacency and stops when all adjacencies have been crossed by a forward line. The backward lines are generated in much the same way except the starting point is the rightmost adjacency. From there the algorithm proceeds backwards. The merging is done by merging the left most lines in both sets in succession. The algorithm presented in this document that places lines in chains of convex polygons uses a slightly different approach but results in a similar solution.

Next are the trees. A tree of orthogonal rectangles is a collection of adjacent orthogonally aligned rectangles, where each rectangle is joined on the left (right) end to at most one rectangle and on the right (left) to zero or more rectangles.

An $O(n^2)$ algorithm was developed that is surprisingly similar to the chain case. The algorithm begins by generating the forward lines in much the same way as the chain algorithm. The backward lines are now called "leaf" lines and are generated starting from the adjacency of each "leaf" of the tree (note that there can only be one adjacency for each leaf polygon). Overlapping forward and backward lines are merged in a similar way to the chain algorithm.

This section discusses work done on configurations of rectangles where only orthogonal lines could cross the adjacencies. The following section, discusses work done on configurations where the polygons are still rectangles but the lines are allowed to be of arbitrary orientation.

2.3.2 Placing axial lines with arbitrary orientation in orthogonal rectangles (ALP-ALOR)

This section deals with ALP-ALOR (axial line placement – arbitrary lines in orthogonal rectangles). Figure 2.4 demonstrates ALP-ALOR. ALP-ALOR was shown to be NP-complete in Sanders [2002 1999] by transforming from *biconnected* planar vertex cover. The proof follows along the same lines as for ALP-OLOR. The first step is to transform biconnected planar vertex cover to "stick diagram". The second step is to transform "stick diagram" into ALP-ALOR.

Heuristics

Sanders [2002] discusses some possible heuristics or ALP-ALOR(also in Sanders and Kenny [2001]). In ALP-OLOR (see section 2.3.1) the first step in the heuristic was to generate all possible axial lines. However, it has not been proven that there are a polynomial number of non-redundant axial lines of arbitrary orientation so heuristics are implemented at this level. Generating all possible lines for convex polygons has the same problem.

The first heuristic suggested was to extend the lines to all neighbours. This started with a rectangle on the outer edges of the configuration and extended all axial lines that originated from that rectangle as far into the other rectangles as they could go. In other words at each step each rectangle is considered in turn. For each neighbour of this rectangle an axial line

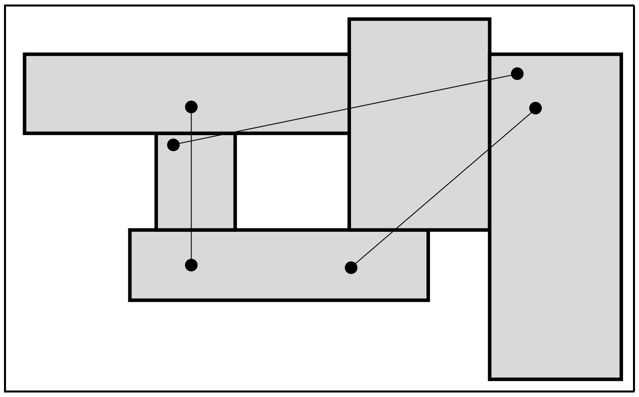


Figure 2.4: Example of axial line placement with arbitrary lines in orthogonal rectangles (ALP-ALOR.

is placed that starts in the original rectangle, passes through the current rectangle and into its neighbour (if such a line exists of course). The problem with this algorithm is it misses out many lines that may be part of an optimal solution so is unlikely to come close.

The second heuristic tries to deal with the vertical and horizontal adjacencies in different passes. The vertical pass would traverse the rectangles in order from top to bottom based on the top y-coordinate. Only axial lines would be placed on horizontal adjacencies. The horizontal pass works analogously. This leaves out many lines but is more efficient than the first heuristic.

The next heuristic suggested is that of finding the longest chain of rectangles that can be crossed by one axial line. This is done by first identifying *extreme rectangles*. An extreme rectangle is defined as a rectangle with no neighbours on both the left and right and the top and bottom sides. Then for each extreme rectangle all possible chains are generated. The longest chain is identified and it is determined if an axial line can be placed upon this chain. If so then the line is added to the final set. If not then the next longest chain is considered. The process continues until an axial line is placed or there are no more chains. Some adjacencies may not be crossed at the end of this process so the current policy crosses each uncrossed adjacency with a single axial line. This heuristic requires development since the requirement of placing a line across the configuration is unreasonable. A different method to generate the chains was suggested by Sanders [2002] that creates chains that could possibly have a line going across. This involves detecting places were it would be impossible to place an axial line and removing them from the chain.

Special cases

Special cases considered for ALP-ALOR have been chains, circular chains and “acyclic” trees. Figure 2.5 shows an example of an “acyclic” tree and a chain of rectangles. These are considered by Phillips [2001] and all three are shown to have polynomial solutions. The definition of a chain used is slightly different to the definition given for ALP-OLOR. Vertical and horizontal adjacencies are allowed. In the ALP-OLOR definition only horizontal or only vertical adjacencies are allowed. Not surprisingly, the algorithm to place axial lines with arbitrary orientation upon a chain of orthogonal rectangles is very similar to the orthogonal lines case. Forward and backward lines are created then merged together to get the final set. This $O(n^2)$ algorithm for chains of rectangles may have been generalised to obtain a solution for chains of convex polygons but due to comments made by O'Rourke [2002], a slightly different approach was taken.

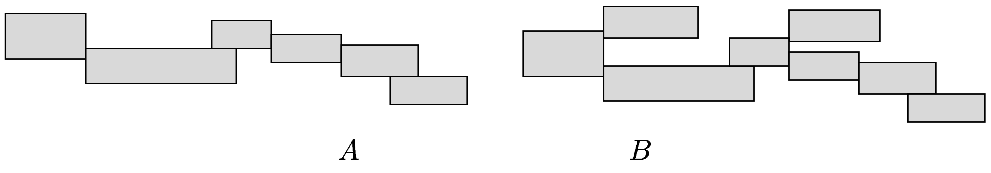


Figure 2.5: Special cases of ALP-ALOR. Example of a chain A and a tree B of rectangles for ALP-ALOR.

The next case considered is the *circular* chain of orthogonal rectangles. A circular chain of orthogonal rectangles is basically a chain of orthogonal rectangles but *all* of the rectangles are adjacent to two other rectangles in the collection forming a loop. The approach to solving this was to generate n chains of rectangles with $n + 1$ rectangles in each, where n is the number of rectangles in the circular chain. Each chain would have the same starting and ending rectangle. Then the normal chain algorithm is applied to each chain and redundancies are then removed from the end points. The chain with the least number of axial lines placed upon it would then be the final result. This is an $O(n^3)$ algorithm. Generalizing this algorithm to the convex case would be direct since the only change would be the chain algorithm part.

The final case considered, is what Phillips [2001] refers to as the *acyclic tree*. The definition for an acyclic tree is relatively complicated with many constraints that are based upon the number of adjacencies that each rectangle can have and on which side these adjacencies can occur. Many constraints are placed upon it in order for it to have a polynomial solution. For this reason, trees of convex polygons are not considered in this research. The algorithm basically works by finding chains in the tree and solving those cases. Then these solutions are merged to obtain the final solution.

2.3.3 Placing axial lines with arbitrary orientation in convex polygons (ALP-ALCP)

Previously, ALP-ALCP has not been given much attention and the only special case considered so far is that of deformed urban grids (see section 2.3.4). Sanders [2002] shows that it is NP-complete by stating that ALP-ALOR is a special case of ALP-ALCP. ALP-ALCP is the problem that this document addresses by finding two, non-trivial special cases that have polynomial solutions and developing the first heuristic that can be applied to any problem that can be considered in the area of axial line placement. This means that this document solves many problems in ALP-ALCP and opens up the area for future research.

2.3.4 Placing axial lines with arbitrary orientation in deformed urban grids

The problem of placing axial lines with arbitrary orientation in deformed urban grids is a special case of ALP-ALCP and is shown to be NP-complete in Wilkins and Sanders [2004] by a transformation from 3SAT. The original definition of the problem is given by Sanders [2002] and is based on the idea of a complete grid [Gewali and Ntafos 1993] – the *complete two-dimensional grid* of size n is the graph with vertex set $V = \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$ and the edge set $E = \{\{(i,j),(k,m)\} : |i-k| + |j-m| = 1\}$ where all edges are parallel to the major axes. In a geometric setting, the grid edges can be thought of as corridors and the grid vertices as intersections of corridors. A (partial) grid is any subgraph of the complete grid. In a simple urban grid the corridors and intersections are given dimension. A simple urban grid is thus a polygon with holes that is the union of rectangles and squares. The rectangles are the "corridors" and have width w and length l. The squares are the "intersections" and have width w. Each corridor must begin and end with an intersection. The short side of a corridor should be flush with the side of an intersection.

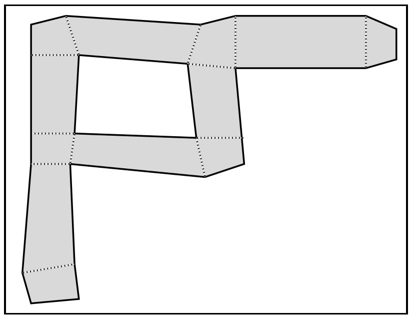


Figure 2.6: Example of a deformed urban grid.

No intersection can be adjacent to another intersection. A horizontal (or vertical) sequence of corridors and intersections is called a "thoroughfare". A deformed urban grid (DUG) is generated from a simple urban grid and is formed by deforming (moving the vertices of) the intersections, therefore deforming the corridors since they share the same vertices. Corridors, intersections and thoroughfares in DUGs are similar to those in simple urban grids. An example of a deformed urban grid is given in figure 2.6.

The degree to which the intersections should be deformed is also an issue. Consider a simple urban grid of dimensions w and l where $w < l$. To form the deformed urban grid the vertices of any intersection may be moved around but only so that they still lie inside the original simple urban grid's intersection. The intersections and corridors must remain convex.

A heuristic algorithm was also developed in Konidaris and Sanders [2002] that had a similar framework to that of the heuristic algorithm developed for ALP-OLOR. All the axial lines are first generated and maximized. Redundant lines are removed and the remainder are placed into a candidate set. The essential lines are identified and removed from the candidate set then placed in the final set. Then some policy is used to resolve the choice conflict (or deadlock).

The resolution of the choice conflict is also similar. The line that crosses the most uncrossed adjacencies is removed and placed in the final set. The essential lines are identified once again and the deadlock removal is applied. This is repeated until there are no more candidate lines.

The major difference between the ALP-OLOR algorithm and this one was that of placing all the lines. This can be done efficiently because of the information inherent in the deformed urban grid. This problem is quite restricted so the fact that it is NP-complete shows how hard this problem really is.

2.4 Research directly related to axial line placement

The work presented in this section covers adjacency detection in configurations of convex polygons and visibility. Two of the algorithms that are developed in these two areas are necessary before axial line placement can take place.

2.4.1 Finding adjacencies in a collection of convex polygons

Two polygons are said to be adjacent if the intersection of an edge from one polygon and an edge from the other polygon is a line segment. This intersection, or shared segment is then called an adjacency. In order to place axial lines upon these shared segments between the convex polygons, one needs to know where the shared segments occur. This can be done trivially in $O((mn)^2)$ where there are n polygons where each polygon has a maximum of m edges. However, Adler *et al.* [2001] gives a $\Theta(mn \log n)$ algorithm, which was reduced to $\Theta(n \log n + n \log m)$ in Konidaris *et al.* [2003]. These last two algorithms use a line sweep strategy.

This section gives an overview of the $\Theta(n \log n + n \log m)$ algorithm. Before continuing, it must be noted that this algorithm assumes that the edges are in counter-clockwise order. Next is a discussion on the algorithm to determine whether one polygon is adjacent to another. Following this is a discussion of the partial ordering used by the algorithm then the main algorithm is explained.

Testing if two convex polygons are adjacent

The algorithm begins by separating each polygon into upper and lower chains as shown in figure 2.7. The chain boundries are indicated by the arrows where the upper chain is in bold and the lower chain is represented by dashed lines.

The points indicated with the arrows are the points with the smallest and largest x-values, which are called the extreme points and can be computed in $\Theta(\log m)$ time, using

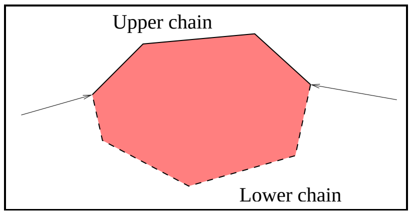


Figure 2.7: The polygon is decomposed into *upper* and *lower* chains for adjacency detection.

a modified binary search [O'Rourke 1995]. Furthermore, the line segment formed by the intersection of the polygon and a vertical line at a particular x value can be obtained in $\Theta(\log m)$, since the extreme points can be determined in $\Theta(\log m)$, and the chains are monotonic in x, allowing a binary search to obtain the edges that cover the relevant x value in $\Theta(\log m)$ time.

The algorithm proceeds by splitting each polygon into their respective upper and lower chains, and checks the first polygon's upper chain against the second polygon's lower chain, and vice versa. Then half of the edges from at least one of the chains are repeatedly eliminated until an adjacency is found or none can exist. Elimination ceases when either of the chains has fewer than four edges, in which case a binary search on gradient is performed for each edge.

The chains are reduced as follows. First, the middle edge of each chain is obtained, the midpoints of these middle edges are computed, and the angles between the line segment joining the two midpoints and their middle edges are obtained.

The algorithm is then split into two cases which are defined by the angles between the line segment joining the two midpoints and their middle edges. Figure 2.8 is an example of the first case because the angles A and B are both less than or equal to π. If either A or B was greater than π then figure 2.8 would be an example of the second case.

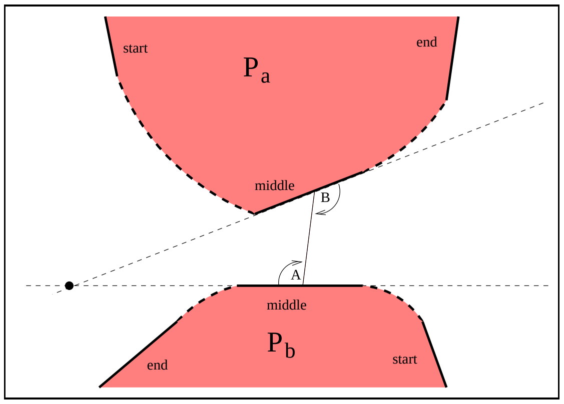


Figure 2.8: Case 1 of checking if the upper chain of polygon P_b shares an edge with the lower chain of polygon P_a. Angles A and B are less than π.

In this case, the middle edges are extended and the intersection is found, as indicated by the broken lines in figure 2.8. If the middle edges are parallel then there can be no such intersection and the polygons are not adjacent. The edges from *start* to *middle* in P_a are not eliminated and neither are the edges from *end* to *middle* in P_b.

The second case occurs when at least one of the two angles is greater than π, and therefore the line connecting the midpoints of the middle edges of the polygon goes through at least one of the polygons. In this case, the edges that lie on the same side of the midpoint as the other polygon's midpoint are kept, and the others removed. Figure 2.9 illustrates this. Here, the edges from *middle* to *end* from P_a are retained as well as all the edges from P_b. Note that the edges beginning at *start* up until, but not including, *middle* from P_a could only be eliminated if B was greater than π.

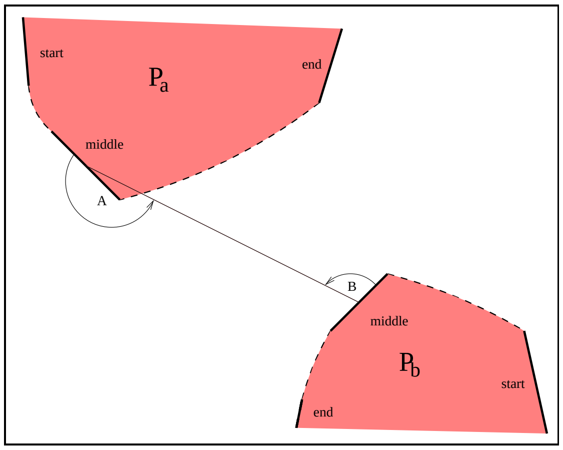


Figure 2.9: Case 2 of checking if the upper chain of polygon P_b shares an edge with the lower chain of polygon P_a. Angle A is greater than π.

The elimination continues until one of the chains has fewer than four edges left. In this case, the longer chain is searched using a standard binary search on edge gradient for each one of the remaining edges in the shorter chain, and if any of them are found to overlap the polygons are adjacent; otherwise, they are not.

A Partial Ordering for Convex Polygons

The line sweep algorithm finds the adjacencies in a configuration based on the intuition that two polygons must be “nearby” if they are to be adjacent. In this section, a partial ordering is presented that determines if two polygons are “nearby” if there can be no polygons between them.

First, the line joining the first and last vertex of polygon P_i's upper chain is labelled as ML_{P_i}, and its y value at a given x coordinate is referred to as $ML_{P_i}(x)$. Figure 2.10 shows an example with ML shown as a dashed line.

The relation $P_i \prec P_j$ is defined below, where the interval I is defined as the intersection of the intervals spanned on the x-axis by ML_{P_i} and ML_{P_j}.

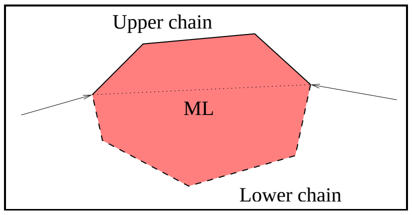


Figure 2.10: A convex polygon with its interior midline (ML).

$P_i \prec P_j$ if

1. $\exists k \in I$ such that $ML_{P_i}(k) < ML_{P_j}(k)$ or
2. $\forall k \in I, ML_{P_i}(k) = ML_{P_j}(k), I$ is not a singleton, and p_i lies below ML_{P_i} or
3. $I = \{k\}, ML_{P_i}(k) = ML_{P_j}(k)$, and ML_{P_j} extends further than ML_{P_i}.

Number 2 can only happen when the opposing chains are of length 1 and the polygons are adjacent along them. The third case is only intended to handle the special case where one polygon's upper chain starts at the same point as another's upper chain ends.

The Algorithm

The algorithm for finding all the adjacencies in a configuration involves sweeping a vertical line from the far left of the plane to the far right, while maintaining a list of the polygons that the line intersects with as it moves through the plane. The list of polygons is kept ordered using the $\prec$ operator.

To do this, a list of the start and end vertices of the upper chains of each polygon is obtained. The list is then sorted on each point's x coordinate, breaking ties on the point's y coordinate, then on polygon number. This list now contains a set of event points, where each event point indicates that the set of polygons intersecting with the sweep line changes – either a polygon is leaving the set (an end point), or a polygon is entering the set (a start point).

During the scan, only those polygons that are at some point next to each other in the ordered list are tested for adjacency.

This section has discussed an algorithm to detect all the adjacencies in a configuration of convex polygons. Next, the area of visibility is discussed, where an algorithm is presented that detects if the adjacencies in a configuration are visible to each other.

2.4.2 Visibility

Visibility mainly has to do with whether two points inside a polygon can "see" each other. In other words, can a line segment be drawn from one point to the other without it intersecting the edges of the polygon? The same analogy holds for visibility between lines. This section briefly discusses the area in general, however, more information can be found in Asano *et al.* [1999] which gives a more complete overview of the area. For some discussion on open problems see O'Rourke [1998] which deals with point and edge visibility, floodlight illumination and visibility graphs.

This section discusses the various types of visibility such as point to point visibility, edge and polygon visibility with various combinations thereof such as point-to-edge visibility. However, the algorithm for edge-to-edge visibility is discussed in detail at the end of the section because it is the most relevant to axial line placement.

Types of Visibility

One of the more natural problems in visibility is point-to-polygon visibility, which is the problem of determining how much of a polygon is visible from a point. This is done by computing the visibility polygon $V(p)$ of a point p. Formally $V(p) = \{q \in P | p \text{ sees } q\}$ [Asano *et al.* 1999]. Figure 2.11 shows a visibility polygon of a point. El Gindy and Avis [1981] and Lee [1983] developed linear time algorithms to calculate $V(p)$. Both were later shown to fail in certain cases by Joe and Simpson [1987] with corrections to the algorithm given in Joe [1990].

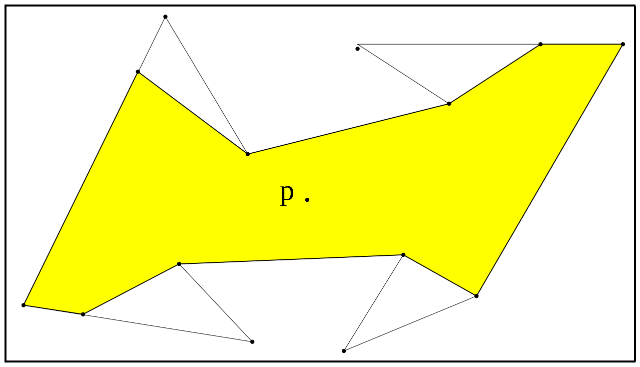


Figure 2.11: The shaded polygon is the visibility polygon of point p.

Avis and Toussaint [1981] introduced the notions of weak and complete visibility. A point is *weakly visible* from a polygon P if it is visible to at least one point in P, and it is *completely visible* from P if it is visible to all points in P. They also introduced notions of weak, strong and complete visibility of a polygon to an edge. An $O(n)$ algorithm was presented that determines whether a given polygon P is completely, weakly or strongly visible from an edge uv. Note that this is a decision algorithm and does not determine the polygons themselves. Avis and Toussaint [1981] derived an algorithm to determine the complete visibility polygon of an edge. A linear algorithm to compute the weak visibility polygon came later from Guibas *et al.* [1986].

Another type of visibility which has relevance to axial line placement is that of *link visibility* [Asano *et al.* 1999]. The *link distance* between two points p and q in a polygon P is the minimum number of line segments in a polygonal path from p to q that stays within P.

The *link diameter* of a polygon is the maximum link distance between any pair of points out of all the pairs of points in the polygon. So the link diameter of a polygon is 1 if and only if it is convex. The axial lines can be seen as the links between points in the convex polygons. This makes it easy to at least approximate the link distance between points in the convex map. ALP is used to do something similar, but instead of finding the link distance between two points in an area, ALP can be used as a step in finding the link distance between two sub-areas in a given area. In this case the areas are convex.

The most relevant type of visibility to this research is edge-to-edge visibility. There are many types of edge-to-edge visibility. Avis *et al.* [1981] gives the following classifications:

- Edge uv is said to be *completely* visible from edge xy if for all points z on edge xy and all points w on edge uv, w and z are visible.

- Edge uv is said to be *strongly* visible from edge xy if there exists a point z on edge xy such that for all points w on edge uv,w and z are visible.

- Edge uv is said to be *weakly* visible from edge xy if for each point w on edge uv there exists a point z on edge xy such that w and z are visible.

- Edge uv is said to be *partially* visible from edge xy if there exists a point w on edge uv and a point z on edge xy such that w and z are visible.

The relevant type of visibility here is *partial* visibility, because a single line can only cross all the adjacencies in a sequence of convex polygons if there exists a point on each adjacency in all pairs of adjacencies in the sequence that are visible. Avis *et al.* [1981] gives an $O(n)$ algorithm to compute these four edge-to-edge visibilities and the parts that compute partial edge visibility are presented here, but first a method is presented that finds the polygon that gets inputted into the visibility algorithm. This polygon is called the *adjacency polygon*.

Determining the adjacency polygon

The following method for finding an *adjacency polygon* is detailed in du Plessis and Sanders [2000], though only orthogonal rectangles were considered in this paper. This is expanded in this document to encompass convex polygons.

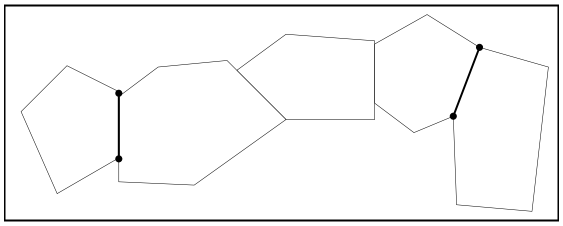


Figure 2.12: A chain of convex polygons polygons. The thick line segments are the line segments we wish to place an axial line upon

A prerequisite for the algorithm is that the polygons that the axial line should pass through are known. Figure 2.12 shows the initial configuration. The two polygons on either end of the sequence or chain are removed and the adjacency polygon is formed from the union of the other polygons in the chain (see figure 2.13). The *reduced* adjacency polygon can be formed by joining the end points of the adjacencies to create a chain of quadrilaterals (see figure 2.14). The reduced adjacency polygon is not completely necessary as shown in the following paragraph. It will however make the line visibility algorithm more efficient. Note that the step to find the *un-reduced* polygon can be skipped and the vertices of the adjacencies joined in order to find the reduced adjacency polygon. Obviously it has to be ensured that the line segments that join the adjacencies do not cross.

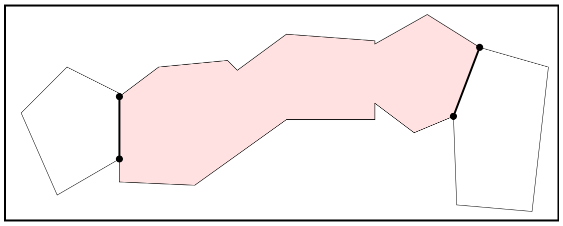


Figure 2.13: The adjacency polygon has been found and is indicated by the shaded region.

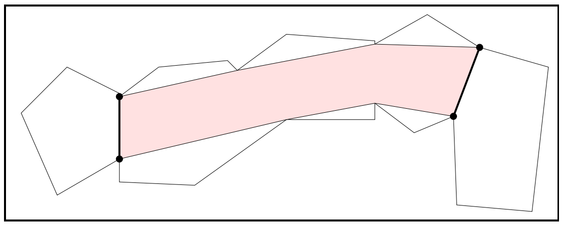


Figure 2.14: The reduced adjacency polygon is indicated by the shaded region. It is found by reducing the adjacncy polygon shown in figure 2.13.

Partial edge visibility through a polygon

Once the adjacency or reduced adjacency polygon has been determined the lines must be tested for *partial visibility*. Edge uv is said to be partially visible from edge xy if there exists a point w on edge uv and a point z on edge xy such that w and z are visible. In figure 2.15 the edges uv and xy in polygon A are partially visible because there exist two points w and z that can see each other, illustrated by the line segment wz. Clearly edges ab and mn in polygon B are not partially visible because there exists no line segment that joins the two lines such that the segment lies entirely in B. The fact that edges uv and xy are visible indicates that an axial line can be placed upon these two edges.

Determining if two edges in a polygon are partially visible is done using the linear algorithm derived in Avis *et al.* [1981]. This algorithm determines if two edges are completely,

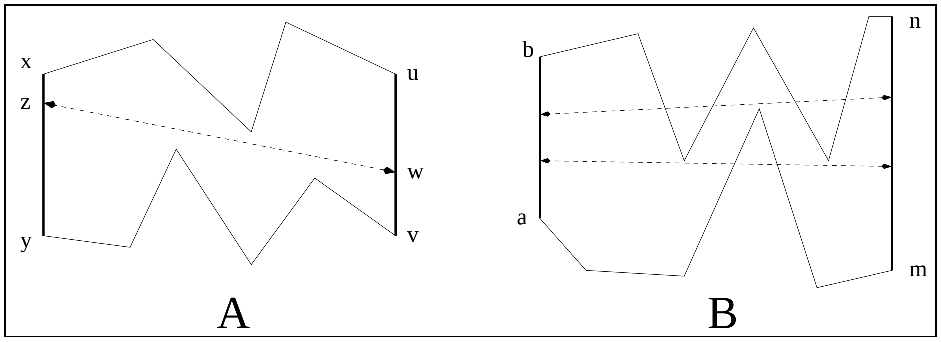


Figure 2.15: Illustration of partial edge visibility. uv is partially visible to xy but ab is not partially visible to mn.

strongly, weakly or partially visible, though only partial visibility is required here and so will be the only type discussed. There are many different special cases that need to be addressed that depend upon how the lines face each other. The simplest case is where the two lines are totally facing each other so will be the case considered in the explanation of the algorithm. It is recommended that the reader refers to the original text for a complete explanation. The edges in polygons A and B in figure 2.15 are totally facing each other.

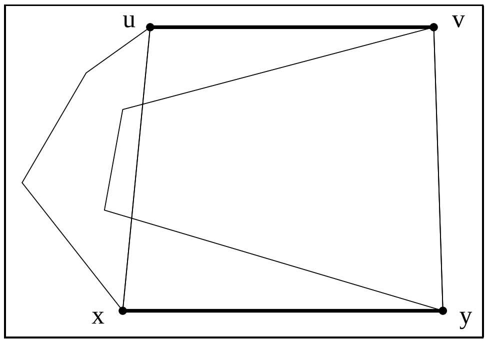


Figure 2.16: An example of $C(v, y)$ cutting through $Q(u, v, y, x)$ indicating that uv is not partially visible to xy.

Given is polygon P and edges uv and xy. The algorithm takes the following steps:

1. Create quadrilateral $Q(u, v, y, x)$.

2. Create chains $C(u, x)$ and $C(y, v)$ where $C(u, x)$ is the chain of vertices in the polygon from vertex u to vertex x (analogous for $C(y, v)$).

3. Determine if $C(u, x)$ or $C(y, v)$ *cuts* through $Q(u, v, y, x)$. If it does then no visibility is possible and the algorithm terminates. Figure 2.16 shows an example of $C(y, v)$ cutting through $Q(u, v, y, x)$.

4. Create $RC(u, x)$ and $RC(y, v)$ where $RC(u, x)$ is a reduced chain of vertices. This

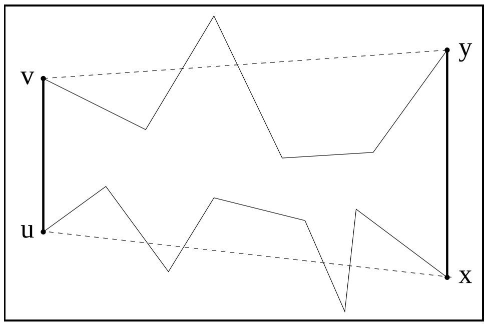


Figure 2.17: Initial polygon with $Q(u, v, y, x)$ in dashed lines

chain contains vertices in $C(u, x)$ that lie outside $Q(u, v, y, x)$ and also contains points where $C(u, x)$ intersects with the line segment ux.

5. Calculate $ICH(u, x)$ and $ICH(y, v)$ of $RC(u, x)$ and $RC(y, v)$ respectively. Where $ICH(u, x)$ is the *inner convex hull* of $RC(y, v)$. Figures 2.17 to 2.19 show the steps in this process beginning with the input polygon and ending with the inner convex hulls.

6. Form polygon H from $ICH(u, x), xy, ICH(y, v), vu$.

7. If H is a simple polygon (ie. doesn't intersect itself) then uv is partially visible from xy and vice versa.

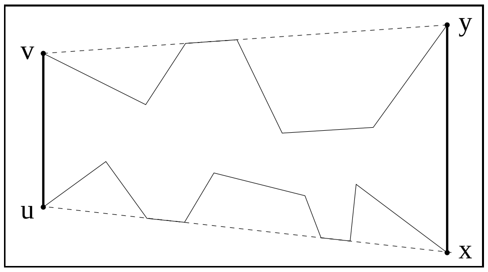


Figure 2.18: Illustration of reduced chains. The polygon from figure 2.17 has been "cut off" by $Q(u, v, y, x)$ to form the reduced chains.

This section discusses an algorithm to determine if two edges are partially visible through a chain of convex polygons. However, this only indicates that an axial line can be placed upon these edges and does not say where the axial line should be placed. Fortuitously, the actual placement of a line is only important for a graphical representation. At this point it is sufficient to know that an axial line can be placed.

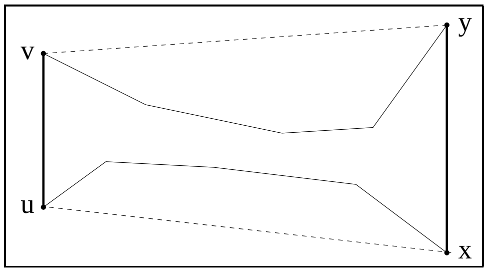


Figure 2.19: Illustration of inner convex hulls. The inner convex hulls are formed by finding the convex hulls of the reduced chains from figure 2.18.

2.5 Maximum cardinality matching

This section describes an algorithm to solve *maximum cardinality matching*, which is a well known problem in graph theory. Stars of convex polygons are one of the special cases considered in this document and maximum cardinality matching is part of the algorithm that solves the problem of axail line placement in this configuration.

A *matching* of an undirected graph $G = (V, E)$ is a subset of the edges $M \subseteq E$ such that no two edges in M touch a common vertex where $n = |V|$ and $m = |E|$. A *maximum cardinality* matching is a matching with the maximum number of edges. In this research, maximum cardinality matching is used for axial line placement in a special type of configuration of convex polygons called a *star* of convex polygons discussed in chapter 4.

The first polynomial time algorithm for this problem appears in Edmonds [1965], which is described in this section. At each stage of the algorithm there is a matching M. Initially, M is empty. A vertex i is *matched* if there is an edge (i, j) in M and *single* otherwise. An edge is *matched* if it is in M and *unmatched* if it is not in M.

Simply put, the algorithm works by repeatedly searching for a type of path called an *augmenting path* and terminates when none are found. During the search for an augmenting path a cycle can be found that Edmonds called a *blossom*. The concepts of *augmenting paths* and *blossoms* are explained in the following section. Then the algorithm will be explained.

Augmenting paths and blossoms

An *alternating path* (with respect to M) is a simple path, such that every other edge on it is matched. An *augmenting path* (with respect to M) is an alternating path between two *single* vertices. In figure 2.20, the paths *s,a,r,b,c,d,i* and *s,a,r,g,f,e,j* are augmenting paths. The path *s,a,r,g,f* is an alternating path but is not an augmenting path because *f* is one end of the path and is matched. Any contiguous part of these paths are alternating paths.

It is easy to see that the size of a matching can be increased by one if the matched (unmatched) edges in an augmenting path are changed to unmatched (matched) edges. This operation is called *augmenting the matching M*. In fact, Berge [1957] and Norman and

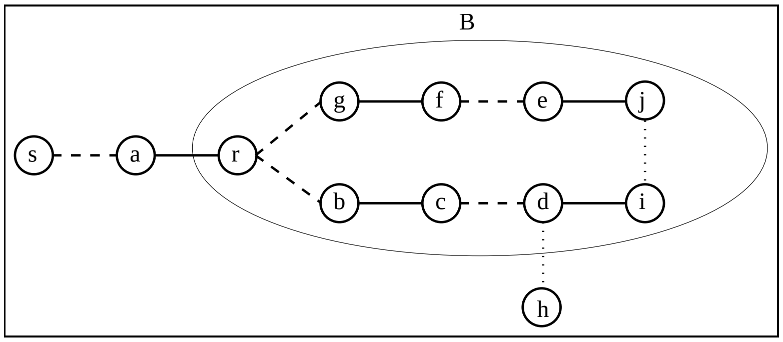


Figure 2.20: The solid lines represent the matched edges. The dashed lines represent edges that have been considered by the search but not matched. Edges (i, j) and (d, h) have not yet been considered by the search. B is a blossom formed when edge (i, j) is considered.

Rabin [1959] showed that a matching M has maximum cardinality if and only if there is no augmenting path with respect to M. This result gives the following algorithm for bipartite graph. At each stage a search is performed for an augmenting paths which is augmented if found and terminates if no augmenting path is found. However, for the non-bipartite case, the algorithm is complicated by the existence of odd-length cycles. Edmonds introduced the concept of blossoms to resolve this.

Consider figure 2.20. If edge (i, j) is considered during the search for an augmenting path, then the cycle created is called a *blossom* and r is called its base because the paths that end in i and j begin at r.

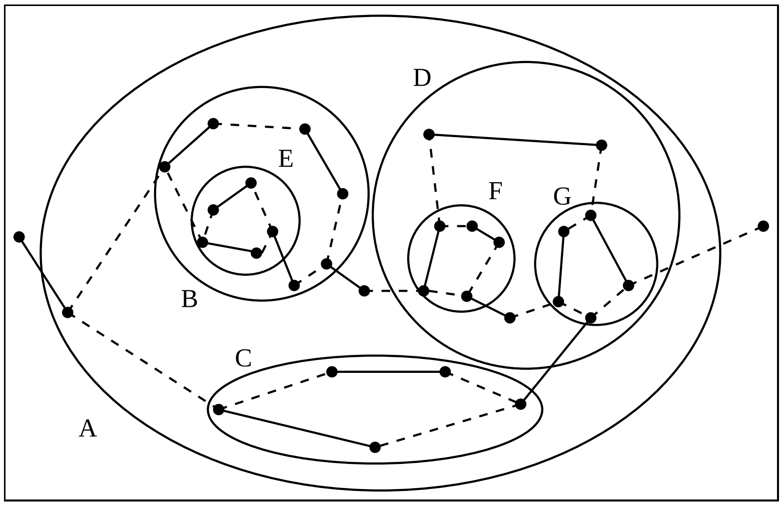


Figure 2.21: A graph with blossoms within blossoms. The circles show the blossoms or super vertices. The set of solid edges is the maximum cardinality matching for this graph.

The blossom is now considered a super vertex, so the path from s to h is *s,a,B,h*. A blossom is a recursive structure because it can contain other blossoms and its structure is stored in a tree called the *structure tree*. Figure 2.21 demonstrates how blossoms can occur within other blossoms. A contains B,C and D; B contains E; and D contains F and G. For convenience, single vertices are considered degenerate blossoms.

The blossom now replaces all the vertices that are in it creating a new graph. Edmonds showed that if there is no augmenting path in the graph with the blossom then the matching is a maximum cardinality matching.

How the algorithm works

This explanation is taken from Galil [1986]. At each stage of the algorithm a search for an augmenting path is conducted. If any augmenting paths exist, the search finds one (and only one) and the matching is augmented. The algorithm terminates when no augmenting paths can be found. The algorithm labels vertices S or T where each label contains the vertex from which the label arrived.

The search works as follows. At each stage all the labels are cleared then all the single or unmatched vertices are labelled S and the rest are unlabelled. Initially all the vertices in the graph are labelled S. At this stage the origin of each S vertex is the vertex itself. Now the S vertices are inserted into a queue Q and are scanned in the order that they occur in the queue. Scanning a vertex means considering in turn all its edges except the matched edge (there will be at most one).

The following rules are applied to the edges.

1. If (i, j) is not matched and i is labelled S and j is unlabelled, then label j by T and use rule 2 to label the spouse of j with S.

2. If (i, j) is matched and j is labelled T and i is single, then label i with S.

The application of these rules depends upon the following cases.

1. j is single

2. j is an S-vertex

The case where j is a T vertex is discarded.

In case 1 rule 1 is applied. In case 2 the following is done: Backtrack from i and j, using the labels, to the single persons s_i and s_j from which i and j got their S labels. If $s_i \neq s_j$, an augmenting path from s_i to s_j is found and the matching is augmented. The first time that the search is done all the vertices are labelled S and point to themselves, so the first edge that is considered is matched because both vertices are unmatched S vertices and both point to themselves. However the case where $s_i = s_j$ is more difficult and requires the use of blossoms.

If $s_i = s_j = s$, let r be the first common vertex on the paths from i and j to s. It can be seen that r is an S vertex because the labels only point to other labelled vertices and all T vertices get their labels from S vertices. Moreover, the part of the two paths from i and j to r are disjoint, and that the parts from r to s are identical. There is now an alternating path from r to itself through (i, j). This cycle is now the blossom and r is its base. The blossom itself is now inserted into Q and labelled as an S vertex. Its sub blossoms are removed from Q.

If the search succeeds, an augmenting path has been found. The next stage begins with all labels and blossoms cleared.

Implementations and complexity

The algorithm presented in Edmonds [1965] is $O(n^4)$ but was improved in Gabow [1976] to $O(n^3)$, where $m = |E|$ and $n = |V|$. The implementation used for the empirical testing presented in the document is the $O(n^3)$ algorithm and was downloaded from `http://elib.zib.de/pub/Packages/mathprog/matching/weighted/`. Further improvements were done using Union and Find [Gabow and Tarjan 1983] which improved the algorithm to $O(nm)$.

The algorithm with the best running time is the $O(m\sqrt{n})$ algorithm of Micali and Vazirani [1980]. However, the algorithm is relatively difficult to understand and it took nearly 10 years after the initial publication to prove its correctness [Vazirani 1989].

2.6 Research that is similar to axial line placement

There are various other areas of research that seem similar to ALP. Such areas are art gallery guarding and stabbing. This section discusses these problems and how they differ from axial line placement.

As mentioned in section 2.3, axial line placement was once called *ray guarding*. This name originated from the *Art Gallery* problem. This is the problem of placing guards in an art gallery so that every point in the gallery is visible to at least one guard. The gallery is represented by a polygon and the guards are points in the polygon. There are different types of possible *guards* such as vertex guards. This is where the guards can only be placed upon the vertices of the polygon. Other types of guards are edge guards and diagonal guards. In this section is a discussion on *point* guards. Urrutia [1999] and O'Rourke [1987] give reviews of art gallery problems and will point the reader to more literature in the area. Shermer [1992] discusses some more results in art gallery problems.

Figure 2.22 shows an optimal placement of guards in a polygon. Optimal, in this cases, means the least number of guards placed. The black area represents a hole in the polygon. The hole is representative of a wall in an art gallery. If each *guard* or point placed in the polygon generates its own visibility polygon (see section 2.4.2) and the union of these polygons is the original polygon (the art gallery) then the placement is valid. The shaded area in figure 2.22 is the visibility polygon of point A. If not then another guard needs to be placed or the current points should be moved around. This shows that art gallery problems are reliant upon the work done on visibility. In fact all art gallery problems are visibility problems.

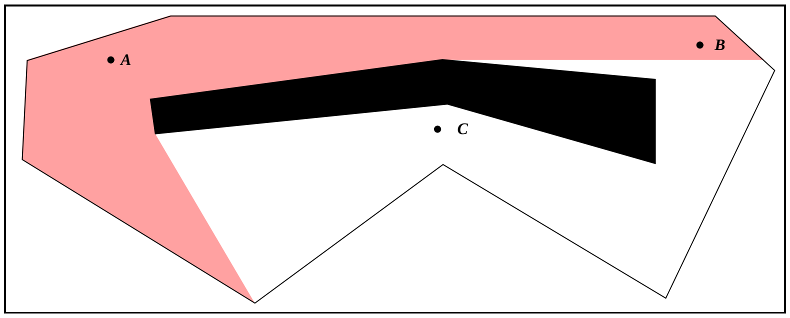


Figure 2.22: Optimal placement of guards (points) in an art gallery (polygon).

Axial line placement is also heavily reliant on visibility and so has this in common with art gallery problems but they are still separate. The point of placing the guards in an art gallery is so that they can collectively see the whole area. Axial line placement on the same configuration would tell us how well these guards can see *each other*. The example to follow will demonstrate this.

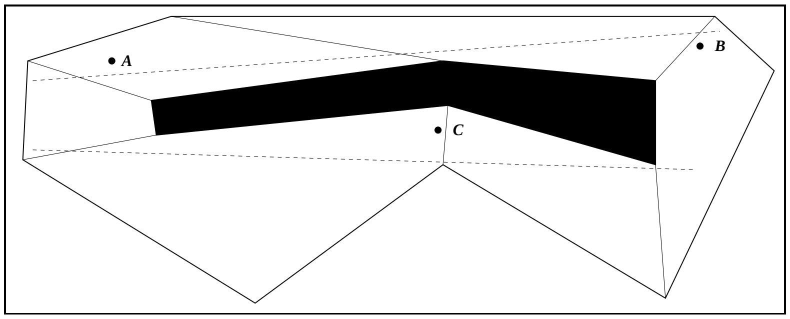


Figure 2.23: Optimal axial line placement upon a partitioned polygon

Figure 2.23 shows the polygon from the example above partitioned into convex polygons with optimal axial line placement. Points A and B only have one line joining their areas so they are the most accessible to each other. Point C requires that two lines are followed to join its area to A or B so it is the most remote point out of the three. This shows that axial line placement and the art gallery problem are inherently different.

Another problem more closely related to ALP is that of *stabbing* [Avis and Wenger 1987; Edelsbrunner *et al.* 1982; Pellegrini and Shor 1992; Pellegrini 1993]. This is the problem of finding a *stabber*. This is a line that intersects every member of a given polyhedral set. Such a line may not exist. Clearly no *stabber* exists for the polygons in figure 2.24 and so the problem cannot be solved. This is one aspect of stabbing that differs from axial line placement. Stabbing doesn't look for the smallest number of stabbers to cover the whole set it just looks for one line. In axial line placement the line segments have to stay inside the polygons and the polygons cannot overlap. These restrictions do not apply to stabbing.

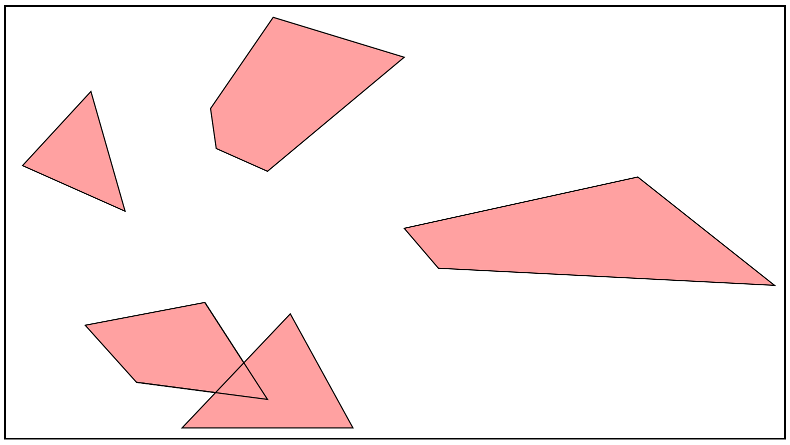


Figure 2.24: Illustration of stabbing in 2-space

One aspect of stabbing that can be used in axial line placement is the actual drawing of the axial lines. Once the adjacencies that will be crossed by line segments have been determined, stabbing can be applied to the adjacencies to determine where the line segments should go. This is but a small part of axial line placement, in fact, it need not be done at all if the application does not require that the lines be displayed graphically.

2.7 Conclusion

Axial line placement is a relatively new area of research but this chapter has shown it to tie in with other areas such as visibility. The work on visibility can completely abstract the polygons themselves from the problem and the problem can be dealt with at a higher level. This chapter has shown that there are no heuristics for the general case of axial line placement other than the heuristic presented in this dissertation.

Chapter 3

Chains of convex polygons

3.1 Introduction

The aim of this research was to produce the heuristic for axial line placement in convex polygons presented in chapter 6. This chapter discusses chains of convex polygons, which is the first of three special configurations of convex polygons that will be used for preprocessing. Chains of convex polygons are joined together to form stars of convex polygons, which is the subject of chapter 4. Chapter 5 is the final chapter discussing special cases and shows how the network of stars configuration is created by combining stars of convex polygons.

The network of stars configuration is created to make the process of detecting stars in a general configuration of convex polygons more effective and forms a method for preprocessing. However, the detection of networks of stars forms only one of three methods considered for preprocessing. Chapter 6 gives the details of the other two.

Chains of convex polygons is the most basic special case considered by any research in axial line placement. Algorithms have been developed to solve axial line placement in rectangles ([Phillips 2001; Sanders *et al.* 2000b]) but previously there was no algorithm to place lines in chains of *convex* polygons. This chapter presents an $O(n^2)$ algorithm to do this. The algorithm begins by finding all the axial lines that cross the adjacencies in a chain then transforming to a variation of Interval Point Cover (***IPC***). The intervals that are returned represent the axial lines in the solution.

This chapter begins with a formal definition of a chain of convex polygons, then Interval Point Cover is introduced. Following that, the transformation process is given and ***IPC*** is solved. The transformation process can be skipped so a more efficient algorithm is given that solves axial line placement in convex polygons but only mimics the algorithm for ***IPC***.

3.2 A formal definition

A chain of convex polygons is a configuration of convex polygons where each polygon is adjacent to exactly two other polygons with the exception of two polygons which are

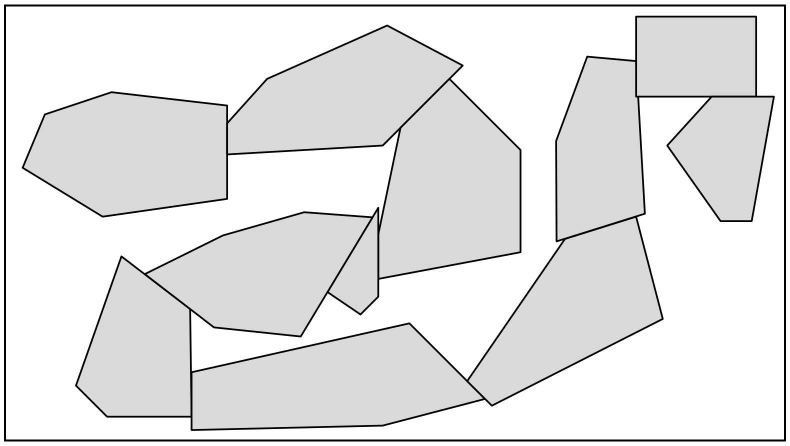


Figure 3.1: A chain of convex polygons

adjacent to only one other polygon each. The polygons that are adjacent to only one other polygon are the ends of the chain. An example of a chain of convex polygons is shown in figure 3.1. If the chain is denoted by C then the polygons in the chain are $c_1, c_2, ..., c_n$ where n is the number of polygons in the chain. These are listed in chain order, meaning c_1 is at one end of the chain, c_2 is adjacent to c_1, c_3 is adjacent to c_2 and so on. This implies that c_n is at the other end of the chain. Note that $c_a|c_b$ denotes the adjacency between polygons c_a and c_b. $|C| = n$.

3.3 Interval Point Cover (*IPC*)

Mirzaian [2002] defines interval point cover (***IPC***) as follows. Given a set $P = \{p_i | i = 1..n\}$ of n points and a set $I = \{[s_i, f_i] | i = 1..m \text{ and } s_i < f_i\}$ of m intervals, all on the real line. The interval cover problem is to find a minimum cardinality subset C of intervals from I that collectively cover all points in P. If there is a point that is not covered by any interval in I then that point should be returned as opposed to C because there is no solution.

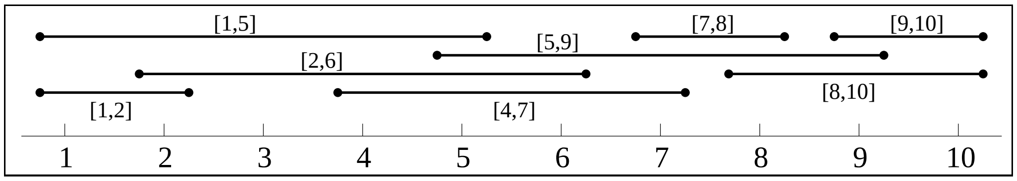


Figure 3.2: An example of a set of intervals over the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

Figure 3.2 shows an example problem with $P = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and $I = \{[1, 2], [1, 5], [2, 6], [5, 9], [7, 8], [4, 7], [8, 10], [9, 10]\}$. Figure 3.3 shows a solution set, $C = \{[1, 5], [5, 9], [8, 10]\}$. Note that interval $[9, 10]$ could have been chosen instead of $[8, 10]$, but the problem of axial line placement requires that the lines are as long as possible so the longer interval is chosen.

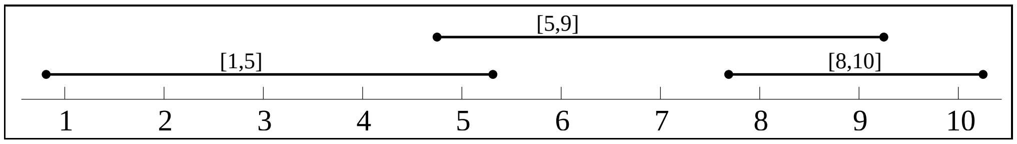


Figure 3.3: A solution to ***IPC***

To transform ALP in chains of convex polygons to ***IPC***, each adjacency in a chain of convex polygons is considered to be a point in P and the axial lines are the intervals. This means that only the version of the problem where a solution is guaranteed is considered since all of the adjacencies are crossed by some axial line.

3.4 Transformation to *IPC*

In this section, the problem at hand is transformed to ***IPC***. The reason for doing this is that it gives an abstract view of the problem and is easy to come up with examples that show specific characteristics about the problem. The axial lines and the adjacencies between polygons will be transformed into the intervals and points respectively. The transformation begins by first generating all possible axial lines in a given chain of convex polygons then transforming these into intervals. The transformation is presented as an algorithm and the output of the algorithm is shown to produce the required input for ***IPC***.

In order for algorithm 1 to transform the problem correctly it is necessary to show that all elements of the problem are transformed as well. Showing this requires that the algorithm transforms *all* possible axial lines into intervals. The following lemma says that all axial lines in chains of convex polygons must cross consecutive adjacencies. This means that an axial line that crosses some adjacencies in a chain, C, can be represented by the interval $[a, b]$ if $c_a|c_{a+1}$ and $c_b|c_{b+1}$ are the adjacencies where the axial begins and ends respectively.

Lemma 1 *Suppose that l is an axial line in a chain C. If $i \leq k \leq j$ and l crosses $c_i|c_{i+1}$ and $c_j|c_{j+1}$ then l must cross $c_k|c_{k+1}$.*

Proof – by contradiction

Consider figure 3.4. Suppose that a line, l, crosses $c_i|c_{i+1}$ and $c_j|c_{j+1}$ but doesn't cross $c_k|c_{k+1}$ where $i \leq k \leq j$. l must stay within the chain so in order for it to cross both $c_i|c_{i+1}$ and $c_j|c_{j+1}$ it must pass in between these two. However, $c_k|c_{k+1}$ divides the chain into two pieces where $c_i|c_{i+1}$ is in one piece and $c_j|c_{j+1}$ is in the other piece so l must cross $c_k|c_{k+1}$, resulting in a contradiction of the fact that l does not cross $c_k|c_{k+1}$. □

Theorem 1 *If C is a chain of convex polygons then there are at most $\frac{q(q+1)}{2}$ axial lines in C where $q = |C| - 1$.*

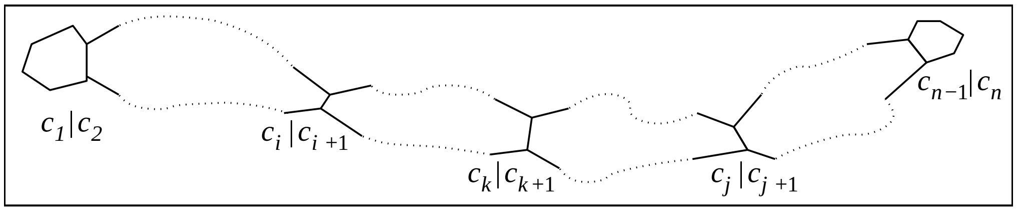


Figure 3.4: A representation of a chain with adjacencies $c_i|c_{i+1}$, $c_j|c_{j+1}$ and $c_k|c_{k+1}$ indicated.

Proof – by induction

Assume that $q = 1$. Since there is only one adjacency there can only be one axial line to cross it and $\frac{1(1+1)}{2} = 1$ so the theorem has been proven for $q = 1$.

Induction Hypothesis: assume that the theorem is true for $q = k$ where $k \geq 1$. It is now required to show that the theorem is true for $q = k + 1$.

Let C be a chain of convex polygons with $|C| = k+2$. If c_1 is removed from C then the chain C' is created with $|C'| = k + 1$. By the induction hypothesis there are at most $\frac{k(k+1)}{2}$ axial lines that cross the adjacencies in C', call this number a. If the number of axial lines needed to cross the adjacencies in C is m and the largest number of lines that could possibly cross $c_1|c_2$ is b then $m \leq a + b$ since the inclusion of another adjacency cannot increase the number of axial lines required to cross adjacencies that are not the added adjacency.

By the induction hypothesis a is known. In order to find the bound on m only b needs to be found. In order for the number of lines that cross $c_1|c_2$ to be maximized it is required that all of the adjacencies in C' be visible to $c_1|c_2$. To see this consider the case where one adjacency is not partially visible to $c_1|c_2$. This would mean that there is no axial line that crosses both adjacencies resulting in the number of axial lines crossing $c_1|c_2$ being reduced by one so the number would not be maximal and less than b. If $c_1|c_2$ is visible to all adjacencies in the chain then there is at least one axial line per adjacency that extends from $c_1|c_2$ to that adjacency and stops there. There are k adjacencies in the rest of the chain and one axial line crosses $c_1|c_2$ itself therefore there are at least $k + 1$ axial lines to cross $c_1|c_2$ provided that $c_1|c_2$ is visible to all other adjacencies in C'.

Now, by lemma 1, if an axial line, l, crosses $c_1|c_2$ and $c_i|c_{i+1}$ all the adjacencies between these adjacencies must be crossed by this line. Now if l does not cross adjacencies $c_{i+1}|c_{i+2}$ to $c_q|c_{q+1}$ then l is the only axial line that crosses $c_1|c_2$ and $c_i|c_{i+1}$ but does not cross adjacencies $c_{i+1}|c_{i+2}$ to $c_q|c_{q+1}$. This means that there can be no more than $k + 1$ axial lines to cross $c_1|c_2$ therefore $b = k + 1$.

Substituting in for a and b:

$$m \leq \frac{k(k+1)}{2} + k + 1$$
$$m \leq \frac{k(k+1)+2(k+1)}{2}$$
$$m \leq \frac{(k+2)(k+1)}{2}$$

Similarly shown if c_{k+2} is removed. So there are at most $\frac{(k+2)(k+1)}{2}$ axial lines that cross the adjacencies in C. By induction the theorem is true for any q. $\square$

This proof leads on to algorithm 1 which transforms the problem of axial line placement into that of ***IPC***. The while loop finds where all axial lines that begin at $c_j|c_{j+1}$ would be and adds the corresponding intervals to the final set of intervals directly. The **for** loop begins this process at the first adjacency then the second then the third and so on. This means that it is not necessary to find the axial lines that don't start at $c_j|c_{j+1}$ in the while loop because those would have already been found previously.

Algorithm 1 Transform axial line placement in chains of convex polygons into ***IPC***.

```
input: A chain of convex polygons $C$.
output: An set of intervals $I$ over $\mathbb{N}$.
 1: $I \leftarrow \emptyset$
 2: $n \leftarrow |C| - 1$
 3: for $i = 1$ to $n$ do
 4:     $j \leftarrow i + 1$
 5:     while $j \neq n + 1$ AND visible$(c_i|c_{i+1}, c_j|c_{j+1})$ do
 6:         $I \leftarrow I \cup \{[i, j]\}$
 7:         $j \leftarrow j + 1$
 8:     end while
 9: end for
10: return $I$
visible$(a, b)$ returns TRUE if $a$ is partially visible to $b$ and FALSE otherwise.
```

So algorithm 1 has found a set of intervals $I = \{[i,j] | i,j \in \mathbb{N}, 1 \leq i \leq |C| - 1, 1 \leq j \leq |C| - 1\}$ where C is a chain of convex polygons and the beginning and the end of each interval is the index of an adjacency in C where these two adjacencies are partially visible to each other. The set of points can be formed by considering all the indices of the adjacencies in C. This set would just consist of the numbers from 1 to $|C| - 1$. Call this set of points P. Now there is a set of intervals I and a set of points P which satisfies the input for ***IPC***.

The problem is to find the minimum number of axial lines to cross the adjacencies in C. These axial lines must be as long as possible but this will be discussed later. The intervals in I correspond to all possible axial lines crossing the adjacencies in C and the points in P correspond to all adjacencies in C so the equivalent problem is to find minimum cardinality subset D of intervals I that collectively cover all points in P. Now since I corresponds to all axial lines in C all the adjacencies are crossed and so all points in P are covered by some interval. This satisfies the output condition for ***IPC***. The input and output conditions have been satisfied so algorithm 1 transforms axial line placement in chains of convex polygons to ***IPC***.

3.5 Solving *IPC*

The algorithm presented here solves ***IPC*** with the added condition of ALP that the lines chosen should be as long as possible. Algorithm 2 solves ***IPC*** and the proof of correctness follows. Algorithm 2 should not be used in practice and is only included because the theoretical work that follows. A more efficient algorithm is given later on in the chapter

Algorithm 2 Solving ***IPC*** with modifications

```
   input: A set of intervals $I = \{[i,j] \mid i,j \in \mathbb{N},\ 1 \leq i \leq n,\ 1 \leq j \leq n\ i \leq j\}$ where $n = |C| - 1$.
   output: A minimum cardinality subset $D$ of $I$ such that the intervals in $D$ collectively cover all points in $P = \{i|i = 1..n\}$.
1: $D \leftarrow \emptyset$
2: $i \leftarrow 1$
3: while $i \neq n$ do
4:     $T \leftarrow$ set of intervals from $I$ where each interval is of the form $[x,y]$ where $y$ is as large as possible with $x \leq i \leq y$.
5:     $\sigma \leftarrow$ the interval from $T$ that has the lowest left index
6:     $D \leftarrow A \cup \{\sigma\}$
7:     $i \leftarrow$ (right index of $\sigma$)+1
8: end while
9: return $D$
```

Correctness Proof – by induction

The proof that algorithm 2 solves ***IPC*** is done inductively and is derived from a proof found in Mirzaian [2002] which is only an outline of a proof. The proof given here fills in the gaps. This proof proves that the solution, D , found by algorithm 2 solves ***IPC*** by showing that each interval in any solution can be replaced by the intervals in D but the new set still solves ***IPC***.

First consider the case $i = 1$. Clearly, all intervals that include 1 are in the set $R = \{[1,a]|1 < a \leq n\,[1,a] \in I\}$, so an interval must be picked from R. The interval picked by algorithm 2 is the interval that includes 1 and has the largest right index, as specified at line 4. Let this interval be $[1,k]$. Note that there is only one such interval because 1 is the smallest vertex so no interval can extend to the left of it.

So algorithm 2 picks $[1,k]$ from R and places it in D. Now assume that there is another set E of intervals that is a solution for ***IPC*** and it contains $[1,j] \in R$ where $j < k$ resulting in the situation shown in figure 3.5. Now consider $E' = (E - [1,j]) \cup [1,k]$. $[1,k]$ covers all the points that $[1,j]$ covers because $j < k$, so E' is also a solution for ***IPC***. Therefore, $[1,k]$ can be substituted into any solution for ***IPC***.

This shows that the interval chosen to cover the first point in P can replace an interval in any solution and the newly formed set of intervals would still solve ***IPC***. Consider the case where $i = a$ where $1 < a < n$, this means that all the points before a have been covered by some intervals D chosen by algorithm 2. The induction hypothesis is that a new solution to

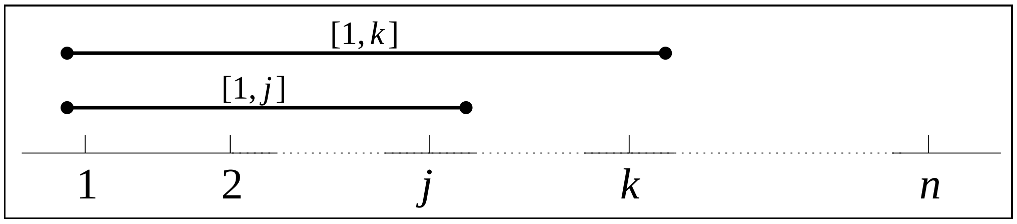


Figure 3.5: The base case for the proof of correctness of algorithm 2. $[1, k]$ is the interval chosen by algorithm 2 and $[1, j]$ represents any other interval that can be chosen to include 1.

IPC can be formed by replacing some intervals in any solution with all the intervals in D. What needs to be shown is that the intervals D, with the addition of the next interval chosen by algorithm 2 can replace adjacencies in any solution of ***IPC*** while still being a solution to ***IPC***. The proof is similar to that of the base case.

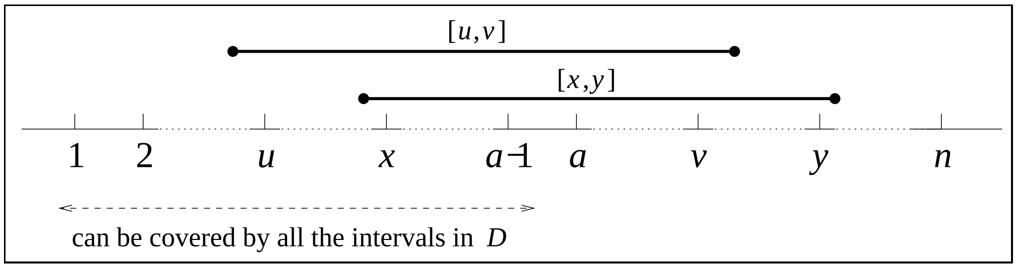


Figure 3.6: The inductive case for the proof of correctness of algorithm 2. $[x, y]$ is the interval chosen by algorithm 2 and $[u, v]$ represents any other interval that can be chosen to include a.

The intervals in $Q = \{[x, y] | x, y \in P, 1 < x \leq n, 1 < y \leq n, x < a < y\}$ are all the intervals that can cover a. Algorithm 2 picks an interval $[x, y] \in Q$ such that y is as large as possible and the difference between the chosen y and x is as large as possible. Figure 3.6 shows the position of $[x, y]$ where no interval from Q can cover points that are greater than y and no interval that covers y cannot cover points that are less than x. Note that all the points from 1 to y can be covered by all the intervals in $D \cup \{[x, y]\}$.

Line 4 of algorithm 2 chooses the set of intervals with the largest y and line 5 chooses the interval with the smallest x from that set. The fact that the difference is the largest possible means that $[x, y]$ is as large as it can possibly be, satisfying the condition that the intervals should be as large as possible.

Now, suppose that there is a set of intervals E that is a solution to ***IPC*** and contains an interval $[u, v] \in Q$ that is not $[x, y]$ as shown in figure 3.6. However, point u in 3.6 is less than x but in general it may be equal to or greater than x. By the induction hypothesis, some intervals in E can be replaced by all the intervals in D to form a new solution to ***IPC*** E'. The intervals in D were chosen by algorithm 2 so these intervals cover all points before

a. Now consider $E'' = (E' - [u,v]) \cup [x,y]$. $[x,y]$ covers all points after and including a that are covered by $[u,v]$. Additionally, all points before a are covered by intervals in D. Therefore, $D \cup [x,y]$ covers all points $[u,v]$ covers so E'' is a solution for ***IPC***. Therefore, the set formed by replacing the intervals in any solution for ***IPC*** by all the intervals in $D \cup [x,y]$ is itself, a solution for ***IPC***.

If $y = n$ then $[x,y]$ is added to D and the algorithm terminates. Therefore, if E is some solution to ***IPC***, then *all* the intervals in E can be replaced by the intervals in D forming E' and E' is a solution to ***IPC***. But D covers all points in P so $E' = D$ and D is the set of intervals found by algorithm 2. Therefore, algorithm 2 solves ***IPC***. $\square$

3.6 Improved chain algorithm

The process of transforming the problem into ***IPC*** was done to aid the theoretical work but is unnecessary from an implementation point of view. This section provides algorithm 3 that skips the transformation process.

Algorithm 3 Perform axial line placement on a chain of convex polygons

```
input: a chain of convex polygons $C$ and $n = |C| - 1$
output: a minimal set of axial lines $F$
1:  $\alpha \leftarrow 1$ {Source of current axial line in the loop}
2:  $F \leftarrow \emptyset$ {Will contain final set of axial lines}
3:  for $i = 2$ to $n$ do
4:    if $i = n$ or not visible$(c_\alpha|c_{\alpha+1}, c_i|c_{i+1})$ then
5:      $j \leftarrow \alpha - 1$
6:      while $j \neq 0$ or visible$(c_j|c_{j+1}, c_{i-1}|c_i)$ do
7:        $j \leftarrow j - 1$
8:      end while
9:      $F \leftarrow F \cup \{$axialLine$(c_{j+1}|c_{j+2}, c_{i-1}|c_i)\}$
10:     $\alpha \leftarrow i$
11:   end if
12: end for
13: return $F$
```

`visible`(a,b) returns TRUE if a is partially visible to b and FALSE otherwise.
`axialLine`(a,b) returns an axial line that crosses all adjacencies from a to b in the chain.

Algorithm 2 looks at the set of intervals that cover a certain point and chooses the one that first goes as far right as possible and then chooses, from that set, the one that goes as far left as possible. Since this interval is representative of an axial line it would be easier to just extend an axial line from the adjacency being considered as far along the chain as possible then to extend that line as far backwards along the chain as possible. Algorithm 3 does exactly this. The variable α is the index of the source adjacency of the current line that is being extended forwards along the chain. Line 1 in the algorithm indicates that this starts at $c_1|c_2$. The for loop visits each adjacency in turn and checks if the current line cannot be extended from $c_\alpha|c_{\alpha+1}$ to $c_i|c_{i+1}$ were i is the current value of the for loop counter. As soon as the current line cannot be extended from $c_\alpha|c_{\alpha+1}$ to $c_i|c_{i+1}$ the line has been extended as far forward as possible and it must now be extended backwards. At this stage the while loop is entered into. At each step of the while loop an attempt is made to extend the line from $c_{i-1}|c_i$ to an adjacency before $c_\alpha|c_{\alpha+1}$. As soon as this attempt fails the line has been extended as far as possible and the next line is started by setting $\alpha = i$ and the for loop continues until the end of the chain is reached.

3.7 Example

A small example is worked through here to show how algorithm 3 works. The example is shown in figure 3.7. The algorithm draws each axial line by first choosing an initial adjacency. This initial or source adjacency is indexed by α in the algorithm. Initially it is set to index the first adjacency at line 1. This adjacency is then checked for visibility with each adjacency that comes after it starting with $c_{\alpha+1}|c_{\alpha+2}$. Figure 3.7 shows the first source adjacency ($c_1|c_2$) and the adjacency that will be checked for visibility. Note that the adjacencies that are being checked for visibility against $c_\alpha|c_{\alpha+1}$ are indexed by i. The **for** loop in line 3 shows that the initial value of i is 2.

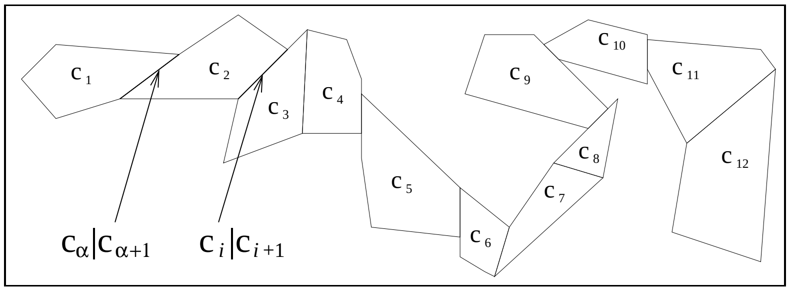


Figure 3.7: Tracing through algorithm 3. Example with $i = 2$.

Now that the first two adjacencies have been chosen the visibility is checked in line 4 of algorithm 3. If they are visible then the loop continues onto the next adjacency. It is clear from figure 3.7 that $c_\alpha|c_{\alpha+1}$ and $c_i|c_{i+1}$ are visible to each other so it is possible to draw an axial line through these two. This is not done yet because it may be possible for such an axial line to cross more adjacencies further on. Since the condition of the **if** statement evaluates to false nothing happens on this iteration of the loop. The next iteration of the loop begins and now $i = 3$. The **if** statement checks to see if $c_\alpha|c_{\alpha+1}$ is visible to $c_3|c_4$. It is easy to see that they are visible to each other from figure 3.7 so the next iteration of the **for** loop begins and $c_4|c_5$ is checked with the same result. Now on the iteration where $i = 5$ it is found that $c_5|c_6$ is not visible to $c_\alpha|c_{\alpha+1}$. This indicates that an axial line can be drawn through all the adjacencies from $c_\alpha|c_{\alpha+1}$ to $c_4|c_5$ but not though $c_5|c_6$ as well and a new line must start at $c_5|c_6$.

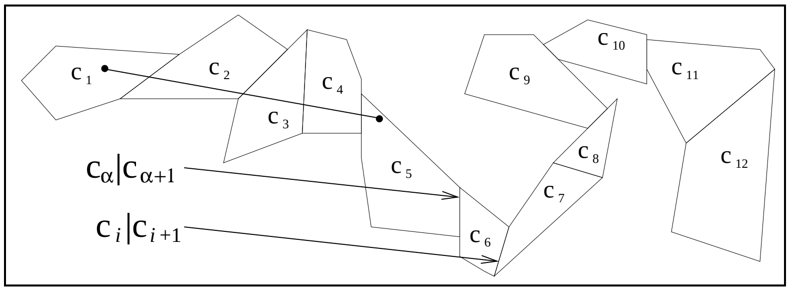


Figure 3.8: Tracing through algorithm 3. Example with $i = 6$.

The lines must also be maximal, so before a new line can be started it must be checked to see if the line that ends at $c_4|c_5$ can be extended further backwards before the source adjacency, $c_\alpha|c_{\alpha+1}$. This is taken care of in the body of the **if** statement from lines 5 to 8. Each adjacency before $c_\alpha|c_{\alpha+1}$ is checked in succession until an adjacency is found that is not visible to $c_i|c_{i+1}$ or the beginning of the chain is reached. These adjacencies are indexed by j. In line 5 j is set to the index of the adjacency before $c_\alpha|c_{\alpha+1}$. In this case j is set to 0 because $\alpha = 1$ which means that j does not index *any* adjacency. The **while** loop is what iterates through these adjacencies. Notice one of the conditions for the while loop is $j \neq 0$ which means that the loop will not execute in this case and line 9 is reached. Note that $j \neq 0$ can be exchanged for $\alpha \neq 1$. Line 9 is what places the axial line in the final set of lines. After placing the line in F a new source adjacency needs to be set. The current value of i is the index of the first adjacency that is not visible to the current source adjacency so it is used as the new source adjacency. This is done in line 10 by setting α to i. Figure 3.8 shows the state of the algorithm at the beginning for the *next* iteration of the **for** loop i.e. when $i = 6$

The algorithm works as before to check if $c_\alpha|c_{\alpha+1}$ is visible to the adjacencies that come after it to draw the "forward" part of the line. From the figure it is easy to see that $c_5|c_6$ is visible to $c_6|c_7$ but not to $c_7|c_8$ so the **if** condition becomes true when $i = 7$ signaling the end of the forward extension phase. The adjacencies before $c_\alpha|c_{\alpha+1}$ must now be checked for visibility with $c_6|c_7$ beginning with the adjacency right before $c_\alpha|c_{\alpha+1}$ which is $c_{\alpha-1}|c_\alpha$ or $c_4|c_5$. In line 5 j is set to $\alpha - 1$ to facilitate this. Figure 3.9 shows the state of the algorithm up until this point.

The current value of i indexes the first adjacency in the chain after $c_5|c_6$ that is *not* visible to $c_5|c_6$. This means that $i - 1$ must be the index used for visibility checking and is reflected in the condition of the **while** loop. Each iteration of the **while** loop checks to see if $c_6|c_7$ is visible to the adjacency indexed by j then decrements j. It continues to do so until j indexes an adjacency that is not visible to $c_6|c_7$. This adjacency happens to be $c_1|c_2$ in the example. Since the loop checked for visibility with all adjacencies from $c_2|c_3$ to $c_4|c_5$ with $c_6|c_7$ it means that an axial line can be drawn through all those adjacencies. The **while** loop is then exited and the line is added and the new source adjacency is set. Note that the index used to add the line is $j + 1$ and not j because j now indexes the first adjacency that is *not*

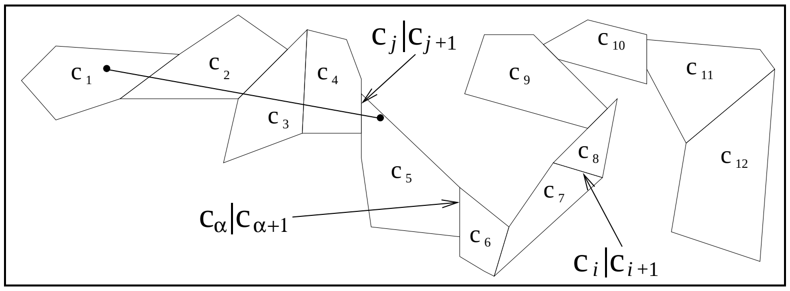


Figure 3.9: Tracing through algorithm 3. Example with $i = 7$ and $j = 4$.

visible to $c_6|c_7$. The state of the algorithm at the beginning of the next iteration is shown in figure 3.10. Note that two axial lines have now been placed.

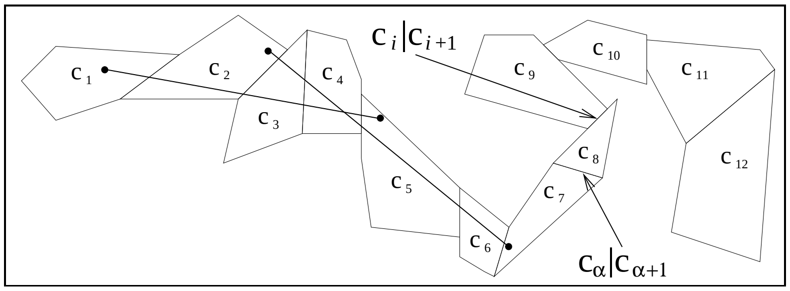


Figure 3.10: Tracing through algorithm 3. Example with $i = 8$.

The algorithm repeats the process to place two more lines and the final solution is shown in figure 3.11.

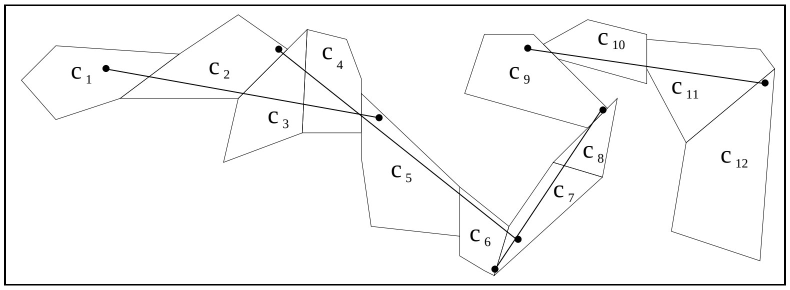


Figure 3.11: The final solution

3.8 Complexity analysis

This section discusses the complexity of algorithm 3. Only algorithm 3 is analysed because it is the algorithm that will be used in practice. Algorithms 1 and 2 were only presented because of their usefulness in deriving and proving the correctness of the solution. The best and the worst case performances are shown to be $\Theta(n)$ and $\Theta(n^2)$ respectively where n is the number of polygons in the chain.

3.8.1 Best case analysis

The best case for the algorithm happens when none of the adjacencies are visible to each other in the chain and an example is shown in figure 3.12. This is expected because each of the adjacencies must be checked for visibility which means there must be at least one operation per adjacency. This section first shows that this special case does cause the algorithm to perform in linear time and then shows that it is impossible to find input that causes the algorithm to perform sub-linearly.

Consider first the body of the loop for an arbitrary i. Since no adjacency is visible to any other in the chain, `visible`$(c_\alpha|c_{\alpha+1}, c_i|c_{i+1})$ is false, therefore the body of the **if** statement is executed. The condition `visible`$(c_j|c_{j+1}, c_{i-1}|c_i)$ which is a condition in the **while** loop is always false so the body of the **while** loop is never executed. No command in the **for** loop has been executed more than once so one iteration of the loop executes in constant time provided that the sub-functions `visible` and `axialLine` execute in constant time. To show this it is first necessary to show that $\alpha = i - 1$ in all iterations of the **for** loop.

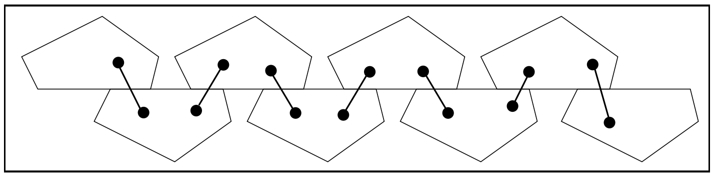


Figure 3.12: Best case performance of the chain algorithm with solution

Before the loop is executed α is set to 1 in line 1 and can only change at line 10 which can only happen at the end of an iteration of the **for** loop. In the first iteration of the loop i is set to 2 and doesn't change until the next iteration of the loop. So $\alpha = i - 1$ for $i = 2$. For all iterations of the loop the condition in the **if** statement is always true and line 10 is contained inside the **if** statement and at the end of each iteration of the loop. Line 10 says that α is set to the value of i. In the next iteration of the loop i is incremented so at the beginning of each iteration $\alpha = i - 1$.

Now the first instance of the `visible` function is in line 4 and has arguments $c_\alpha|c_{\alpha+1}$ and $c_i|c_{i+1}$. Since $\alpha = i - 1$ the function is trying to find if two adjacencies that are in the same convex polygon are partially visible to each other. The only vertices required for this operation are the four that define the adjacencies so the `visible` function executes in constant time.

The other instance of the `visible` function occurs in line 6 and has arguments $c_j|c_{j+1}$ and $c_{i-1}|c_i$. In line 5 j is set to $\alpha - 1$, also $i - 1 = \alpha$ so again the function is trying to find if two adjacencies that are in the same convex polygon are partially visible to each other. So similarly the `visible` function executes in constant time.

The only instance of `axialLine` occurs in line 9 with arguments $c_{j+1}|c_{j+2}$ and $c_{i-1}|c_i$. Now from the arguments above it is clear that $j+1 = i-1$. This indicates that the axial line only has to be placed upon one adjacency which can be done in constant time, c.

Now it has been shown that the body of the **for** loop executes in constant time for all i. It is easy to see that the body of the **for** loop executes $n-1$ times because i goes from 2 to n with a step size of 1. So the time function is $g(n) = c(n-1) + d$ where d represents the time for the initializations. Clearly $g(n) \in \Theta(n)$. $\square$

It must also be shown that there cannot be a better time complexity for the algorithm. The counter for the **for** loop never changes in the body of the loop so the body of the loop is always executed $n-1$ times regardless of the input so the algorithm has to do at least $n-1$ operations and $n-1 \in \Theta(n)$. So you cannot get input that causes the algorithm to perform in sub linear time.

This analysis assumes two things about the arrangement that may not be true depending on the application. The first of these is that two polygons can be adjacent to the same side of another polygon. If the only line of visibility from one adjacency to another runs along an edge or intersects a vertex of any polygon in the arrangement then it may or may not indicate partial visibility. The second assumption is that this does not indicate partial visibility. If for any reason these do not apply then the special case shown in figure 3.13 gives a similar result and is still linear.

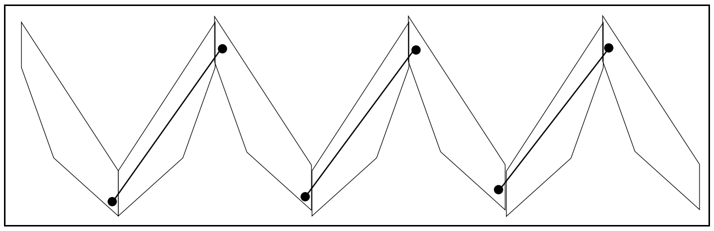


Figure 3.13: Alternative best case performance of the chain algorithm with solution

3.8.2 Worst case analysis

The worst case performance occurs when all of the adjacencies can see each other except the first and the last. Figure 3.14 shows an example of such a case with the solution. The solution always involves two lines and each cross $a-1$ adjacencies if a is the number of adjacencies in the chain. The final part of this section shows that this is indeed the worst case.

Before any analysis is done it is necessary to show that the complexity of the `visible` function depends upon the number of polygons directly and hence upon the number of adjacencies between the polygons. This has been done for the case with only one polygon

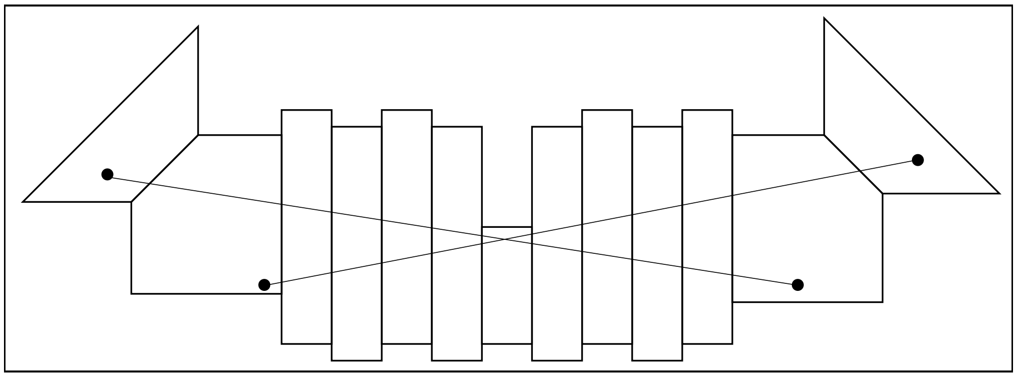


Figure 3.14: Worst case performance of the chain algorithm with solution

in the best case analysis. Section 2.4.2 describes the `visible` algorithm in detail. The first step is to construct an adjacency polygon which can be done independently of the points in the polygon that are not the vertices of the adjacencies between the polygons being considered. This is done by joining up the vertices of the adjacencies. So this process is linear with respect to the number of polygons in the adjacency polygon. The second part of `visible` is to plug this adjacency polygon into the partial visibility algorithm. Since no information about the rest of the polygons is given to the partial visibility algorithm its performance also depends on the number of polygons directly. So `visible` is linear with respect to the number of polygons but since there are as many adjacencies as there are polygons minus one `visible` is linear with respect to the number of adjacencies being crossed as well.

The algorithm goes through two main stages when tackling this special case. The first part is when the first line is being "drawn" and the second part is when the second line is being "drawn". Both parts are shown to happen in $\Theta(n^2)$ time and are then added together to get the final result.

At line 1 in algorithm 3, α is set to 1. The **for** loop begins with $i = 2$. The condition in the **if** statement always evaluates to false until $i = n - 1$ because of the condition that all of the adjacencies are visible to each other except for the first and the last. What needs to be shown first is that the condition in the **if** statement is true when $i = n - 1$, α is always 1 up until the point where $i = n - 1$ because the only other statement that changes it is inside the **if** statement which was never entered into. This means that when $i = n - 1$ $\alpha = 1$, and $c_\alpha | c_{\alpha+1}$ and $c_i | c_{i+1}$ refer to the first and last adjacencies and are not visible to each other because of the condition specified for this special case. So the condition in line 4 is true and the body of the **if** statement is entered into when $i = n - 1$. Note that the first line is still being drawn because the `axialLine` function has not yet been called.

Now that it has been shown that the body of the **if** statement is not entered into until $i = n - 1$ it is necessary to show how many computations happen up until that point. All of the computations in the loop happen in constant time except the `visible` function. It was shown above that the complexity of `visible` linearly depends on the number of adjacencies that are in the adjacency polygon generated. The function takes in two arguments, namely the first and last adjacencies of the chain of polygons that will be considered. Therefore the difference in the indices of these adjacencies gives an approximation of the number of computations multiplied by a constant, c. So for any i the number of computations in `visibility`$(g(i))$ is $c(i - \alpha)$. α is always 1 so $g(i) = c(i - 1)$ for $i < n - 1$. For $i = n - 1$ $g(i)$ still includes the number of computations of the `visible` function which is $c(n - 2)$. The rest comes from the rest of the body of the **if** condition. Line 5 says j becomes $\alpha - 1 = 0$ which means that the body of the **while** loop is not executed and so is a constant number of computations. The `axialLine` function does $O(n)$ computations so $g(n - 1) = c(n - 2) + d + O(n)$.

So to get the running time of the algorithm up until $i = n - 1$, $g(i)$ must be summed up for $i = 2$ to $n - 1$. Which gives:

$c\sum_{i=2}^{n-1}(i-1)+d+O(n) = c\sum_{i=1}^{n-1}(i-1)+d+O(n) = \frac{c}{2}(n^2-3n-2)+d+O(n) \in \Theta(n^2)$

With the inclusion of the `axialLine` function the first line has now been placed and it has been shown to be done in $\Theta(n^2)$. This also means that it took $O(n^2)$ time to place the line. Additionally the next part shows that the next line also takes $O(n^2)$. If it takes $O(n^2)$ time to place a line then it may seem possible for there to be $O(n)$ lines that each take $O(n^2)$ time to place, but there is no such case. This will be shown at the end of the section.

Now all iterations of the **for** loop have been considered besides the iteration where $i = n$. Now i is equal to n so the body of the **if** statement is entered into. Note that it is assumed that the **or** is short circuit so `visible` is not executed and it would have been done in constant time if it had. This means that the major contributing factor to the number of computations is the **while** loop. Now $c_{i-1}|c_i$ refers to the last adjacency in the chain and initially $c_j|c_{j+1} = c_{\alpha-1}|c_\alpha$ which is the adjacency just before $c_{i-1}|c_i$. As before the number of computations of `visible`$(f(j))$ is $c(n - 1 - j)$. Now `visible` is only going to be false once $c_j|c_{j+1} = c_1|c_2$ because of the condition upon the worst case. j is decremented once per iteration of the **while** loop so that loop is executed $n - 2$ times. The number of computations is now equal to the sum of $f(j)$ for $j = 1$ to $n - 2$ plus the time for `axialLine` which is $O(n)$. Which gives:

$c\sum_{j=1}^{n-2}c(n-1-j) = c\sum_{j=1}^{n-2}(n-1-j) = e(n^2 - 3n - 7) \in \Theta(n^2)$

where e is an arbitrary constant.

With the inclusion of the second call to `axialLine` the second line has been placed and the algorithm terminates. Both cases execute only once and independently of each other and both are $\Theta(n^2)$ therefore the worst case is $\Theta(n^2)$. $\square$

What remains to be shown is that there is no special case that has complexity larger than $O(n^2)$ when algorithm 3 is applied to it. Recall from earlier in the section that algorithm 3 could have a greater complexity if there are $O(n)$ lines that take $O(n^2)$ (or anything greater than $O(n)$) to place. This would happen if the lines were placed in a pattern that is depicted in figure 3.15.

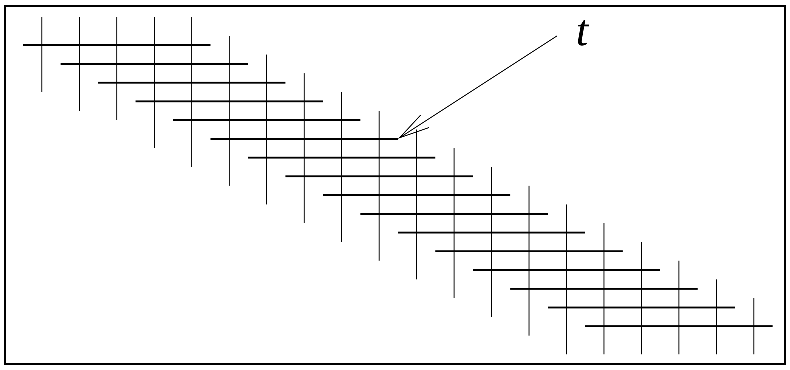


Figure 3.15: Axial lines and adjacencies are represented by horizontal and vertical lines respectively.

Algorithm 3 cannot produce this pattern because firstly it is not an optimal placement as most of the lines are redundant. Secondly as soon as the first line is placed the second line would start at the next uncrossed adjacency and would be extended as far forward as possible resulting in line t and the next line would start after that. This is generalized lemma 3 in apppendix B which says that no minimum cardinality solution to axial line placement in chains of convex polygons can have three of more lines that cross a single adjacency.

Lemma 3 means that if there are n adjacencies in a chain of convex polygons then the sum of the length of the lines must be less than or equal to $2n$ for the solution to be optimal. The bound on the sum of the length is actually a bit lower but it is sufficient for now. This is the property that will be used to show that there is no special case that causes algorithm 3 to perform worse than $O(n^2)$. Note that the bound applies to any solution that comes from algorithm 3 because the algorithm has been shown to always find a solution with the least number of lines..

Suppose that there are m lines from an arbitrary solution produced by algorithm 3 performed on a chain with n adjacencies and the length of line i is denoted by l_i. The complexity of placing line i has been shown to be $O(l_i^2)$ so the complexity of placing all of the lines is $O(\sum_{i=1}^{m} l_i^2)$. Also $2n \geq \sum_{i=1}^{m} l_i$ from the property just shown. This means that

$$\begin{array}{rcl} (2q)^2 & \geq & (\sum_{i=1}^{m} l_i)^2 \\ & \geq & \sum_{i=1}^{m} l_i^2 + \text{some positive terms} \\ & \geq & \sum_{i=1}^{m} l_i^2 \\ \text{now} & & \\ \sum_{i=1}^{m} l_i^2 & \leq & (2n)^2 \in O(n^2) \end{array}$$

So $O(\sum_{i=1}^{m} l_i^2) \subseteq O(n^2)$ which means that the worst case complexity of algorithm 3 is $O(n^2)$.

3.9 Conclusion

This chapter has expanded the notion of chains to its most general form; chains of convex polygons and presented an algorithm to place axial lines in this configuration. This means than any future work done on this special case can focus on making the algorithm more efficient since this configuration cannot be generalised further without creating a new special case.

The work done on this special case began with a restricted case of chains of rectangles in ALP-OLOR meaning that the axial lines had to be orthogonal to an axis Sanders *et al.* [2000a]. The algorithm developed for ALP-OLOR in chains of rectangles was generalised to get an algorithm for ALP-ALOR in a less restricted case of chains of rectangles [Phillips 2001]. However, due to comments in O'Rourke [2002], the approach used in this chapter to solve ALP-ALCP in chains of convex polygons was used in favour of the approach taken to solve ALP-OLOR and ALP-ALOR for this configuration.

The approach taken by this chapter relates the problem to another problem and produces an algorithm that can be generalised to form the heuristic presented in chapter 6.

The algorithm developed in this chapter may not be optimal because it may be possible to find an online visibility algorithm that works in $O(n)$ time, but that falls under visibility and the actual finding of axial lines. This special case plays a large part in the next chapter which deals with another special case named stars of convex polygons.

Chapter 4

Stars of convex polygons

4.1 Introduction

This chapter presents a quadratic algorithm to solve another special case of axial line placement in convex polygons called *stars of convex polygons*. The overall aim of this research is to develop a heuristic to solve the general case of axial line placement and the algorithm for stars of convex polygons is used as part of this heuristic.

This special case consists of a central polygon, which is adjacent to one of the ends of many chains of convex polygons which were presented in the previous chapter. The previous chapter also contains an algorithm to solve axial line placement in chains of convex polygons which is used in this chapter to solve axial line placement in stars of convex polygons in conjunction with a well known graph theory problem named maximum cardinality matching.

This chapter begins with a formal definition of the star of convex polygons, then an overview of the algorithm is given. The algorithm for placing the lines upon the chains follows, then the problem is transformed to maximum cardinality matching and the final algorithm is presented in detail.

4.2 Definition

A star of convex polygons consists of a central polygon, A and a set of n chains of convex polygons, $\beta = \{B^1, B^2, ..., B^n\}$. If b^i_j denotes the jth polygon in B^i then for $i = 0...n$, $b^i_{|B^i|}|A$ must exist. In other words the last polygon in each chain in β must be adjacent to A. Figure 4.1 shows an example of a star of convex polygons.

4.3 Overview of the star algorithm

The algorithm begins by adding polygon A to the end of all the chains creating new chains. If C is chain B^i with A added then $c_{|C|} = A$ and $c_{|C|-1} = b^i_{|B^i|}$. Then a modified chain algorithm is applied to all the new chains starting at b^i_1 for all i from 1 to n. This means

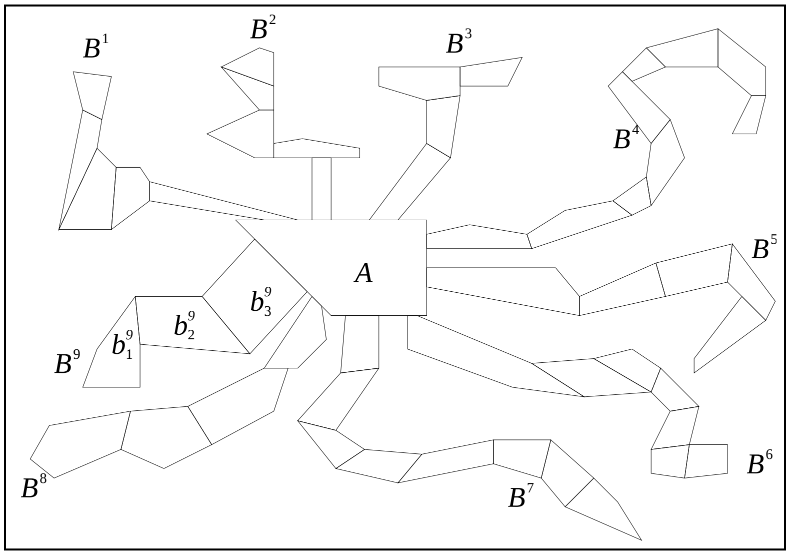


Figure 4.1: A star of convex polygons

that all of the adjacencies are now crossed but the set of lines does not contain the smallest number of lines.

The algorithm reduces the number of lines by merging the lines that cross all the adjacencies between polygon A and the chains. Figure 4.2 shows the central polygon with surrounding chains and the lines that are coming into A from the chains. Two lines can only be merged if each adjacency crossed by one of the lines is partially visible to each other adjacency crossed by the other line. If these two lines start in the same polygon and extend away from each other, then the shorter they are the more likely it is that they will merge. All of the lines that can be merged start in A and extend into their respective chains, therefore, they should be as short as possible to maximise the chance of them merging. The chain algorithm is modified to ensure that the lines that are to be merged are as short as possible.

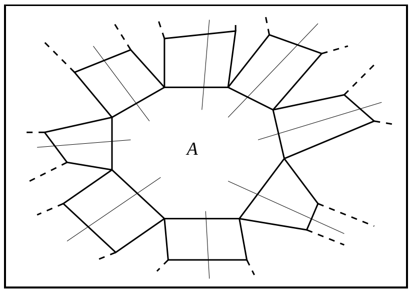


Figure 4.2: The lines to be merged by maximum cardinality matching in the star algorithm

Each time a merge is made the number of lines is reduced by one, so it is necessary to ensure that the number of merges is maximised. This is done using maximum cardinality matching. Each line is a node in the graph and each possibility for a merge between two lines is an edge. The matching is transformed into a set of lines and these lines replace the lines that originate in A. This produces a set of lines with minimum cardinality.

This set of lines is not the maximal set because the part of the chain algorithm that extends the lines backwards is omitted. However, maximising the lines is relatively simple and is done by trying all possibilities.

4.4 The modified chain algorithm

This algorithm is the same as algorithm 3 given in chapter 3 except the **while** loop is excluded. The purpose of the **while** loop in algorithm 3 is to extend the lines backwards so they are as long as possible. It plays no role in finding the set of lines with minimum cardinality. Note that because of this, algorithm 4 does not meet the requirements of axial line placement. The maximizing of the lines has been left out for now because algorithm 4 ensures that the last line that crosses the last adjacency in the chain is the shortest line to do so.

Algorithm 4 produces the same number of lines that algorithm 3 produces because no part of the condition in the **if** statement is dependent on any part of the **while** loop and therefore hasn't any part in determining when the **if** statement is entered into. An axial line is placed each time the body of the **if** statement is entered into so algorithm 4 places the same number of axial lines as algorithm 3 does. The fact that the last line in this solution is the shortest is expressed in theorem 2 but before that property can be proven it is necessary to show that there can only be one such line in any set.

Algorithm 4 Modified algorithm to perform axial line placement on a chain of convex polygons.

```
input: a chain of convex polygons $C$ and $q = |C|$
output: a minimum cardinality set of axial lines $F$
1: $\alpha \leftarrow 1$ {Source of current axial line in the loop}
2: $F \leftarrow \epsilon$ {Will contain final set of axial lines}
3: for $i = 2$ to $q - 1$ do
4:     if $i = q - 1$ or not visible($c_\alpha | c_{\alpha+1}, c_i | c_{i+1}$) then
5:         $F \leftarrow F \cup \{$axialLine($c_\alpha | c_{\alpha+1}, c_{i-1} | c_i$)$\}$
6:         $\alpha \leftarrow i$
7:     end if
8: end for
9: return $F$
```

`visible`(a, b) returns TRUE if a is partially visible to b and FALSE otherwise.
`axialLine`(a, b) returns an axial line that crosses all adjacencies from a to b in the chain.

Lemma 2 *In any minimum cardinality set of axial lines that crosses the adjacencies in a chain of convex polygons there is only one line to cross the the first adjacency in the chain and only one that crosses the last adjacency in the chain.*

Proof - by contradiction

Assume that there are two distinct lines, a and b that cross the first adjacency in a chain, C, and both belong to the same minimum set of axial lines, A, that cross the adjacencies in C. Since a and b are distinct either line must cross one or more adjacencies that the other does not. Without loss of generality assume that a crosses one or more adjacencies that b does not. Let l be the first adjacency not to be crossed by b but crossed by a. Since b crosses the first adjacency but not l it follows from lemma 1 in section 3.4 that no adjacency after l is crossed by b either. Since a crosses the first adjacency and l it also follows from lemma 1 that a crosses all of the adjacencies between the first adjacency and l. Finally b crosses the first adjacency and the adjacency before l (from definition of l) so using lemma 1 once more, it follows that b crosses all of the adjacencies between the first adjacency and l. Now it has been shown that the *only* adjacencies that b crosses are all of those before l but it has also been shown that a crosses all of these adjacencies as well so a crosses all of the adjacencies b crosses. This means that b is redundant so A contains a line that is not needed and hence not the minimum number of lines which contradicts the assumption that A is a minimum cardinality set of lines. To show that only one line crosses the *last* adjacency the chain just has to be relabelled so that the first is the last and the last is the first and the proof follows. □

Theorem 2 *Let $m = |C| - 1$. If algorithm 4 is applied to a chain of convex polygons C and F is the set of lines produced by algorithm 4, then the axial line from F that crosses the last adjacency in C ($c_m|c_{m+1}$) is the shortest line to do so from all possible sets of axial lines to cross the adjacencies in C with minimal cardinality.*

This theorem does not say that there is only one optimal solution. There could be another set of lines with minimal cardinality where the line that crosses the last adjacency is longer but the theorem says that this line cannot be shorter

Proof

If F is the set of lines produced by algorithm 4 then F has minimum cardinality because 4 finds a minimum cardinality set of lines. So by lemma 2, the line that crosses the last adjacency is the only line to do so. Call this line a and call the last adjacency l. Assume that a is not the shortest line that crosses l and there is another set of lines, E that contains a line, b that crosses l but is shorter than a as shown in figure 4.3. Clearly b doesn't cross some adjacencies that a crosses so E must contain another line that crosses them, call this line b'. Call the line that comes before a in F a' and let l' be the last adjacency that it crosses.

Now b' cannot extend before a'. To see why, consider the adjacency l'' where a' began. Algorithm 4 extended a' forward from l'' and stopped at l' because n, the adjacency just right of l', was not visible to l''. Now, b' crosses the adjacencies left uncrossed by b, so it crosses some adjacency between n and l. Therefore, b' must cross n in order to cross l'' because an axial line must cross consecutive adjacencies in a chain (lemma 1). However, l''

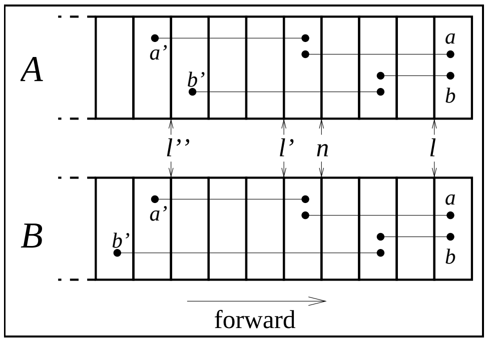


Figure 4.3: Line placements at the end of a chain of convex polygons. The placement A is possible while B is not.

is not visible to n so b cannot cross l'' and therefore cannot extend backwards past l'' and hence, cannot extend before a'.

b' does not cross the source of a' so a' extends further backwards than b' which is the same situation with a and b. This means that the pattern will be the same with the lines that come before a' and b' and repeats until the first adjacency is reached. Now the line in F that crosses the first adjacency corresponds to a line in E which does not cross the first adjacency, so one or more lines must be added to E to cross the first adjacency and hence there are more lines in E than there are in F and so E is not a minimum cardinality set and a contradiction is reached. $\square$

4.5 Transformation to the matching problem

This section presents an algorithm to transform axial line placement in a star of convex polygons to maximum cardinality matching and explains why it does so. The algorithm first applies algorithm 4 to the chains adjacent to the central polygon. Each chain that is given as input to algorithm 4 must include the central polygon. Then the process of producing the graph for matching begins. A check is done to see if each pair of lines that crosses the adjacencies involving the central polygon can be merged. If they can be merged then two vertices are added that represent the two lines being merged. If these vertices don't exist, and an edge between these two vertices is added to represent the fact that these two can be merged. This transforms the problem to the matching problem. The algorithm to do this is algorithm 5.

Algorithm 5 also returns the set of lines that cross the adjacencies of the chains in β. This doesn't play a part in transforming to the matching problem but is needed for a later stage since these are the lines that will be used to form the final set.

Consider the first **for** loop, particularly line 6. This loop applies algorithm 4 to all of the chains that are adjacent to the central polygon where each chain includes the central polygon. Each of these chains has the minimum number of lines that cross their adjacencies if these chains are considered on their own, which means that all of the adjacencies in the

Algorithm 5 Transformation to maximum cardinality matching

input: a star of convex polygons with central polygon A and a set of chains of convex polygons, $\beta = \{B^1, B^2, ..., B^n\}$.

output: a graph, $G = (V, E)$ and a set of sets of axial lines $\mathcal{F} = \{f_1, f_2, ..., f_n\}$

```
1:  $V \leftarrow \emptyset$
2:  $E \leftarrow \emptyset$
3:  $n \leftarrow$ number of chains in $\beta$
4:  **for** $i = 1$ to $n$ **do**
5:      $\gamma \leftarrow B^i \cup A$ $\{\gamma$ becomes $B^i$ with $A$ added to the end$\}$
6:      $f_i \leftarrow$ the set of lines formed from applying algorithm 4 to $\gamma$
7:      $l_i \leftarrow$ the line in $f_i$ that crosses $b^1_{|B^i|}|A$
8:  **end for**
9:  **for** $i = 1$ to $n$ **do**
10:     **for** $j = i + 1$ to $n$ **do**
11:         **if** merge?$(l_i, l_j)$ **then**
12:             **if** $l_i \notin V$ **then**
13:                 $V \leftarrow V \cup l_i$
14:             **end if**
15:             **if** $l_j \notin V$ **then**
16:                 $V \leftarrow V \cup l_j$
17:             **end if**
18:             $E \leftarrow E \cup (l_i, l_j)$
19:         **end if**
20:     **end for**
21: **end for**
22: return $\mathcal{F}$ and $G$
```

merge?(a, b) returns TRUE if all of the adjacencies that a crosses are partially visible to all the adjacencies that b crosses, and returns FALSE otherwise.

star have now been crossed. However, this does not mean that this new set of lines has a minimum cardinality so more work is required.

One possible way to reduce the number of lines in this new set is to merge the lines across the central polygon, because doing this would reduce the number of lines by one. Note that all of the lines that cross the adjacencies that involve the central polygon are the lines that cross the last adjacency of all the chains with the central polygon added on to the end as shown in figure 4.4. To see if any of these lines can merge it is necessary to check if all of the adjacencies that each line crosses are visible to each other, which means that the lines that are being merged should be as short as possible so that the smallest number of adjacencies have to be checked. However, it is easy to see from figure 4.5 that the only adjacencies that need to be checked are the first adjacencies that are crossed by the two lines that are being merged, but having the lines as short as possible means that the adjacency polygon is small and the chance of the adjacencies being visible is maximised and hence the chance of lines being merged is maximised. Lemma 2 shows that these lines are, in fact, as short as possible.

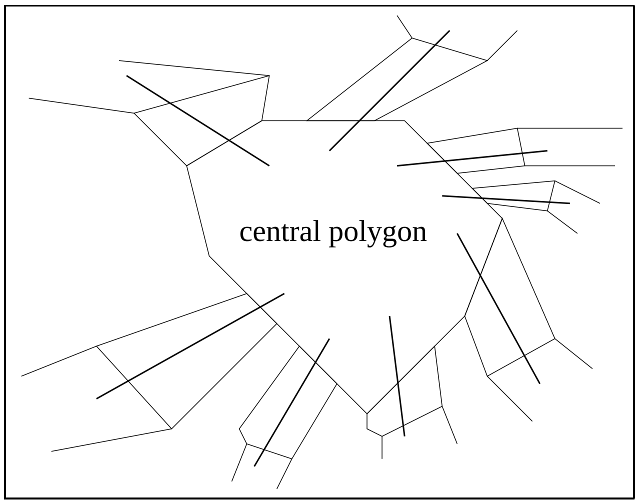


Figure 4.4: A star configuration of convex polygons with axial lines placed upon the chains

A minimum cardinality set of axial lines that crosses the star of convex polygons in question is found by finding a maximum cardinality matching on the graph produced by algorithm 5 then placing the lines that correspond to the matching.

Now merging two lines across the central polygon reduces the number of lines in the set by one, which means that as many merges as possible should be done to make the set have a minimum cardinality. To do this it is necessary to find all possibilities for merges and then find the configuration that has the most merges. Finding all of the possibilities is done by lines 9 to 21 in algorithm 5, which also generates a graph corresponding to these merges. So the problem has now been transformed to the maximum cardinality matching problem.

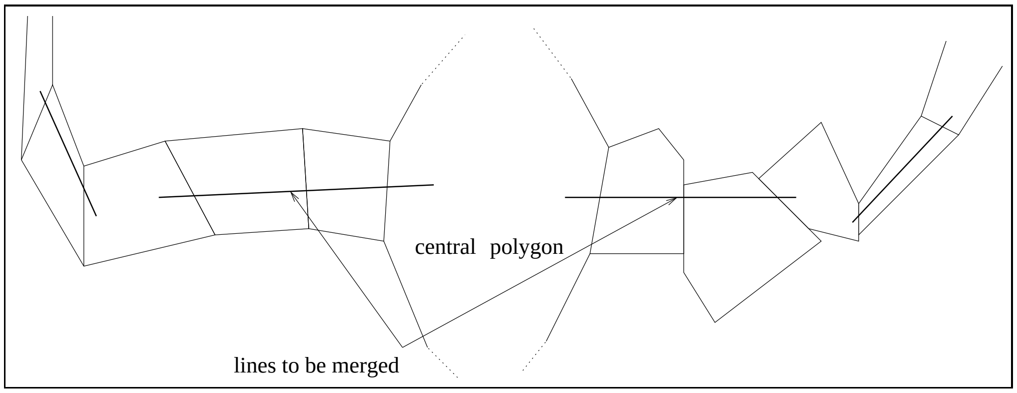


Figure 4.5: Two chains in a star with lines to be merged across a central polygon

Complexity analysis

The complexity of algorithm 4 has been shown to be $O(s^2)$ in chapter 3 where s is the number of polygons so the complexity of placing the axial lines in *all* the chains in the star is $O(m_1^2 + m_2^2 + ... + m_n^2)$ where n is the number of chains in the star, and m_i is the number of polygons in chain m_i. If q is the number of polygons in the configuration then $2q \geq (m_1 + m_2 + ... + m_n)$. Now

$$\begin{aligned}(2q)^2 &\geq (m_1 + m_2 + ... + m_n)^2 \\ &= m_1^2 + m_2^2 + ... + m_n^2 + O(m_1 + m_2 + ... + m_n) \\ &= m_1^2 + m_2^2 + ... + m_n^2 + O(q)\end{aligned}$$

So $O(m_1^2 + m_2^2 + ... + m_n^2)$ reduces to $O(q^2)$

The complexity of checking whether a line can be merged with another line is linear because it only requires that two adjacencies be checked for visibility. However, it is linear in terms of the number of polygons in each chain which is bounded by $m_i + m_j$. This means that the line from chain 1 can be checked with no more than $n - 1$ other chains making the complexity of one check $O(nm_1 + m_2 + ... + m_n)$ which generalises to $O((n - i)m_i + m_{i+1} + ... + m_n)$ for the ith stage. Summing up all the stages gives $O(n(m_1 + m_2 + ...m_(n - 1)) - (m_1 + 2m_2 + ... + (n - 1)m_{n-1}) + (m_2 + 2m_3 + ...(n - 1)m_n))$ which simplifies to $O(nq - (m_1 + m_2 + ... + m_{n-1}) + (n - 1)m_n)$. n and m_n are bounded by q so this further simplifies to $O(q^2)$ which is the complexity of algorithm 5.

The matching itself is $O(i\sqrt{j})$ where j is the number of vertices and i is the number of edges in the graph. The number of vertices is bounded by the number of chains and the number of edges is at most j^2 so the worst case complexity of the matching is $O(n^2)$. Summing up the complexities gives $O(q^2 + n^2)$ as the worst case complexity.

Now n is bounded by q so the worst case complexity of the given algorithm to place the smallest number of axial lines in a star of convex polygons is $O(q^2)$ where q is the number of polygons.

4.6 Maximising the lines

Maximising the lines in the chains apart from the lines that cross into the central polygon is a simple matter of extending them backwards away from the central polygon. To extend each line that has been merged it is a simple matter of trying all possibilities. This is done by first extending the line as far as it can go in one direction (forwards) then form another line by extending the *original* line in the opposite direction (backwards) by one and extending it as far forwards as it can go. Now form another line by extending the original line *two* adjacencies backwards and extend it as far forwards as possible. Carry on forming lines by extending the original backwards by one more each time and then extending it forwards until the original cannot be extended any further backwards. Just choose the longest line to get the longest line.

The only lines left to maximise are the lines that cross into the central polygon but did not get merged, which means that they may be extended into any of the other chains. Maximising these lines can be done by applying the same process to get the longest merged line to each possible combination. This would still be polynomial but may be inefficient but the definition of maximising in axial line placement says that the lines in the minimum solution should be as long as possible but does not say that the minimum solution must be the one with the longest possible lines so it may not matter which chain an unmerged line extends into.

4.7 Worked example

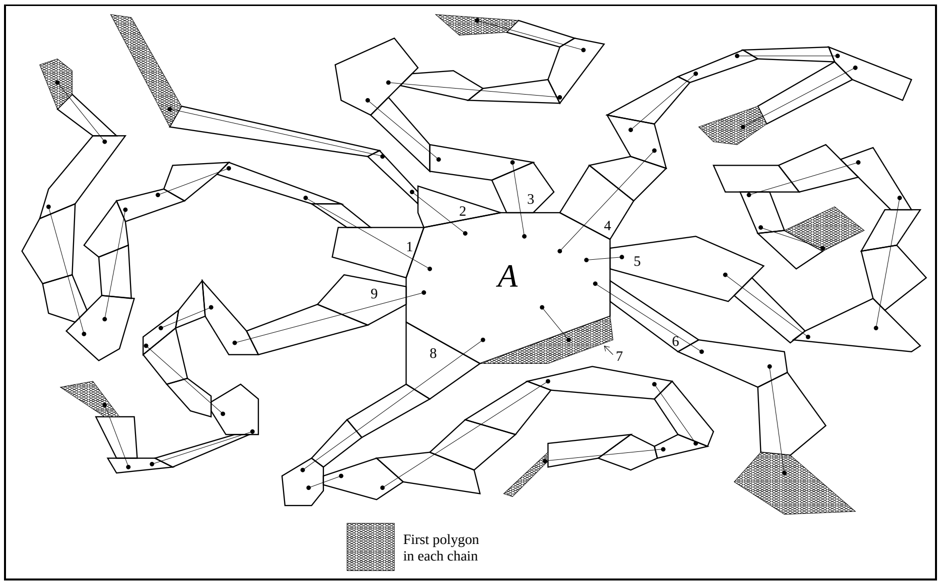


Figure 4.6: A star configuration of convex polygons with unextended axial lines placed upon the chains. Polygon A is the central polygon. The chains are labelled 1 to 9.

An example is worked through in this section to clarify the star algorithm. The first part of the algorithm uses the modified chain algorithm to place axial lines upon the adjacencies in all the chains, starting from the polygons furthest away from the central polygon chainwise. Figure 4.6 shows an example star with the modified chain algorithm applied as specified in lines 4 to 8 in algorithm 5. The modified chain algorithm starts at the beginning of each chain as shown by the hatching in figure 4.6, and continues placing lines in the chain until the central polygon, A, is reached. This means that all the lines that end in A are the shortest axial lines that cross the last adjacencies in their respective chains. They are also the lines that are candidates for merging.

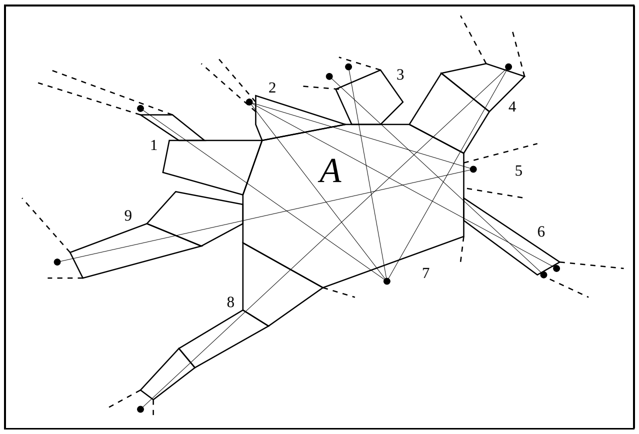


Figure 4.7: All the possible merges across the central polygon.

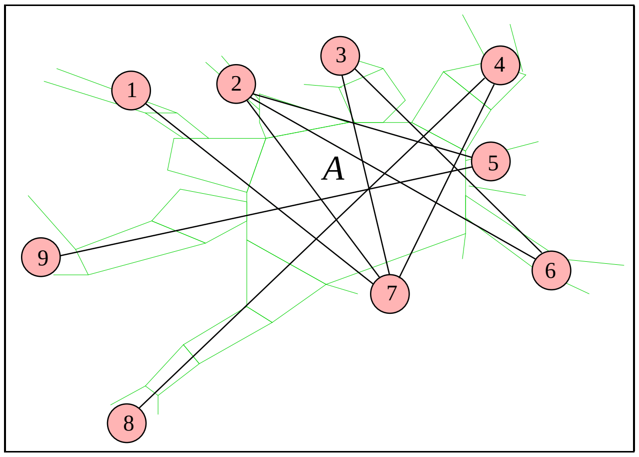


Figure 4.8: A graph representing all possible merges across the central polygon.

Figure 4.7 shows all the possible lines that can come from the merging of the lines that end in polygon A without extending them backwards. Lines 9 to 21 produce the graph shown in figure 4.8 where the vertices have the same labels as the chains. If the edge from vertex 1 to vertex 7 is chosen it means that a line must be drawn to cross adjacencies in chains 1 and 7.

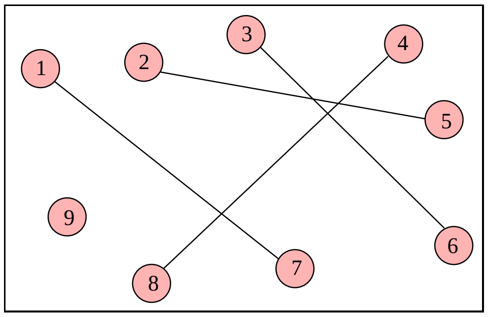


Figure 4.9: A maximum cardinality matching on the graph from figure 4.8

Figure 4.9 shows the result of the maximum cardinality matching algorithm. In this case, the matching indicates that the lines from chains 1, 2, 3 and 4 should be merged with the lines from chains 5, 6, 7 and 8 respectively in order to maximise the number of merges and find the smallest set of axial lines to cross the adjacencies in the star shown in figure 4.8. The fact that vertex 9 was not matched indicates that the line from chain 9 could be merged with another line but doing so would not reduce the number of lines in the final solution if this particular matching was used.

Figure 4.10 shows all the axial lines after the specified merges have been done. Note that the line from chain 9 is left unmerged. However, doing so would not add an axial line but neither would it take one (or more) away.

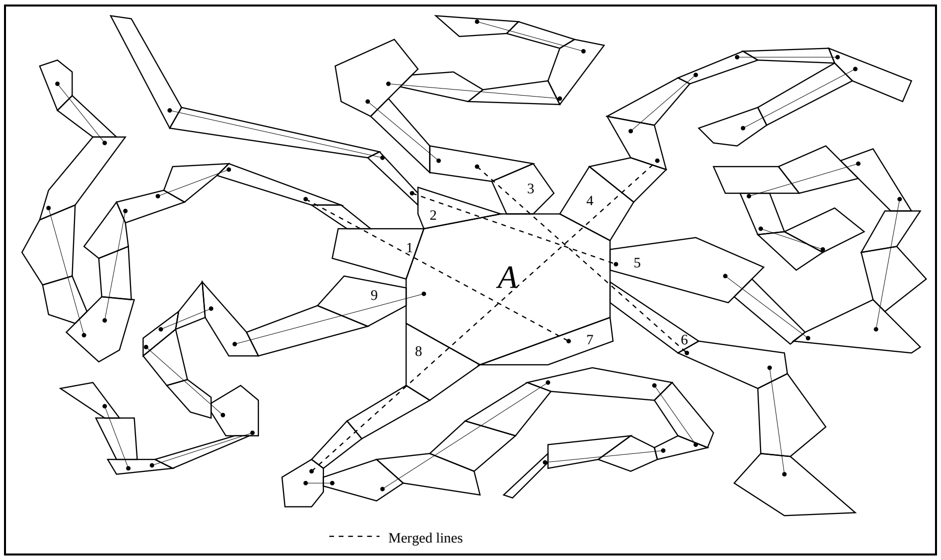


Figure 4.10: A solution to axial line placement that has the smallest number of lines.

The final solution requires that the lines be maximal. This can be done using any method that leaves the lines unextendable. Figure 4.11 shows such a set.

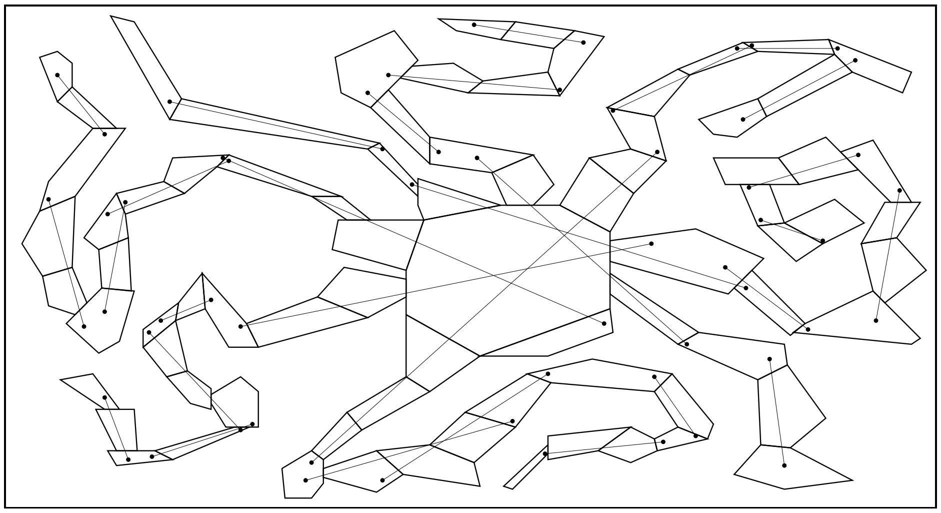


Figure 4.11: A minimal cardinality set of lines with maximal length

4.8 Conclusion

This chapter presents a special case that has not been considered before in any form by research in the area of axial line placement. Previously, work done on special cases of axial line placement has extended the notion of chains to trees of polygons in order to get more general special cases that may have polynomial solutions. However, Phillips [2001] shows that this is not a viable strategy for convex polygons because the tree's usefulness is compromised by the many restrictions that are required. This chapter solves this problem by suggesting a new direction in which to expand the notion of chains in order to obtain special cases that have polynomial time solutions. Chapter 8 discusses some additional special cases that are similar to the star that may have polynomial time solutions.

The purpose of finding a polynomial solution to this special case is to integrate it into a heuristic for the general case. However, in the general case, a star may share several chains with other stars. A solution to this problem is given in the next chapter.

Chapter 5

Networks of stars

5.1 Introduction

Networks of stars are the last of three special cases considered by the research presented in this dissertation. They are constructed from stars of convex polygons which are discussed in the previous chapter and the main component of a star of convex polygons is the chain of convex polygons which are discussed in chapter 3.

In previous chapters, polynomial solutions are found for axial line placement in chains and stars of convex polygons. However, networks of stars differ from these special cases because a heuristic has been found to place the axial lines. Additionally, the problem of finding a minimum cardinality set of lines to cross the adjacencies in networks of stars is thought to be NP-complete due to previous work done on deformed urban grids ([Konidaris and Sanders 2002] and [Wilkins and Sanders 2004]) which is considered to be a similar special case.

Generally, a special case in axial line placement is only considered if it is thought that an exact polynomial time solution exists. However, the aim of this research is to develop the heuristic presented in chapter 6 for the general case of axial line placement and this heuristic requires that special cases be detected in the general configuration for preprocessing. If the stars of convex polygons are detected in a general configuration then some stars may share components and cause a conflict. The network of stars special case is introduced to heuristically resolve this conflict.

The following section contains a precise definition of a network of stars. The algorithm is given in the section following that.

5.2 Definition of a network of stars of convex polygons

A network of stars is made up of stars of convex polygons and an example is given in figure 5.1. A star, as defined in chapter 4, is a collection of convex polygons where one polygon is central and the other polygons form chains where one polygon out of the two that is at the ends of each chain is adjacent to the central polygon. In a network of stars the chains can have each end adjacent to a central polygon, though not the same central polygon.

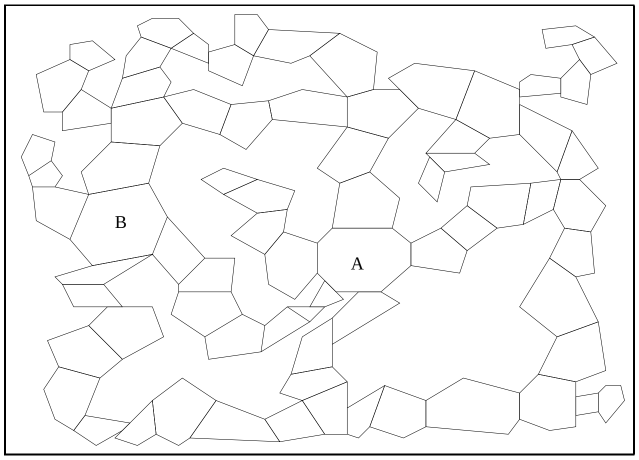


Figure 5.1: A network of stars of convex polygons. Polygons A and B are central polygons.

What remains to be seen is which parts of the network are considered stars for the purposes of the algorithm. In fact, the stars overlap and a choice must be made to determine which stars must be matched first. The way in which the stars can overlap is demonstrated using polygons A and B in figure 5.1. A and B are central polygons where a chain has both ends adjacent to them. In this case the chain is shared between two stars because it forms part of both the stars that contain central polygons A and B. Additionally A will be considered part of the chain in the star formed by B and vice versa for the purposes of the algorithm which will be explained later. A and B are considered neighbours because they share a chain. Figure 5.2 shows what will be considered the star of convex polygons that has polygon A as a central polygon.

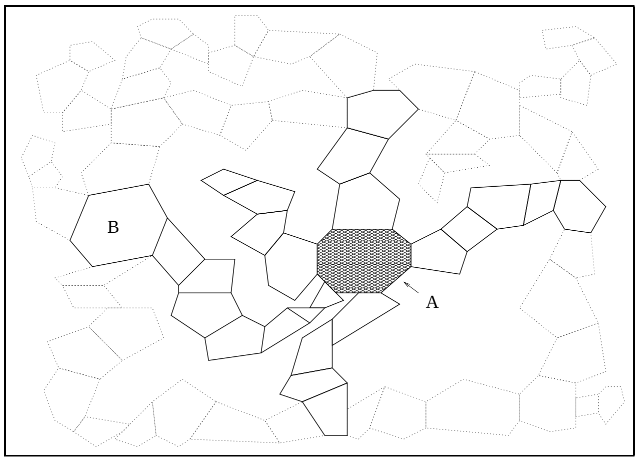


Figure 5.2: The star formed with polygon A as a central polygon.

5.3 A greedy algorithm for networks of stars

The algorithm uses a relatively simple greedy approach by using the star algorithm from chapter 4 on each of the stars in the network, then the star with the most merges will be chosen because that will remove the most lines. This star will be “removed” from the network by adding parts of the star to other stars and placing axial lines upon the rest. The process will then begin again on the new network and continue until there are no stars left and some final cleaning up will be done. The rest of this section will discuss how to choose a star for removal and a method for removing a star.

5.3.1 Removing a star

In order to explain why a star is chosen for removal in the manner that it is, it is necessary to explain exactly how a star is removed. In truth, the star is just reduced to a set of chains which are merged with other stars in the configuration where necessary. Algorithm 6 describes the process with an example to follow.

The basic function of algorithm 6 is to merge the chains that are matched, but the reason why the algorithm is as long as it is, is because of efficiency and the fact that the merging of chains might violate the definition of a star. The algorithm is demonstrated below with the aid of figure 5.3 which shows a star with the star algorithm already applied and the chains labelled 1,2,3,4,5 and 6.

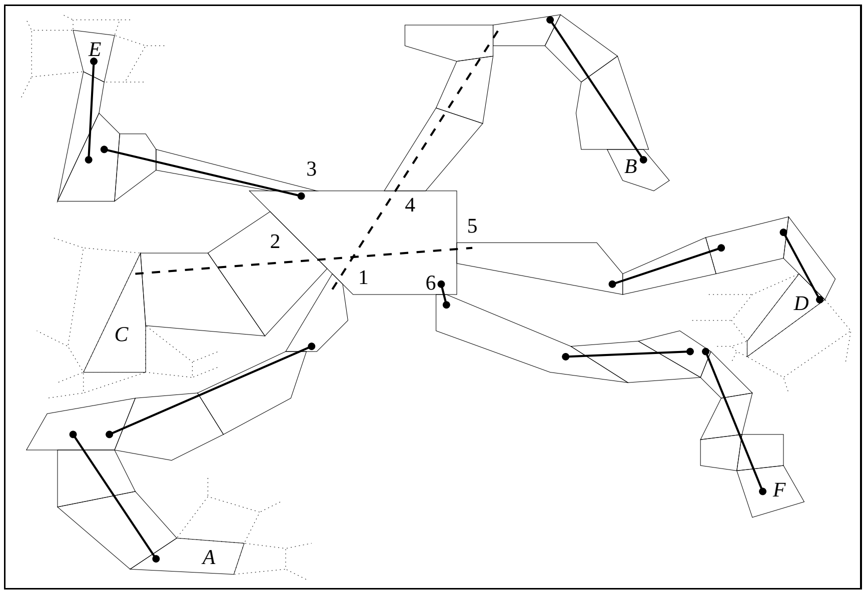


Figure 5.3: A star with exact solution before removal. The dotted lines indicate polygons that are not part of the star.

The heuristic presented in this chapter will find the solution to each star in the network over and over again which means that the chain algorithm will be applied repeatedly and often to the same chain. Lines 12 to 17 and 24 to 28 help reduce the number of times

Algorithm 6 Remove a star from a network of stars.

```
input:
a network of stars $N$
a star to be removed $S$
1:  for each chain $B^a$ in $S$ do
2:    if a line from $B^a$ has been matched with a line from another chain $B^b$ then
3:      merge $B^a$ with $B^b$ to form $B^m$ and replace $B^a$ and $B^b$ with $B^m$
4:      if the ends of $B^m$ are the same central polygon $P$ then
5:        let $a$ be one of the adjacencies that is in $B^m$ and $P$
6:        let $\{T, B^x, L\} = \texttt{reduceChain}(B^m, a)$ /* T is a chain of convex polygons*/
7:        let $b$ be the adjacency at one end of $T$ that isn't $a$
8:        let $\{T, B^y, L\} = \texttt{reduceChain}(T, b)$
9:        extend the lines in $L$ and add it to the solution set
10:       replace both occurrences of $B^m$ with $B^x$ and $B^y$
11:     else
12:       if only one end of $B^m$ is part of a central polygon then
13:         let $a$ be the adjacency at the end of $B^m$ that isn't part of a central polygon
14:         let $\{T, B^x, L\} = \texttt{reduceChain}(B^m, a)$
15:         extend the lines in $L$ and add it to the solution set
16:         replace $B^m$ with $B^x$
17:       end if
18:     else
19:       if both ends of $B^m$ are not part of a central polygon then
20:         use the chain algorithm on $B^m$ and add the solution to the solution set
21:       end if
22:     end if
23:   else
24:     if $B^a$ is shared then
25:       let $a$ be the adjacency that is in $B^a$ and the central polygon of $S$
26:       let $\{T, B^x, L\} = \texttt{reduceChain}(B^a, a)$
27:       replace $B^s$ with $B^x$ in the neighbouring star
28:       extend the lines in $L$ and add it to the solution set
29:     else
30:       use the chain algorithm on $B^a$ and add the solution to the solution set
31:     end if
32:   end if
33: end for
```

`reduceChain`(C, a) uses the chain algorithm on chain C starting from adjacency a without extending the lines backwards forming the set of lines L. Note that a must be an adjacency at an end of C. Then it forms a chain of convex polygons D from the polygons that are crossed into by all the lines in L except the line that crosses the last adjacency in C. The chain E is formed from all of the polygons that the line that crosses the last adjacency in C crosses into. The line that crosses the last adjacency in C is now removed from L and $\{D, E, L\}$ is returned.

the chain algorithm is repeated. Chain 1 in figure 5.3 is the first to be considered by the algorithm and demonstrates this concept.

Line 2 determines that chain 1 has been matched with chain 4. They are now merged to form B^m which is a chain that consists of all polygons in chains 1 and 4. The ends are polygons A and B which are not the same polygon so lines 4 to 10 are skipped. However, A is part of a central polygon so lines 12 to 17 are executed. If lines 12 to 17 weren't executed then B^m would be considered one of the chains that form the star that A is the central polygon of. If this star is only chosen for removal at a much later stage then the chain algorithm will be applied to B^m many times just to get the shortest line that crosses the last adjacency in B^m. Lines 12 to 17 shorten B^m so it is only as long as the shortest line required by the star algorithm, so when the chain algorithm is applied again it only returns that line. The chain algorithm is applied to the rest of B^m and the result is added to the solution set. The chain is reduced with the function `reduceChain`. a is the adjacency at the end of B^m that is part of A and B^x as indicated in figure 5.4. B^x is made up of the shaded polygons in figure 5.4 and the dashed lines make up L.

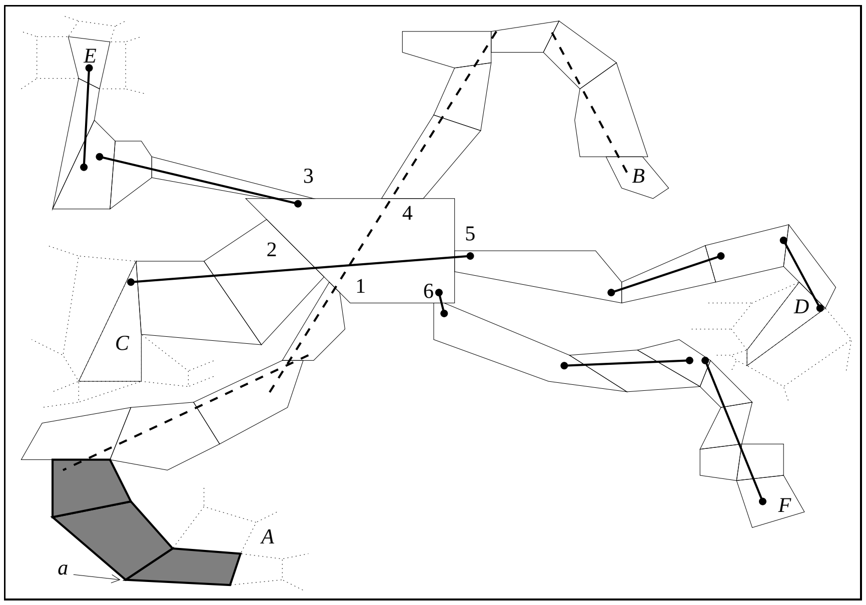


Figure 5.4: A star that is being removed after chain 1 has been considered by algorithm 6. The solid lines are lines that are now in the solution set.

Next to be considered is chain 2 which is matched with chain 5. Chain 2 and chain 5 are both shared and the ends are not the same central polygon so line 3 replaces the chains for C and D with B^m and lines 4 to 22 are skipped. B^m now spans from polygon C, through chain 2 and chain 5 to polygon D and is now shared by the stars that are formed by C and D. The chain algorithm is not applied at this stage because it is impossible to say which side it should start on. If it was started at C to get the shortest line on the D end then it may not have been possible to get the shortest line at the C side. This is one of the main problems with finding the minimum solution to networks and is discussed further in section

5.3.2.

Chain 3 is next in line and it has not been matched but it is shared so lines 24 to 29 are executed. This case is the same as that of chain 1 except that chain 3 is not merged with anything. Chain 3 is reduced so the length of the chain that the chain algorithm has to be applied to is reduced to the essential.

Chains 4 and 5 were merged with chains 1 and 2 in previous steps so they are skipped which leaves chain 6. Chain 6 is not merged with another chain and is not shared so only line 30 is executed. This just uses the chain algorithm and adds the solution to the solution set. No matter what happens in the rest of the heuristic the solution to chain 6 will not change but has to be found at some stage so it is done here. This is not done for efficiency because the chain algorithm would never be applied to it again. Figure 5.5 shows what happens after removal. The parts of the former star that are not shaded have been removed and the shaded polygons form the chains that are still part of the network. The dashed lines are the lines that are in the solution set.

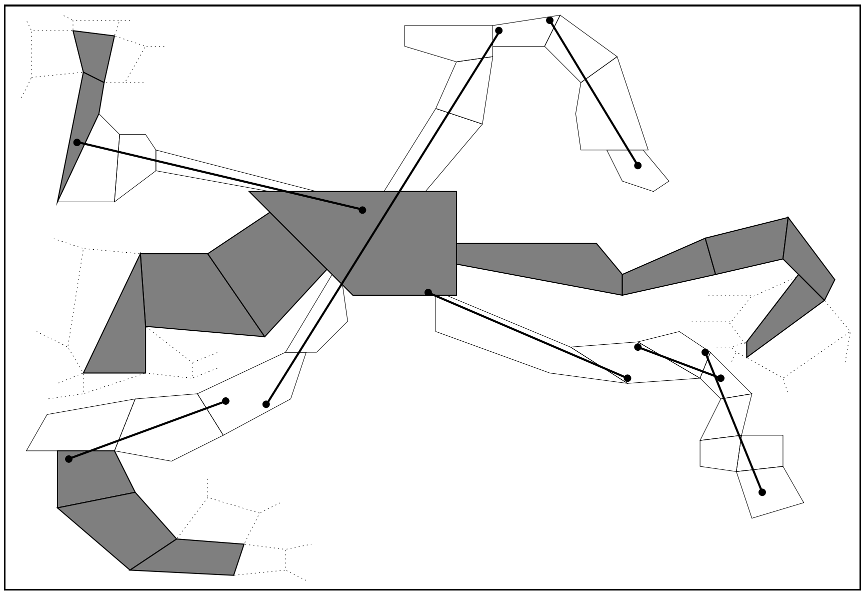


Figure 5.5: The star after removal. The shaded parts are kept and the unshaded polygons with the solid lines are removed.

The only part of the algorithm that hasn't been discussed is contained in lines 4 to 10. This part deals with the case where a star is removed and it creates a chain that has both ends connected to a central polygon which violates the definition of a star. Algorithm 6 breaks up the offending chain into two, though not necessarily in the best way. In figure 5.6, star A has been chosen for removal and there is a match between chains 1 and 2 so they have to be merged, but both of the ends of the merged chain are the same central polygon.

At line 5 in the algorithm, a is randomly chosen. In this case adjacency x will be chosen as a. The function `reduceChain` is applied to the chain starting on x. This results in the division shown in figure 5.7 where the shaded polygons form the first chain to replace the

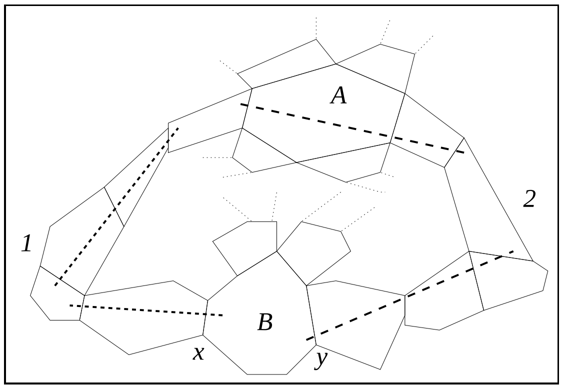


Figure 5.6: Removing star A creates a violation of the definition of a star.

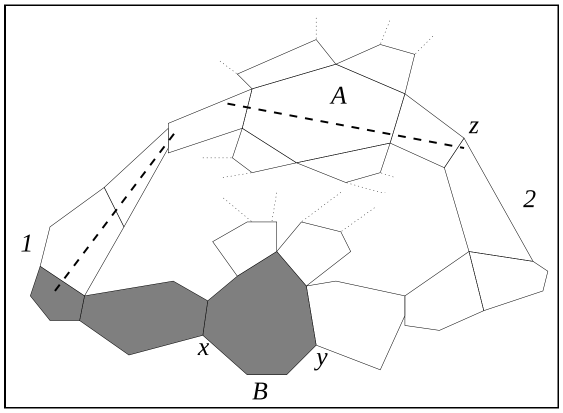


Figure 5.7: After the first reduce. The shaded polygons form the first chain to replace the merged chain.

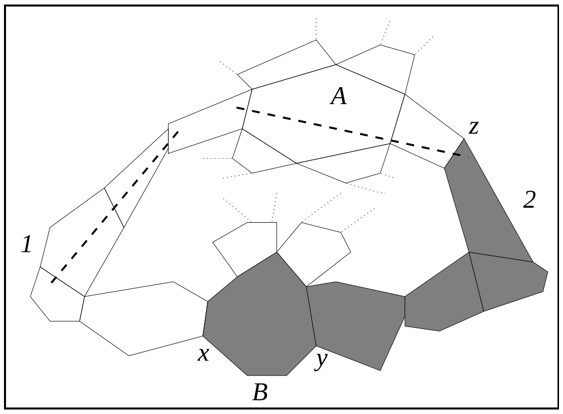


Figure 5.8: After the second reduce. The shaded polygons form the second chain to replace the merged chain.

merged chain.

At line 7, b is assigned adjacency z in figure 5.7 then `reduceChain` is applied to the chain starting from z through chain 1 to adjacency x. This results in the shaded chain shown in figure 5.8 as well as the lines that are added to the solution set. This probably isn't the best solution since starting with adjacency y instead of x would result in different chains and the matching may turn out to be different. Finding the best configuration of chains is thought to be too inefficient and so this heuristic method is used.

Complexity analysis

This analysis is similar to the analysis done for the chain algorithm in section 3.8.2 for the chain algorithm. Let n be the number of chains in the star being removed and let m_i be the number of polygons in the ith chain. Let q be the number of polygons in the star so $2q \geq m_1 + m_2 + ... + m_n$. The function `reduceChain` applies the chain algorithm twice so it is $O(s^2)$ if s is the number of polygons in the chain. The merge process at line 3 is linear, this means that the removal algorithm is $O(2(m_1 + m_2 + ... + m_n)) = O(q)$ when all the chains are shared with different stars. In all other cases, either the `reduceChain` function is called or the chain algorithm is used, which means that the removal algorithm works in $O(q)$ in the best case.

In all other cases, neither the `reduceChain` function or the chain algorithm are used more than twice so each single loop is $O(s^2)$. What remains to be determined is the value of s. Consider the ith iteration of the loop. At lines 6, 8 and 14, the `reduceChain` function receives a chain that is two chains merged whose lengths are m_i and m_j where chain j is the chain that chain i is being merged with. This also holds for the application of the chain algorithm at line 20. At lines 27 and 30, the chain used is only a single chain, and the length of this chain is m_i. Replacing B^m with other chains is linear and can be done no more than twice for each iteration of the **for** loop. Extending the lines is part of the chain algorithm so it is no more than $O((m_i + m_j)^2)$ if the chain can be merged or $O((m_i)^2)$ if not. All of this means that one iteration of the **for** loop is $O((m_i + m_j)^2)$, which is executed n times. However, each chain is matched with only one other chain so the complexity of algorithm 6 is $O(m_1^2 + m_2^2 + ...m_n^2))$ if all of the chains are matched which reduces to $O(q^2)$

5.3.2 Choosing a star for removal

With each matching done in a star the number of lines in the solution will be reduced by one so the number of matchings in a star would be the first criteria for a choice. The second criteria for the choice is choosing a star where the neighbours would have the most matchings done if that star is chosen for removal. The way in which the choice of star affects its neighbours has been somewhat explained in the previous section on removal but is now explained further with the aid of the example shown in figures 5.9 and 5.10.

Figures 5.9 and 5.10 shows the same example just with different lines. Polygons A and B are central polygons for two stars that share the chain shown in the figure. The

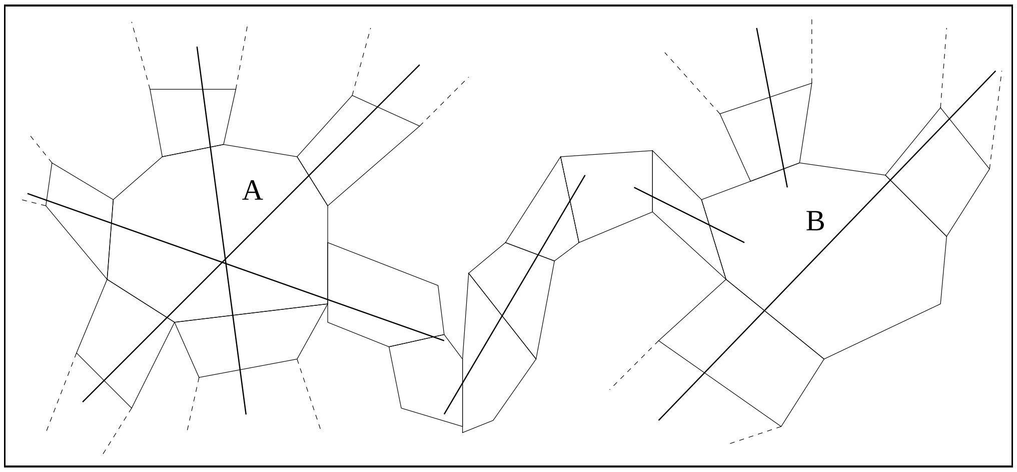


Figure 5.9: Matching on B with the matching on A chosen first

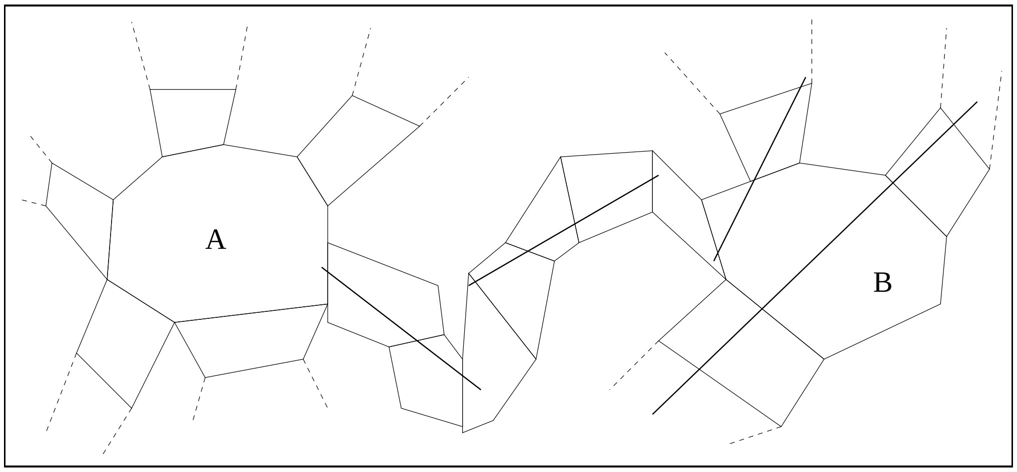


Figure 5.10: Matching on B chosen first.

configuration in figure 5.9 is the line placement where the matching on A is chosen first, then the removal procedure described above is used, then the matching on B is done. The line from the shared chain is matched in A and cannot be extended into the chain as far as the unmatched line could so that results in the line that comes out on the other end not being as short as it could be and hence would not be matched at all resulting in one match across polygon B. Figure 5.10 shows that there are two lines in the matching across B if that matching is done first. This shows that a matching done on a star can reduce the number of matches in its neighbouring stars and so increase the number of lines in the solution set which is not desirable. The matching that results in the most matchings in its neighbours should be chosen. This results in algorithm 7 for choosing a star for removal.

Taking the neighbours into account adds at least another order of complexity. The choice for removal can be based upon the size of the matching in each star only, which can be done

Algorithm 7 Choose a star for removal.

```
1: maxmatch $= -1$
2: for Each star in the configuration do
3:     Save the current state of the configuration.
4:     Use the star algorithm on the star.
5:     Let $k =$ number of matches in the matching.
6:     Remove the star.
7:     for Each of the neighbouring stars do
8:         Use the star algorithm on the star.
9:         $k = k+$ number of matches in the matching on star.
10:    end for
11:    if $k >$ maxmatch then
12:        maxmatch $= k$
13:        beststar $= currentstar$
14:    end if
15:    Revert back to saved state of the configuration.
16: end for
17: Use star algorithm on beststar and remove it.
```

by removing lines 6 to 10 from algorithm 7. Clearly, this may increase the number of lines in the solution set but that is the trade off for efficiency. The version of the algorithm where the neighbours are *not* taken into consideration was not used when the heuristic presented in chapter 6 was implemented for the empirical evaluation of said heuristic.

The final algorithm would be to just choose a star to remove using algorithm 7 repeatedly until there are no stars left.

Complexity analysis

This analysis is similar to the analysis done for the chain algorithm in section 3.8.2. Let k be the number of stars in any network of stars and l_i be the number of polygons in the ith star. If n is the number of polygons in the network then $l_1 + l_2 + ... + l_k \leq 2n$. It is $2n$ and not n because some chains are shared. A star is removed at each stage of the algorithm, meaning that there are k stages. At the first stage of the algorithm the star algorithm is applied to each star in the network then a star is removed. Furthermore, the star algorithm works in q^2 time where q is the number of polygons in a star and the removal is also q^2. Therefore, the first stage of the network algorithm works in $O(l_1^2 + l_2^2 + ... + l_k^2)$ time which reduces to $O(n^2)$.

At the second stage, a star has been removed so there are now $k - 1$ stars. However, some polygons from the removed star may have been added to other stars, meaning that l_i is no longer the length of the ith chain. Let m_i denote the number of polygons in star i, so now $m_1 + m_2 + ... + m_{k-1}$ holds. This means that the complexity of the second stage is $O(m_1^2 + m_2^2 + ... + m_{k-1}^2)$ which reduces to $O(n^2)$ again. This repeats until there is only one star left at the kth iteration which works in order of the size of the final star squared time. This shows that all k iterations work in $O(n^2)$ time so the complexity of the network of stars algorithm is $O(kn^2)$ which reduces to $O(n^3)$ because $k \leq n$.

5.4 Conclusion

The network algorithm completes the work done on special cases for this research. The algorithm presented in this chapter uses a greedy approach to resolve the conflict between stars in a general configuration. However, it may be improved by transforming it to some graph problem by making the central polygons vertices and the chains the edges. Unfortunately, this document does not suggest a suitable graph problem.

The network algorithm differs from the algorithms developed for chains and stars because it is a heuristic and the problem is thought to be NP-complete. However, it is useful because it resolves the conflict between stars when they are detected in the preprocessing stage of the heuristic presented in the following chapter. In addition to this, a network of stars closely resembles a network of roads which is a large application area in space syntax.

The following chapter gives the heuristic for the general case of axial line placement where the network of stars is used as one of the three preprocessing methods for the heuristic.

Chapter 6

Greedy heuristic for the general case of ALP-ALCP

6.1 Introduction

The heuristic presented in this chapter is the overall aim of this research. It uses a generalised version of the chain algorithm from chapter 3 and is called the *line heuristic* because a single line is chosen for the solution set at each iteration of the algorithm. This line is chosen based upon the lines that were chosen in previous iterations, so some method is needed to choose an initial state. This chapter discusses how this can be done by using networks of stars, among other methods. An algorithm for detecting networks of stars is also discussed.

The part of the algorithm that dominates its running time is a function called `search` which is essentially a depth first search. This chapter discusses a data structure that reduces the number of times `search` is called and so increases the efficiency of the algorithm. This data structure is an implementation of a bucket sort.

The algorithm is shown to be $O(lq^3)$ where q is the number of polygons or adjacencies and l is some function of q. The value of l does not depend upon the input set but is a user defined parameter and it restricts the degree to which the `search` function searches. This research only considers values of l that are not of greater order than q^2.

A conceptual view of the algorithm is presented in the following section. Following that, some methods for finding a starting point are discussed as well as the sub functions used in the line heuristic. The algorithm is presented in a way that shows the use of the functions and the data structure then some future work is discussed after the worked example.

6.2 Overview of the line heuristic

The chain algorithm for axial line placement in chains of convex polygons can be seen to work in the following way. It starts at one end of the chain and extends a line as far along the chain as possible. Then the situation shown in figure 6.1 is reached. The iterative step is to find all the uncrossed adjacencies that are part of polygons with crossed adjacencies.

In figure 6.1 the only adjacency that satisfies this condition is a. The next line to be added to the final solution is chosen from all the lines that cross these uncrossed adjacencies. The line chosen has to extend the furthest from any of the uncrossed adjacencies as possible then it must extend furthest backwards if there are ties. The only line to do this in figure 6.1 is the one that would be chosen by the chain algorithm. The process repeats until all adjacencies are crossed. This is the idea that will be used to produce a heuristic for axial line placement in convex polygons.

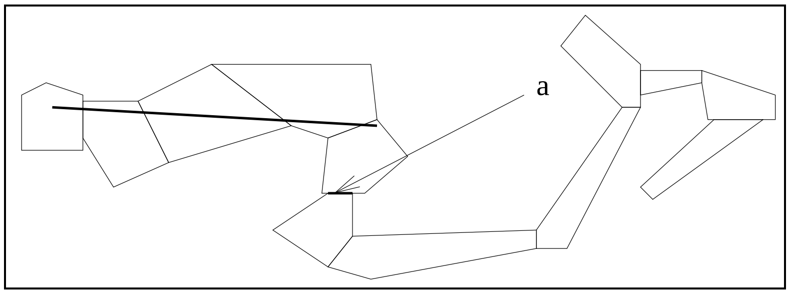


Figure 6.1: The initial state of the chain algorithm

The following steps give a conceptual view of the algorithm.

1. Find a starting point. This could be either a single polygon or a set of lines.
2. Create an empty set F.
3. Let A be all the uncrossed adjacencies that are part of polygons that have *crossed* adjacencies. If there are no lines (so no adjacencies are crossed) then use the adjacencies from the starting polygon.
4. Let L^s be the set of all the lines that cross all of the adjacencies in A.
5. Find the line from L^s that extends the furthest from the adjacency from A that it crosses and add it to F.
6. Repeat steps 3 to 5 until all adjacencies have been crossed.
7. Return F.

Note that, in this context, an adjacency is only considered crossed when it is crossed by a line in F. Also, the distance a line extends from an adjacency is defined as the number of uncrossed adjacencies crossed by the extension.

Finding the starting point required at step 1 is discussed in section 6.3, where three methods are considered. In the case of the chain algorithm the starting point is one of the polygons at either end of the chain. In the example above, the explanation implied that steps 1 to 5 were used to find the first line and this was used as a starting point, but using the first line placed in the chain algorithm would also be a valid starting point. Three starting point methods are discussed, one of which uses the network of stars algorithm.

An example of the adjacencies that will form the set A is shown in figure 6.2. If the two axial lines shown form the starting point then the broken lines indicate the adjacencies that will form set A at step 3. Step 4 finds all the lines that cross these adjacencies.

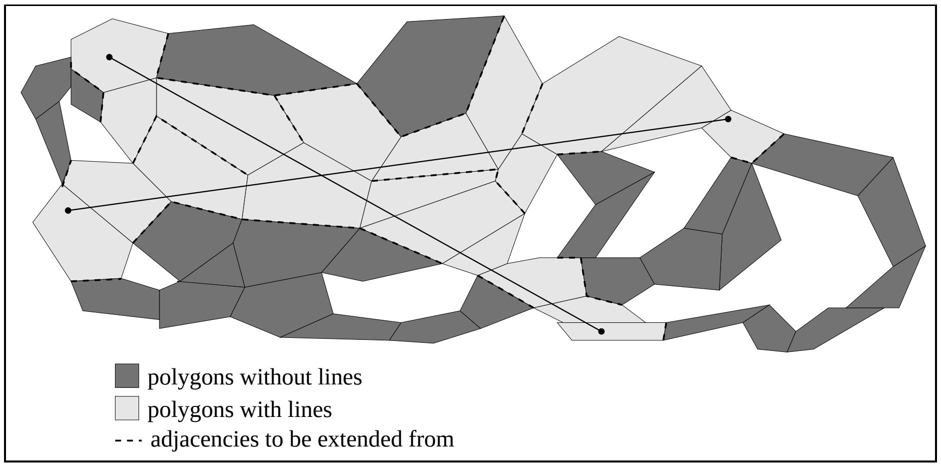


Figure 6.2: Some set of axial lines that may form a starting point for the heuristic

At step 4 of the algorithm it says that *all* the lines that cross each adjacency in A have to be found but this is not actually necessary for step 5. Firstly, lines that are subsets of other lines need not be found. Secondly, only lines that *start* at A should be found then extended backwards. The function `extendFrom` in section 6.4 does this for a single adjacency, so multiple applications of this function get the desired set.

At step 5, only one line is chosen from the lines found at step 4, meaning that step 4 finds lines that cross certain adjacencies multiple times and the cost of finding all of these lines is shown to be high. Section 6.5 presents a data structure that will make sure that step 4 is only executed at most once for each adjacency. This data structure is basically a two dimensional array of lines that is bucket sorted on length.

Section 6.6 presents the algorithm in a form that uses the data structure and the function `extendFrom`.

6.3 Finding a starting point

This section describes the three methods for finding a starting point that will be tested in this research. These are:

1. Random method: pick a random polygon as the start set.

2. Least adjacency method: pick the polygon with the least number of adjacencies that has the adjacency the with smallest x coordinate.

3. Network method: detect any networks of stars embedded in the configuration then use the network of stars heuristic from chapter 5 on the networks. There may be no

stars detected if this method is used, so method 2 is used as a backup.

The first method is only used as a base to compare the other two methods from, and is not expected to work well. Method 2 attempts to maximize the chance of finding a polygon where a line would start. If a polygon only has one adjacency then a line is *guaranteed* to start in that polygon. Furthermore, a line can only cross two adjacencies in a polygon because the polygons are convex, so the number of lines that *must* cross the adjacencies between a polygon and its neighbours is no less than half the number of adjacencies plus one. This indicates that the more adjacencies a polygon has, the more likely a large number of lines will pass through it. If the polygon with the smallest number of adjacencies is chosen for the starting point then there is a high chance of finding the best line that passes through that polygon.

The major difference between the least adjacency method and the network method is that the network method attempts to find some lines for the solution, where the least adjacency method only finds a single polygon to start the line heuristic from. This means the line heuristic that uses the network method is merging two solutions, where the heuristic with the least adjacency method only finds one solution.

The last method is the one that is expected to work the best because the heuristic is specifically tailored for that special case. However, detecting the networks in a configuration is not strictly trivial and is explained below.

Networks of stars can be detected by first detecting all the chains as follows.

1. Make an empty chain C_f and an empty set C_p.
2. Add an unmarked polygon to C_p where that polygon has one or two adjacencies.
3. Make a new set C_p from the polygons adjacent to those in C_p but not in C_f.
4. Add the polygons in C_p to C_f.
5. Remove the polygons in C_p that are not adjacent to exactly two polygons.
6. Repeat 3 to 5 until C_p is empty.
7. Mark all the polygons in C_f except for the end polygons that are adjacent to more than two other polygons.
8. Add C_f it to the set of chains.
9. Repeat 1 to 8 until there are no unmarked polygons with one or two adjacencies.

This process basically picks a polygon with one or two adjacencies then adds the adjacent polygons repeatedly until both ends reach either a polygon with only one adjacency or a polygon with more than two adjacencies, then, it marks those polygons as chosen and picks another polygon. C_f is the set of polygons that have been added to the chain and C_p indicates the current ends of the chain. Figure 6.3 demonstrates this process.

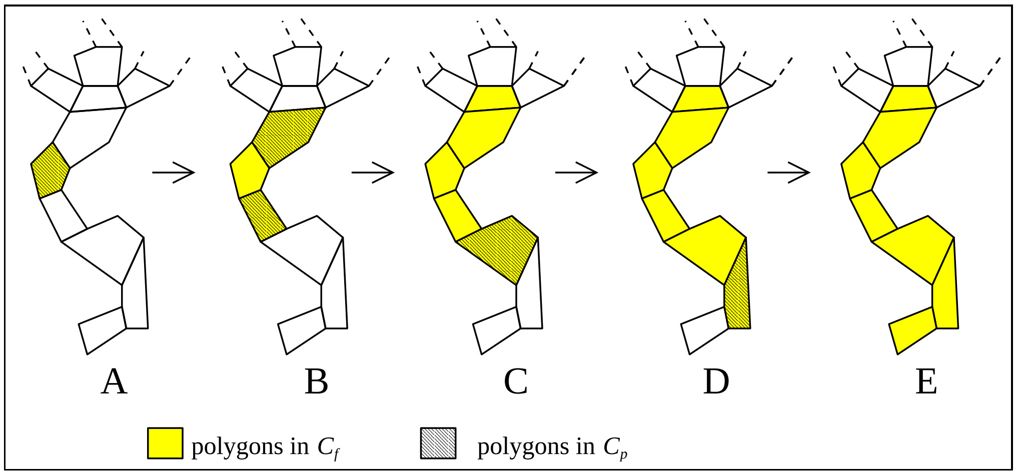


Figure 6.3: Stages in the detection of chains process.

The configuration labelled A shows the state of C_f and C just before step 3 and the configuration labelled B shows the state of C_f and C just before step 3 the *second* time that the algorithm reached that stage and so on. Note that C_p is always contained in C_f right before step 3, hence the overlaps in the diagram.

At C, the polygon at the top is not part of C_p because it was removed at step 5. This polygon is adjacent to more than two polygons so this branch of the chain cannot continue. The reason for stopping at this polygon is that it could form a central polygon for a star. In the star algorithm, the central polygon is added on to the end of all the chains in the star. The process above skips this step by including the possible central polygon in the chain.

After C, C_p only has one polygon, which is extended in D then E where it reaches the end. The end polygon is removed from C_p at step 5 because it is only adjacent to one polygon.

Once the chains have been detected the stars can be detected by placing all the chains that have a common end in one star then throwing away the stars that have one chain. A network of stars is produced by making a link between the stars that share chains.

6.4 Generating all the lines that cross an adjacency

For the purposes of this section, the length of a line is determined by the number of previously *uncrossed* adjacencies that a line crosses, and the extent to which a line extends is determined using this length. The algorithm presented here finds the set of longest lines that *start* at the adjacency in question then returns the set of lines that extend these lines backwards.

The problem with finding all the lines that cross a particular adjacency is that it has not been proved that there are a polynomial number of non-redundant lines that cross this adjacency. The reason why this has not been proved is that the problem is similar to the problem of finding the longest path in a graph which is inapproximable [Karger *et al.* 1997]. However the longest path problem would be solvable in polynomial time if there were a polynomial number of disjoint paths since an exhaustive search could be used to find them all. The same applies to the longest line problem. Each line can be found in $O(n^2)$ time but the number of lines is determined by the geometry of the problem and finding a limit on the number of lines using the geometry is difficult. It is conjectured that the limit on the number of lines in the set found by `extendFrom` is $O(n^2)$.

Finding the limit is left for future work, but in the mean time a user specified limit is placed upon the number of lines found. However, the lower the limit the more efficient the algorithm will be because the search can stop when the limit is reached. The algorithm is given in two parts, `search` and `extendFrom` which are explained next, followed by an example and complexity analysis.

The algorithm is given in two parts as algorithms 8 and 9 which define functions `search` and `extendFrom` respectively. The `search` function is the part of the algorithm that finds sets of lines and `extendFrom` merges the results from `search` to get the final set. It is recommended that the reader refer to the parameter list in the two functions before continuing.

6.4.1 The `search` function

The `search` function is basically a depth first search that stores the current path when an end is reached. It finds all[1] of the lines that can be extended from line L where c is the last polygon in the chain formed by L and $p|c$ is the last adjacency in L. The lines that are subsets of other lines are not included. It does this recursively at line 7 by attempting to extend the line L into each polygon that is adjacent to c by using the partial visibility algorithm to see if the new adjacency is visible to all the adjacencies in L. If the attempt is successful then `search` is called again at line 9 with each extended line as the new L, c as p and the polygon that the line is being extended into as c. Figure 6.4 demonstrates this idea with the n's representing all of the candidate polygons that L may be extended into. Note that polygon p is not considered by the algorithm because $c|p$ is already included in L and a line cannot cross the same adjacency twice.

If the line cannot be extended into any of the adjacent polygons then the line has been extended as far as it can go and needs to be added to the final set of lines. When this happens line 10 is never executed and so *addLine* is never set to **FALSE**, meaning that line 17 is executed which adds L to S.

The idea of putting a limit on the number of lines is implemented at line 5 of `search` by checking if the size of S is less than or equal to the limit given to `search`. If the size of S is equal to *limit* then no more lines can be added to S and the variable *addLine* is set to **FALSE**, resulting in lines 6 to 12 never being executed for the rest of the iterations of the loop. Line 17 isn't executed either because no more lines can be added and the function

[1]If there are more than *limit* number of lines to be found then it will only fi nd *limit* lines.

Algorithm 8 The function `search`

```
function format: search($c, p, L, S, P, limit$)
input:
polygons $c$ and $p$
the current line $L$ as a set of ordered adjacencies
a set of lines $S$
a set of convex polygons $P$
an integer $limit$
1: the length of a line is the number of previously uncrossed adjacencies crossed by the line
2: let $p|q$ denote the adjacency between polygons $p$ and $q$
3: let $T$ be a line as a set of ordered adjacencies
4: $addLine \leftarrow$ TRUE
5: if $|S| \neq limit$ then
6:     for each polygon $n$ adjacent to $c$ that isn't $p$ do
7:         if canAddTo($n|c, L$) then
8:             $T \leftarrow$ addAdjacency($n|c, L$)
9:             $S \leftarrow S \cup$ search($n, c, T, S, P, limit$)
10:            $addLine$ = FALSE
11:        end if
12:    end for
13: else
14:    $addLine \leftarrow$ FALSE
15: end if
16: if $addLine$ = TRUE then
17:    $S \leftarrow$ addLine($L, S$)
18: end if
19: return $S$
```

`addAdjacency`(a, L) inserts adjacency a at the end of the ordered list of adjacency L.
`addLine`(L, S) adds line L to the set of lines S.
`canAddTo`(a, L) checks to see if adjacency a is visible to all adjacencies in line L using the partial visibility algorithm.

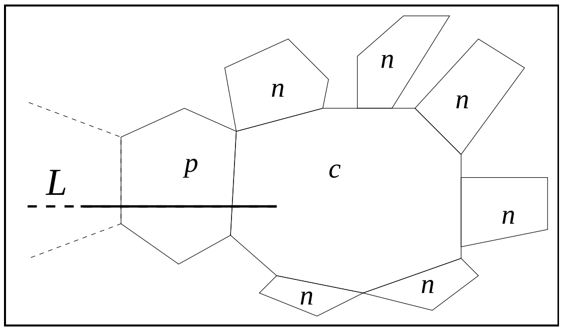


Figure 6.4: Demonstration of a recursive step in the `search` function.

returns S. Note that it may be more efficient to integrate the limit check into the stop condition of the for loop so unnecessary iterations are not executed. However, the number of polygons adjacent to c is probably small compared to the rest of the problem so this is a minor concern. Clearly, if the limit is reached before all the lines are found then all of the lines will not be found.

6.4.2 The `extendFrom` function

The `extendFrom` function, given as algorithm 9, uses results from `search` to find the longest line that extends the furthest from adjacency a. To clarify look at lines X and Y in figure 6.5. Line Y crosses five adjacencies whereas line X only crosses four adjacencies but X is the line wanted because it extends the furthest from a. Y only extends two adjacencies away from a in either direction but X extends three adjacencies to the left.

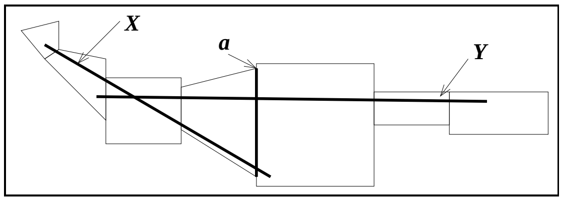


Figure 6.5: The axial lines are X and Y, and a is an adjacency.

First, `search` is used at lines 6 and 7 to get all the lines that *start* at a. Note that the only difference between the two calls to `search` is the order of the polygons x and y, the two polygons that form adjacency a. Figure 6.6 shows an example of the axial lines that are found by the calls to `search` at lines 6 and 7. The first call starts off `search` with a line that crosses a from x to y to find all the lines that extend from x into y and into the polygons that are adjacent to y, as indicated by the broken lines in figure 6.6. The second call finds the lines that extend into the polygons that are adjacent to x, as shown by the solid lines in figure 6.6.

Up until this point, the variable *limit* is the limit on the number of lines found by *each*

Algorithm 9 The function `extendFrom`

```
function format: extendFrom($a, P$, $limit$)
input:
adjacency $a$
set of convex polygons $P$
an integer $limit$
output:
a set of lines line $L^c$ as an ordered set of adjacencies
1:  the length of a line is the number of previously uncrossed adjacencies crossed by the line
2:  let $p|q$ denote the adjacency between polygons $p$ and $q$
3:  let $x$ and $y$ be polygons such that $a = x|y$
4:  $L^1$,$L^2$,$L^s$ and $L^c$ denote sets of lines (ordered sets of adjacencies)
5:  $l_m$ is an integer
6:  $L^1 \leftarrow$ search($x, y, \{x|y\}, \emptyset, P, limit$)
7:  $L^2 \leftarrow$ search($y, x, \{y|x\}, \emptyset, P, limit$)
8:  $L^s \leftarrow L^1 \cup L^2$
9:  $l_m \leftarrow 2 \times limit / |L^s|$ //$a/b$ denotes integer division
10: for each $l \in L^s$ do
11:     let $l^r$ = reverse($l$) //$a$ is now the last adjacency in $l^r$
12:     if $x$ is the polygon that is an end of the chain formed by $l^r$ then
13:         $L^t \leftarrow$search($x, y, l^r, \emptyset, P, l_m$)
14:     else
15:         $L^t \leftarrow$search($y, x, l^r, \emptyset, P, l_m$)
16:     end if
17:     for each line $k \in L^t$ do
18:         $k.u \leftarrow |l|$ //$k.u$ denotes number of uncrossed adjacencies crossed by $l$ indicating the unextended length of $k$
19:         $k.t \leftarrow |k|$ //$k.t$ denotes number of uncrossed adjacencies crossed by $k$ indicating the total length of $k$
20:     end for
21:     $L^c \leftarrow L^c \cup L^t$
22: end for
23: return $L^c$
reverse($l$) returns a line with the reverse ordering of $l$
search($c, p, L, S, P, l$) is algorithm 8.
```

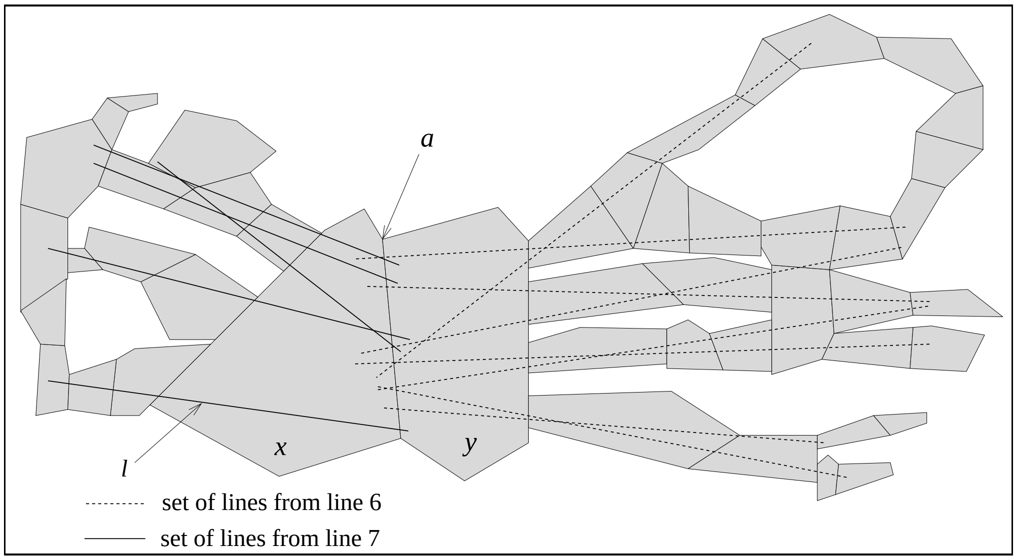


Figure 6.6: Results from lines 6 and 7 of function `extend`

call to `search`, meaning that there are, at most $2 \times limit$ lines in L^s. However, in the next part `extendFrom`, `search` is called on each line in L^s, which means that, if *limit* were to remain unchanged, then the calls to `search` after line 8 could find at most $4 \times limit^2$. Some flexibility has been given to the meaning of the variable *limit* but this would stretch it too far and would make the algorithm inefficient. At line 9, *limit* is changed to *limit* divided by half the number of lines in L^s. Since the size of L^s can't be bigger than twice the original limit, this ratio is always greater than or equal to 1. The new limit ensures that all lines found by all the calls to `search` cannot be more than the original limit.

The lines in L^s that do not cross any adjacencies are not needed since the lines returned from `search` are the lines that are going to be chosen by the heuristic and lines that only cross adjacencies that have already been crossed are useless. If these lines were removed before the main loop, it would lower the number of lines in L^s, making l_m higher so more lines can be found by subsequent calls to `search` giving a larger chance of finding a longer line. However, it has already been established that a is not crossed, but there may still be lines with a as the *only* uncrossed adjacency. This case can be dealt with by leaving one line that crosses a if no lines are left in L^s. However, this is not in the algorithm above and is not implemented for the empirical analysis.

Each line to be extended using `search` has to be reversed, otherwise `search` will try to extend the line forwards even more and find that it can't be extended and return the same line. The reversal happens at line 12 using the `reverse` function. In figure 6.7, line l starts at adjacency a and ends at adjacency b. The reverse function returns a line that starts at b and ends at a but still crosses the same adjacencies as l.

The **if** statement at line 12 determines whether line l was generated by extending from x to y or y to x to establish the order of the variables that are to be passed to `search`. If

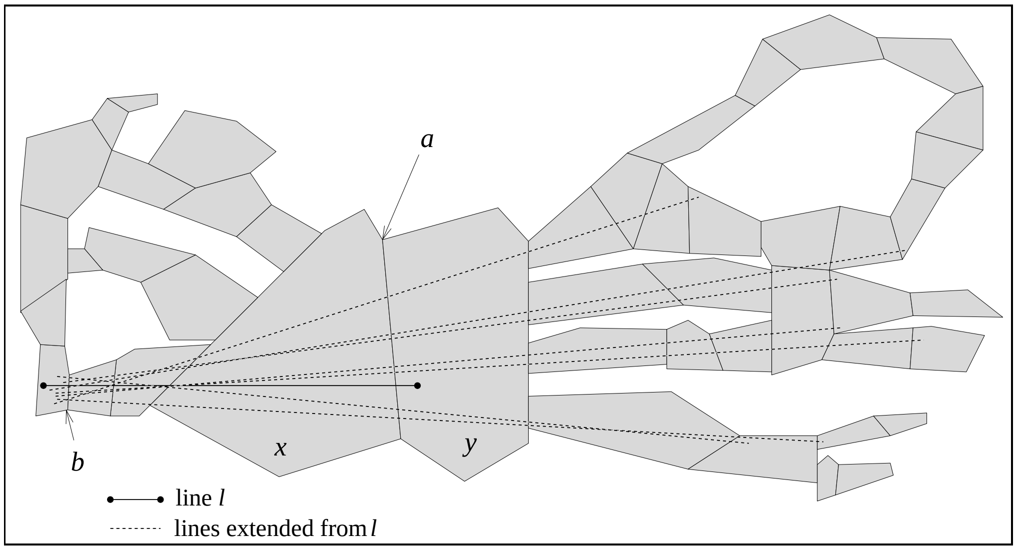


Figure 6.7: The result of one iteration of the main loop in `extendFrom` for the line indicated as l

this wasn't done then *search* may try to extend the line back on itself. Line l in figure 6.7 has an end in y so line 15 is executed. The `search` function at line 15 results in the broken lines shown in figure 6.7. If the number of broken lines shown in figure 6.7 were more than l_m then some would be missing.

The **for** loop at line 18 stores the unextended and the extended length of each extended line. The unextended length is always the number of uncrossed adjacencies crossed by l. In figure 6.6 the unextended length of l is 4 so the unextended length of all the broken lines is 4. The extended length for a line in L^t is the total number of uncrossed adjacencies crossed by that line.

The set of lines that are returned are all the lines that are found from all the calls to `search` in the main loop. There still may be lines that only cross a in the final set but leaving those in will not affect the final solution found by the line heuristic.

These two functions make up the basic heuristic by first using some starting point method to get an initial set of lines, I, then the `extendFrom` function can be applied to each of the uncrossed adjacencies in the polygons with adjacencies crossed by lines in I. Then the longest line can be chosen from the union of the sets returned by `extend-From` and placed into I. The process of applying the `extendFrom` function and picking the longest line is repeated until all the adjacencies are crossed. However, the following analysis shows that the `search` function is inefficient and a data structure is presented in section 6.5 that reduces the number of times that `search` gets called.

Analysis

The `extendFrom` function depends upon the `search` function, so `search` is analysed first. The `search` is shown to be $O(ln^2)$ where l and n are the limits on the number of lines and the number of adjacencies respectively. To analyse the `search` function, a bound will first be placed on the number of recursions. Following this, the complexity of each recursion will be found and multiplied by the number of recursions to get the final complexity. The analysis of `extendFrom` is relatively easy and is shown to be $O(ln^2)$.

Let p be the maximum number of adjacencies in any line. In each recursion of `search`, only one or zero adjacencies are added to *each* line that will be in the final solution because only the adjacencies in one polygon are being considered for addition, so the line coming into this polygon cannot cross more than two adjacencies in it because all polygons in the configuration are convex. One of the adjacencies has already been crossed by the line entering the polygon so there can only be one other adjacency added. The process of adding an adjacency is $O(p)$ because the visibility algorithm must be applied to check if it can be added and the visibility algorithm is linear. In addition to this, the only time that no adjacency is added to any line is when a line is added to the final set meaning that there are at most l of these instances. Moreover, no adjacency is added more than once to any line, so, there are only lp recursions of `search` where an adjacency is added to any line. This means that the number of recursions of `search` is no greater than $lp + l = l(p + 1)$ because of the stage that determines that a line cannot extend any further into the configuration. There can be fewer than $l(p + 1)$ recursions because there may be lines that are not p long and different adjacencies can be added to different lines within one recursion of `search`.

The complexity of each recursion of `search` is dominated by the calls to `canAddTo` which uses the $O(p)$ partial visibility algorithm and the number of calls to `canAddTo` is the number of polygons that are adjacent to polygon c besides p. Recall that c and p are parameters passed to `search`. Now, the number of adjacencies is in the order of the number of polygons, meaning that on average the number of polygons adjacent to each polygon is some constant, say b. Therefore, the complexity of a single recursion of `search` is $O(p)$. Taking this and the fact that there can be no more than $l(p + 1)$ recursions of `search` into account leads to the conclusion that the `search` function is $O(lp^2)$. What remains to be done is to place a bound on p.

Recall that p is the longest a line could be. Now consider the fact that a line cannot cross the same adjacency twice. This means that any axial line cannot cross more than n adjacencies so $p \leq n$. Therefore the complexity of the `search` function in terms of l and n is $O(ln^2)$.

Note that $O(ln^2)$ is a rather liberal upper bound. One reason for this is that most of the lines probably won't be n long. If a line was n long then all the adjacencies would be crossed and only this line would be returned. In fact, the longer the longest line the greater the likelihood that a small number of lines would be returned. More generally, the number of lines returned becomes the limit and would replace l in $O(ln^2)$, so if a limit on the number of lines for a specific configuration can be proven then that would be l instead of a user specified limit and it could be proved that `search` is more efficient for that case.

The analysis of `extendFrom` is relatively simple. The first two calls to `search` are $O(ln^2)$. The rest is constant time except for the main loop at line 10. Each iteration of this loop calls `reverse` which is $O(l)$. Then `search` is called with the limit that is set at line 9, which is equal to $2l$ divided by the number of lines in L^s so the calls to `search` at lines 13 and 15 are $O(\frac{2l}{|L^s|}n^2)$. Each iteration of the nested loop at line 17 executes in constant time and there are the same number of iterations as there are lines in L^t, which is bounded by $\frac{2l}{|L^s|}$. This means that the nested loop is $O(\frac{2l}{|L^s|})$ making the complexity of one iteration of the for loop $O(\frac{2l}{|L^s|} + \frac{2l}{|L^s|} + n)$ which is $O(\frac{2l}{|L^s|}n^2)$ because $\frac{2l}{|L^s|} \geq 1$ and $n \geq 1$. Now, the number of iterations of the loop is $|L^s|$ meaning that the whole of the main loop is $O(|L^s| \times \frac{2l}{|L^s|}n^2)$ which is $O(ln^2)$, since $|L^s|$ is bounded by l.

In summary, the first two calls are $O(ln^2)$ and the main loop is also $O(ln^2)$. Everything else is linear or less so `extendFrom` is $O(ln^2)$.

6.5 The line collections data structure

The `extendFrom` function is shown to be $O(ln^2)$ where l is the limit on the number of lines. Values of l greater than n^2 are not considered in this research, which means that `extendFrom` is $O(n^4)$ within the bounds of this research. An algorithm that is $O(n^4)$ is not very efficient and it would be desirable to reduce the number of times that it is executed. This section presents a data structure to store lines found by `extendFrom` so they don't have to be found again. When inserted, each line is sorted by length using a bucket sort and the function that retrieves a line returns the longest line. The insertion and retrieval functions are `insert` and `removeLongestLine` respectively.

The data structure consists of a two dimensional, $n \times n$ array and two numbers, *biggestU* and *biggestT*. The element at $[u, t]$ contain lines that have an unextended length of u and a total length of t, where the unextended length is the length of the line before it was extended backwards and the total length is the length of the line after it was extended backwards as set at lines 18 and 19 of algorithm 9. Remember that length is determined by the number of uncrossed adjacencies that a line crosses. The element [*biggestU*,*biggestT*] denotes the list of lines with the greatest unextended length with ties broken on the total length.

Note that half of the array would be empty because no line can have an unextended length greater than its total length. For example if a line has an uncrossed length of 4 then it can't be at position $[4, 2]$ and the second value would have to be greater than or equal to 4. Also, the elements in $[0, x]$ and $[x, 0]$ should be empty for all values of x because these lines only cross crossed adjacencies and are redundant.

The unextended length and the total length of each line are assigned by `extendFrom` so `insert`(L) can put each line, l, in L at the head of the linked list at position $[l.u, l.t]$ in constant time. If the unextended length of the new line is greater than *biggestU* then *biggestU* and *biggestT* are set to the lengths of the new line. If the unextended length of the new line is equal to *biggestU* but *biggestT* is greater than the total length then *biggestT* is set to the lines total length.

The `removeLongestLine` function returns the line at the head of the linked list at position [*biggestU*,*biggestT*]. This line is removed from the linked list and if the list is empty then *biggestU* and *biggestT* must be updated by finding the largest value of t such that $[biggestU, t]$ is non empty. If there is no such value of t then decrement *biggestU* until there is. In the worst case this process is $O(n^2)$ because each element in the array may have to be checked. It is possible to store the positions of each non empty element of the array in a balanced tree, which would make insertion $O(log(n))$ and retrieval $O(log(n))$, but this isn't implemented and the largest overhead comes from updating the length of the lines when they change.

When a line is chosen to be in the set of lines that will be returned by the line heuristic, the uncrossed length of some lines change so they have to be updated. The updates can all be done by `removeLongestLine` because a line is removed from the data structure only when that line will be inserted into the set returned by the heuristic. If each adjacency that is crossed by this line stores all the lines that cross it then it is easy to update the lengths. This requires the `insert` function to store a pointer to each line being inserted in each *uncrossed* adjacency crossed by it. It is also necessary to keep track of whether the adjacency being crossed is part of the unextended part of the line. The lines are not stored in crossed adjacencies because an adjacency will not become uncrossed if it is crossed already, so there is no need to update it.

When a line is updated, the line has to be moved within the data structure. Unfortunately, this means that the linked list at the position in the array has to be traversed, but this problem can be solved by storing the line's position in the linked list in the line itself. When a line is inserted or removed the neighbouring elements in the list are updated appropriately.

Analysis

When a line is inserted, the line is added to each uncrossed adjacency that it crosses. However, the number of adjacencies in each line is bounded by the number of adjacencies in the configuration. This shows that insertion of one line is $O(n)$. Therefore `insert` is $O(kn)$ where k is the number of lines in the set being inserted.

When a line is removed, each of the uncrossed adjacencies that that line crosses is operated upon. It has been established that the length of the line is bounded by n. Now each operation decrements the unextended length if this is the unextended portion and the total length of each line crossing this adjacency. `extendFrom` can't get called more than n times so there are at most nl lines in the configuration and in the extremely unlikely case that each of these lines cross all the adjacencies that the line crosses then this operation is $O(ln^2)$. This the same complexity as the `extendFrom` function but the `extendFrom` function would get called many times where the removal would only be called once.

6.6 The line heuristic - bringing it all together

This section explains how the line heuristic works using the starting point methods, the line collections data structure and the functions described in this chapter. First, the use of the starting point methods is discussed then the iterative part of the algorithm is given in the form of algorithm 11. An example is given in section 6.7 to clarify the workings of the algorithm.

6.6.1 Use of the starting point methods

The steps of the heuristic described in section 6.2 are as follows.

1. Find a starting point. This could be either a single polygon or a set of lines.
2. Create an empty set F.
3. Let A be all the uncrossed adjacencies that are part of polygons that have *crossed* adjacencies. If there are no lines (so no adjacencies are crossed) then use the adjacencies from the starting polygon.
4. Let L^s be the set of all the lines that cross all of the adjacencies in A.
5. Find the line from L^s that extends the furthest from the adjacency from A that it crosses and add it to F.
6. Repeat steps 3 to 5 until all adjacencies have been crossed.
7. Return F.

The first step requires a set of lines or a single polygon. The network method returns a set of lines that are placed in F. The random and least adjacency methods return a single polygon. However, the iterative part described in section 6.6.2 requires a set of lines be present in the data structure and that some adjacencies have been marked as “origins”. The reasons why these are required will become clearer in section 6.6.2. Algorithm 10 gives a method to initialise the data structure.

Algorithm 10 puts lines that extend the furthest from the chosen starting point into the line collections data structure, and initialises the set that will form the solution, so it does steps 2 to 4 of the heuristic above.

Figure 6.8 demonstrates the three starting point methods. The “origin” adjacencies, shown as broken lines in the figure, are the adjacencies that form the set referred to in step 3 of the heuristic. The lines in the data structure extend from these adjacencies and are shown as dotted lines in figure 6.8. Placing the lines in the data structure achieves step 4 of the heuristic.

Algorithm 10 achieves steps 1 to 4, however, it only executes 3 and 4 once. The iterative part of the algorithm continues from step 5 then jumps back to step 3 then 4.

6.6.2 The rest of the line heuristic

Algorithm 11 is the iterative part of the line heuristic that uses the functions and the data structure described in the rest of the chapter. Once algorithm 10 has been executed, algorithm 11 is applied repeatedly until no adjacency is left uncrossed.

Algorithm 10 Populating the line collections data structure.

```
input:
a set of convex polygons $P$
an empty set of lines $F$
an integer limit
1: if the least adjacency method is used or the random method is used then
2:     $I \leftarrow$ the polygon produced by the least adjacency or random starting point methods
3: else
4:     $I \leftarrow$ the set of polygons crossed into by the lines produced by the network method
5:     $F \leftarrow$ the set of lines produced by the network method
6: end if
7: for each polygon $p \in I$ do
8:     for each uncrossed adjacency $a$ in $p$ do
9:         mark $a$ as an origin
10:        insert(extendFrom($a, P$,limit))
11:    end for
12: end for
extendFrom($a, P, l$) finds the set of lines that start at $a$ where no line in the set is a
subset of another. The function then extends these lines backwards to find the set of
all lines that contain these lines. The extended lines are returned. The details and an
explanation of $l$ are given in section 6.4.
insert($S$) inserts the set of lines, $S$ into the data structure presented in section 6.5
```

Line 1 chooses a line that extends the furthest away from the current set of lines, which is step 5 of the heuristic. Once step 5 is executed the heuristic returns to step 3. Steps 3 and 4 are executed in the main **for** loop which finds all the lines that extend from the newly found line and places them in a data structure to be used by the next application of the algorithm.

The **for** loop from lines 2 to 6 only considers the polygons crossed by the new line because new origin adjacencies can only be formed by those polygons. Step 3 of the heuristic says that A, which is the set of origins, is the set of all the uncrossed adjacencies that are part of polygons that have crossed adjacencies. The only new adjacencies added to this set will be some of those that are in the polygons crossed by the new line. This is clarified with an example in section 6.7

Step 4 says that the set L^s is all the lines that cross all of the adjacencies in A. This is accomplished by line 5 of algorithm 11.

Algorithm 11 is applied repeatedly until all adjacencies are crossed. This results in a set of axial lines. However, if the network method was used to find a starting point then some lines found by the network of stars algorithm may be redundant once the heuristic is complete because they were not extended into the rest of the configuration. So the final step is to remove all the redundant lines.

It is easy to see that an execution of the heuristic results in all of the adjacencies in the configuration being crossed because a new line that crosses some uncrossed adjacency is chosen at each iteration of algorithm 11 and added to the solution set F and the heuristic only terminates once all adjacencies are crossed.

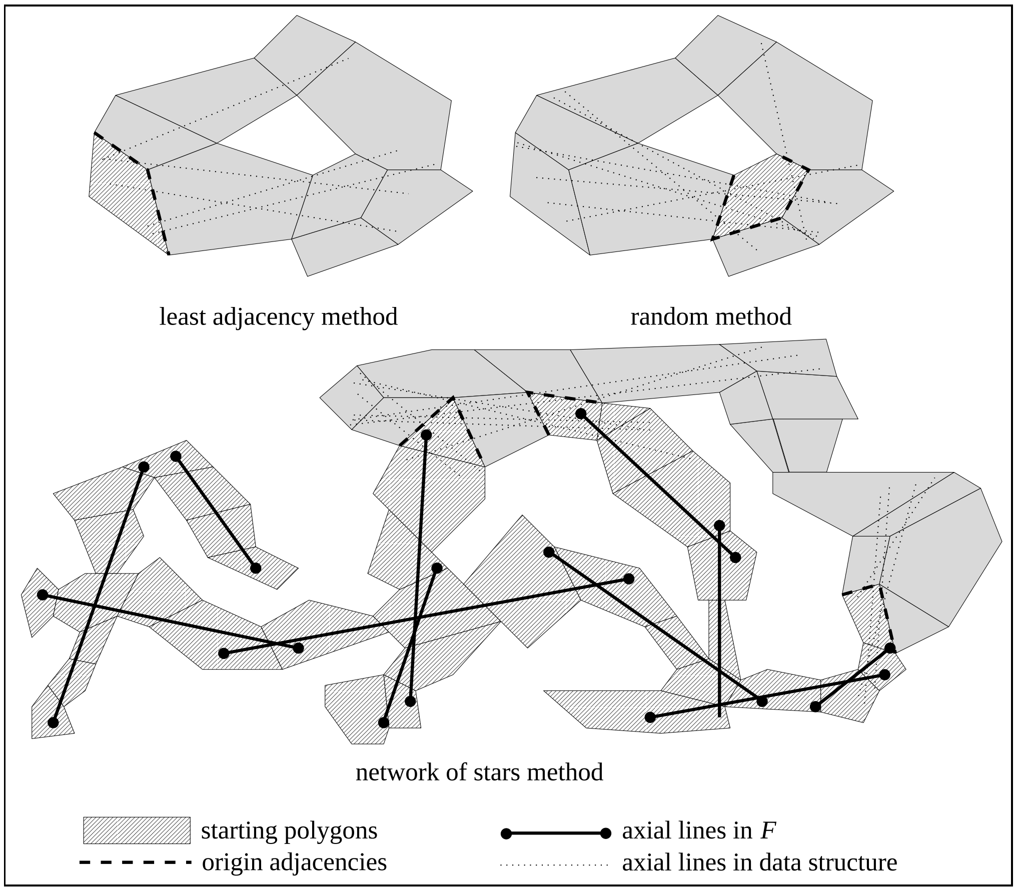


Figure 6.8: A demonstration of the three starting point methods.

Algorithm 11 is executed no more than n times, if n is the number of adjacencies, because one or more adjacencies are crossed at each application. Retrieving a line from the data structure has been shown to be an $O(ln^2)$ operation and is only called once for each application of algorithm 11, so the complexity of all of the retrievals from the data structure for the whole heuristic is $O(ln^3)$. Furthermore, `extendFrom` is only applied to an adjacency that hasn't been marked as an origin and line 4 of algorithm 11 marks an adjacency as an origin right before `extendFrom` is applied to it, so `extendFrom` is applied to no more than once to each adjacency. This means that the complexity of all the calls to `extendFrom` over the whole execution of the heuristic is $O(ln^3)$. The starting point methods are less than $O(n^3)$ so the complexity of the heuristic is $O(ln^3)$. If the number of polygons is q then the complexity is $O(lq^3)$ since the number polygons is in the order of the number of adjacencies.

It is conjectured that the number of actual lines is no more than $O(n^2)$, if this is the case then the heuristic is $O(q^5)$. However, the results of empirical tests presented in chapter 7 with $l = n$ are similar to the results where n^2 was used as a value of l. Moreover, when the configuration is sparse, a value of $\log_2 n$ also gave acceptable results.

Algorithm 11 One step in the line heuristic.

```
input:
a set of convex polygons $P$
a non-empty set of lines $F$
an integer *limit*
1: longestLine ← removeLongestLine()
2: for each polygon $p$ crossed by longestLine do
3:     for each uncrossed adjacency $a$ in $p$ that is not an origin do
4:         mark $a$ as an origin
5:         insert(extendFrom($a$, $P$, limit))
6:     end for
7: end for
8: $F \leftarrow F \cup$ longestLine
```

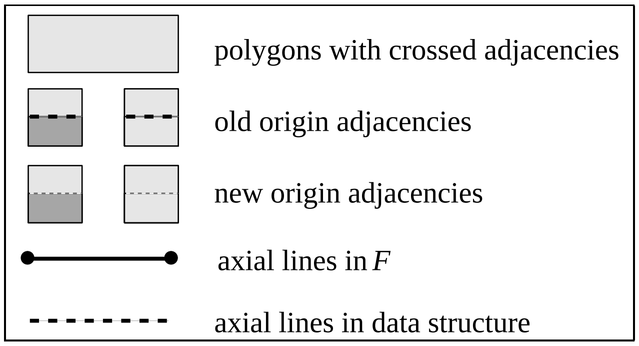
`extendFrom`(a, P, l) finds the set of lines that start at a where no line in the set is a subset of another. The function then extends these lines backwards to find the set of all lines that contain these lines. The extended lines are returned. The details and an explanation of l are given in section 6.4.

`insert`(S) inserts the set of lines, S, into the data structure presented in section 6.5

`removeLongestLine`() returns the line that extends the furthest from the set of uncrossed adjacencies whose polygons contain adjacencies that are crossed. Note that how far a line extends is measured by how many *uncrossed* adjacencies it crosses. This uses the data structure discussed in section 6.5

6.7 Example

The heuristic is explained below with the aid of an example. The variable *limit* is set to n^2 and the least adjacency method is used to find a starting point. Figure 6.9 is a key to the figures in this section.



polygons with crossed adjacencies
old origin adjacencies
new origin adjacencies
axial lines in F
axial lines in data structure

Figure 6.9: The key to the figures in this section.

Figure 6.10 shows the example configuration with algorithm 10 already applied. Polygon A is chosen as the starting point because it is the left most polygon with one adjacency. The only adjacency in this polygon is a so line 9 of algorithm 10 marks it as an origin and line 10 applies `extendFrom` to it. Line 1 is the only line in the set returned from `extendFrom` and is placed in the data structure. Note that, line 1 is not yet in the solution

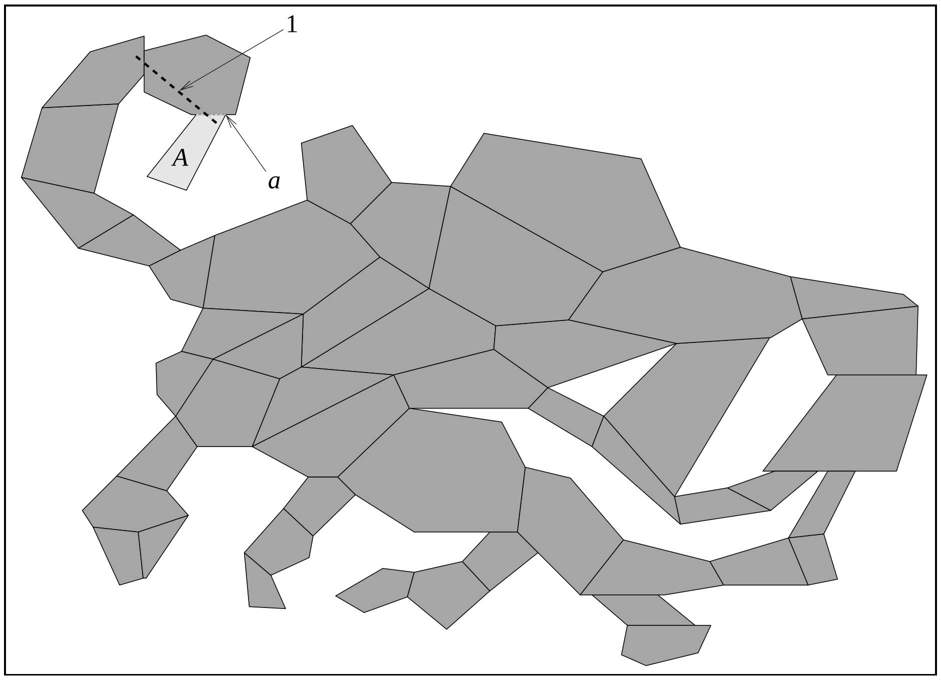


Figure 6.10: A configuration of convex polygons with algorithm 10 already applied. Line 1 is the only line in the data structure.

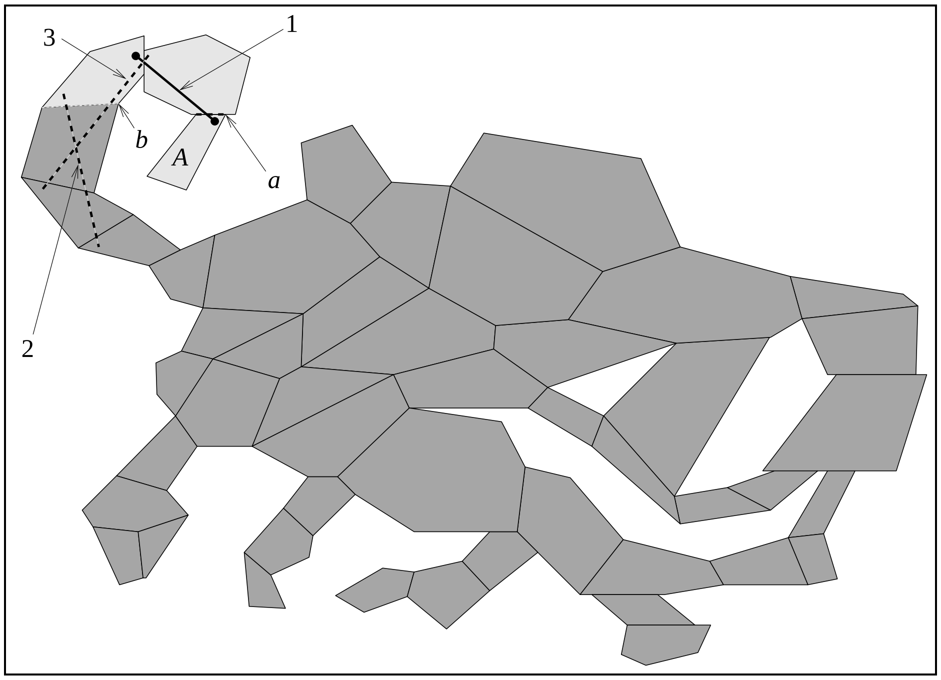


Figure 6.11: A configuration of convex polygons with algorithm 11 applied once. Line 1 is now in the solution set and lines 2 and 3 are in the data structure.

set, F, and will only be placed there on the first execution of algorithm 11.

Figure 6.11 shows the example configuration after the first application of algorithm 11. Line 1 is removed from the data structure at line 1 of algorithm 11 and placed in F at line 8. Adjacency b is marked as an origin and `extendFrom` is applied to it at lines 4 and 5 of algorithm 11. Lines 2 and 3 are returned by `extendFrom` and are now in the data structure. Note that line 2 should be added to F next because it crosses three uncrossed adjacencies, where line 3 only crosses two.

Two more applications of algorithm 11 results in the state shown in figure 6.12. In the second application, line 2 is removed from the data structure and placed in F and c is marked as an origin adjacency. Line 3 is removed from the data structure because its origin is b and any line that is extended from b is no longer a line that extends as far as it can from the current configuration of lines. Note that the lines 1 and 2 are the same lines that would be placed by the chain algorithm if it was applied to this part of the configuration if it started at A. In fact, line 1 is an essential line and line 2 would be in the minimal solution for the whole configuration.

Applying `extendFrom` to c produces a set of many lines that would cover most of the configuration, so they are not shown. However, line 4 is in this set and is the line that crosses the most uncrossed adjacencies. This means that it is removed from the data structure at line 1 of the third application of algorithm 11 and added to F at line 8.

Lines 2 to 7 of algorithm 11 apply `extendFrom` to all of the new origin adjacencies that are shown in figure 6.12. Clearly, there are many lines that extend from the new origin adjacencies so they are not shown either.

The fourth iteration of algorithm 11 is shown in figure 6.13. Line 5 is removed from the data structure and added to F. Now, only five adjacencies are marked as origins so the `extendFrom` function only has to be applied to these. The lines in the data structure that originate from other uncrossed origin adjacencies merely have to be updated because they are already in the data structure. Figure 6.14 shows the solution.

6.8 Conclusion

This chapter contains a heuristic for the problem of axial line placement in convex polygons which uses the special cases that are presented in this document for one of the three proposed methods for preprocessing. The next chapter gives the results from the empirical tests where one of these tests compared these three methods of preprocessing.

The major difference between the random and least adjacency methods and the network detection method is that the random and least adjacency methods attempt to find a starting point for the algorithm while minimising interference with the heuristic itself. The network method interferes with the heuristic by placing lines meaning that the line heuristic must work around this solution. This method works well if a large part of the configuration can be identified as a particular special case but may work badly if many small parts of the configuration are identified meaning that the line heuristic must work around these areas

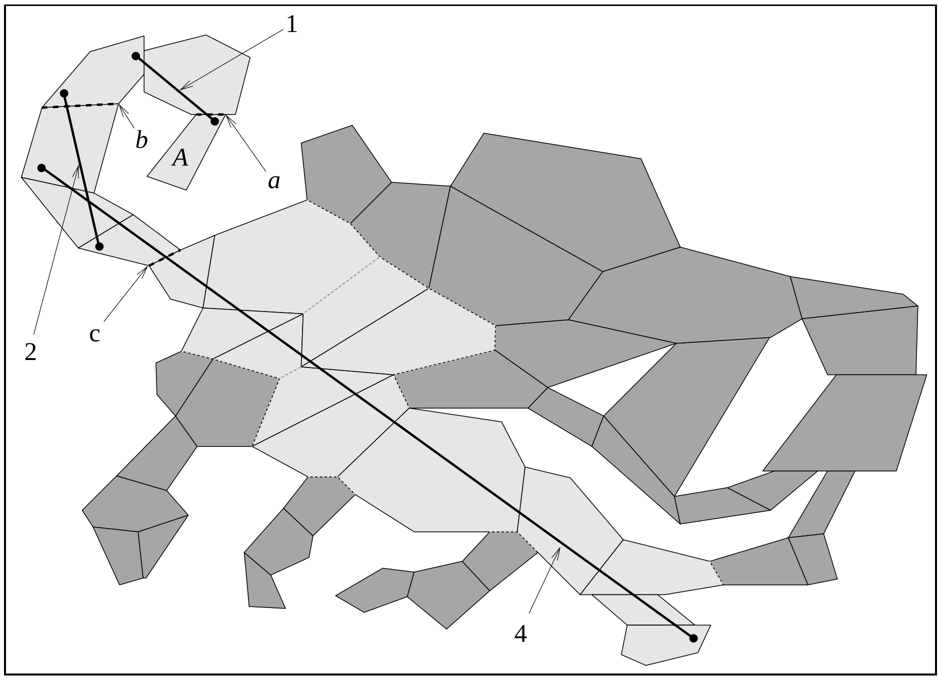


Figure 6.12: A configuration of convex polygons after three applications of algorithm 11. Lines 1, 2 and 4 are in F and there are many lines in the data structure which are not shown.

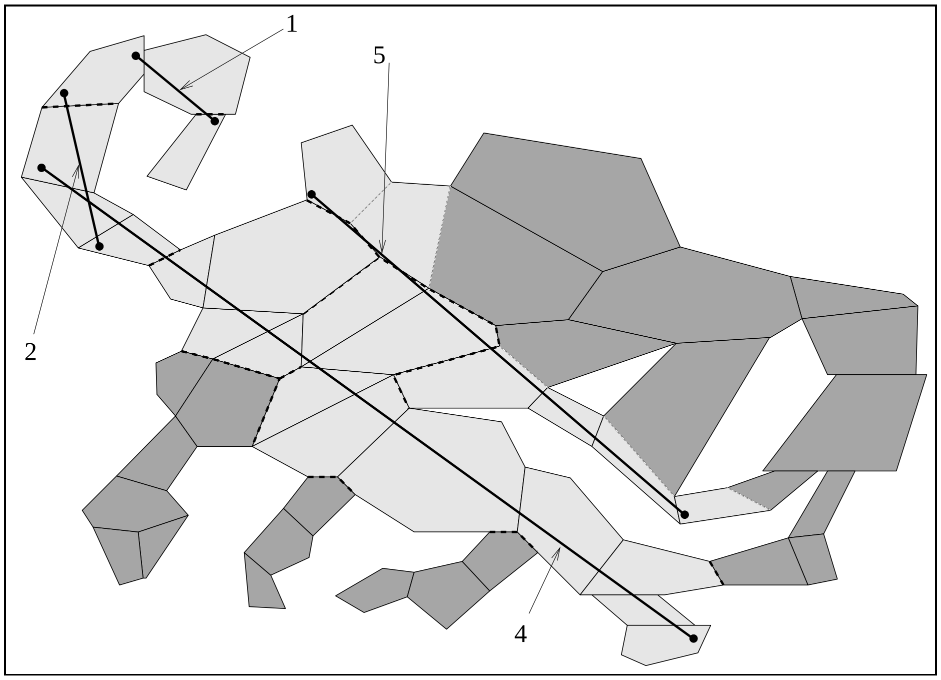


Figure 6.13: A configuration of convex polygons with algorithm 11 applied four times. Lines 1, 2, 4 and 5 are in F and there are many lines in the data structure which are not shown.

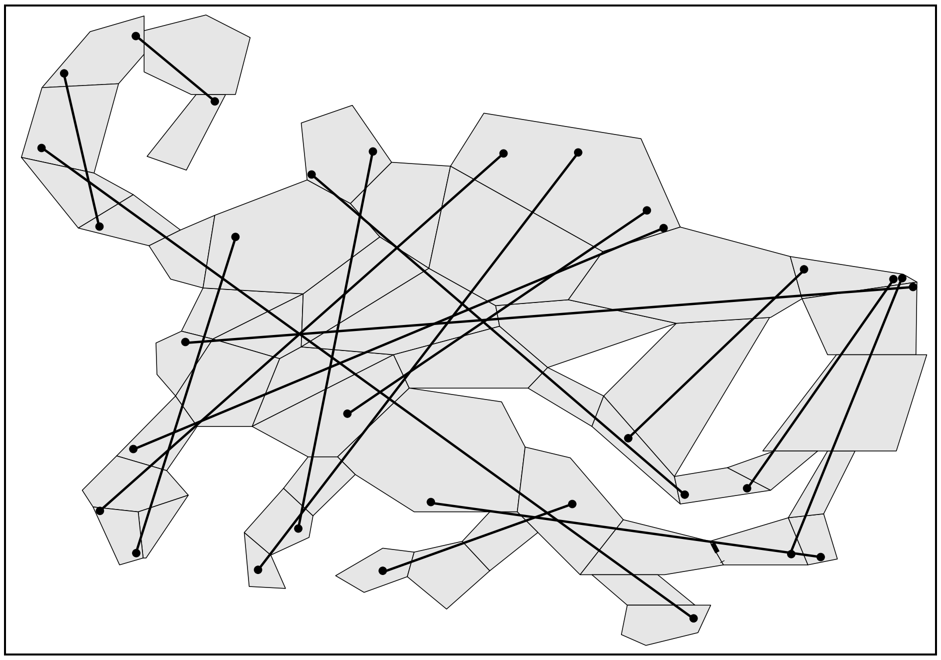


Figure 6.14: The solution found by the line heuristic.

resulting in wasted lines. The following chapter shows that this is a major problem with using special case detection for preprocessing and chapter 8 suggests methods for improving the integration of the line heuristic with special case detection.

The heuristic is shown to work in $O(lq^3)$ time where q is the number of polygons and l can be a function of q or the number of adjacencies. The bound on l is conjectured to be no more than $O(q^2)$ so the algorithm works in no more than $O(q^5)$ time. This may seem inefficient but the problem is NP-complete and, as the next chapter will show, the heuristic finds a good solution. The next chapter contains details of the empirical tests that were done using an implementation of this heuristic where, along with the preprocessing methods, various limits are compared and the solutions found by the line heuristic are compared with exact solutions. These tests justify the high complexity by showing just how well the line heuristic works.

Chapter 7

Empirical results

7.1 Introduction

To evaluate the line heuristic presented in chapter 6, an implementation of this heuristic was used on many configurations of randomly generated polygons. The results from these tests are presented in this chapter. The tests used are the following.

1. Test the three starting point methods from section 6.3.

2. Test the three limits passed to the `search` function from section 6.4.

3. Compare the solutions found by the line heuristic to the corresponding exact solutions.

Test 1 is used to find the best starting point method, which was then used in tests 2 and 3. Test 2 is used to evaluate the limits placed upon the number of lines found by `search` and to give some indication to the actual limit on the number lines that would be found by an unlimited `search` function. Test 3 is the most important test because it is used to determine how well the heuristic works. However, the configurations used in test 3 are small because of the time needed to compute exact solutions so the results are somewhat limited.

The following section briefly discusses the different types of configurations used in the three tests and how these configurations were generated. Following this, the results from each of the three tests are presented and evaluated in sections 7.4, 7.5 and 7.6. Section 7.4 contains an explanation of the tables that are used to present the results for the starting point method, which is analogous to the tables in sections 7.5 and 7.6.

7.2 Test case generation

This section gives the method used to generate two types of configurations which can be classified as “dense” and “sparse”. Dense configurations are configurations of tightly packed polygons, whereas sparse configurations would be “star like” in the sense that they contain many chains or thin rows of polygons that link other parts of the configuration. Examples of dense and sparse configurations generated using this method are given in figures 7.1 and 7.2.

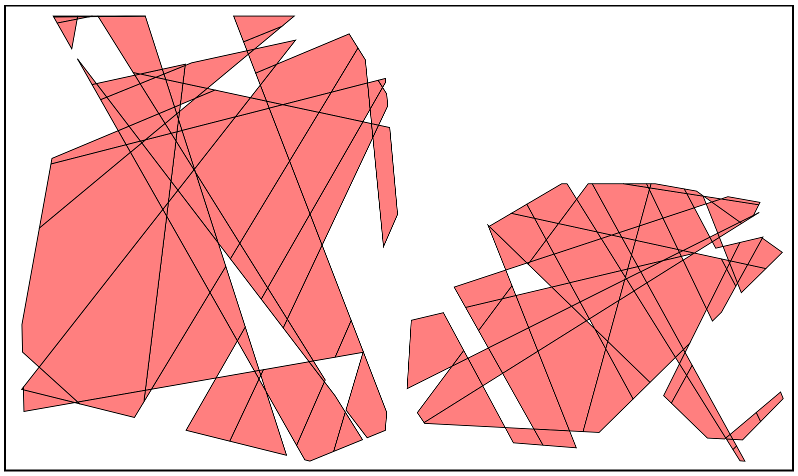


Figure 7.1: Two examples of dense configurations.

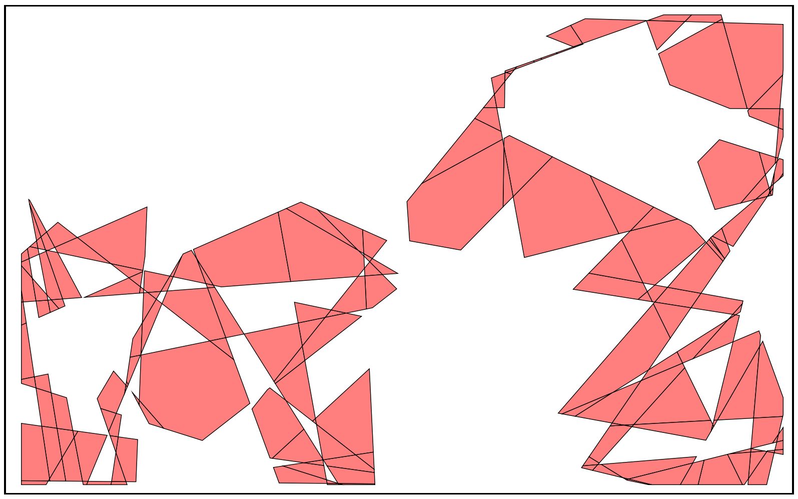


Figure 7.2: Two examples of sparse configurations

The method for generating both dense and sparse configurations begins by randomly drawing lines across a rectangle from side to side as in figure 7.3. Then all the polygons that are formed by the intersections of the lines are detected. Next, an initial polygon is chosen that is close to the center of the configuration. At this point the method starts to iterate and, at each iteration, polygons are chosen that are adjacent to polygons that have already been chosen. The policy for picking polygons at each stage depends on whether a dense or a sparse configuration is desired. The method stops when the specified number of polygons have been chosen.

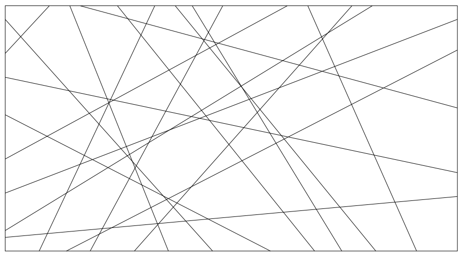


Figure 7.3: Rectangle with lines randomly drawn

The polygons found are limited because a lot of polygons are in straight lines. Some chains lack the desirable bends but this is somewhat compensated by the fact that the vertices are rounded off to integers so situations like 7.4 are possible.

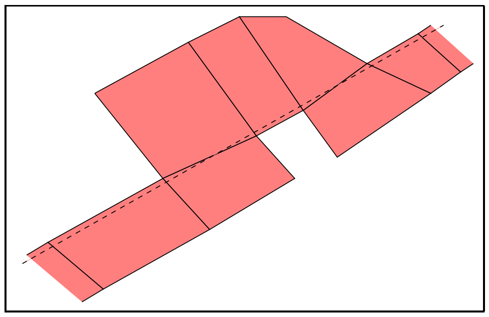


Figure 7.4: A bend that is possible. The broken line is an axial line and the shaded parts are part of the configuration

The configurations generated using this method were as follows.

- 900 dense configurations of 20 polygons each
- 900 sparse configurations of 20 polygons each
- 900 dense configurations of 100 polygons each
- 900 sparse configurations of 100 polygons each

The 20 polygon sets were used for the comparison between the exact solution and the solutions found by the heuristic. These 20 polygon sets were used because they are the largest sets where the exact solution can be determined in a reasonable amount of time.

The 100 polygon sets were used for the starting point tests and the limit tests. These 100 polygon sets were used because they were thought to be large enough to demonstrate the important aspects of the starting point methods. In addition to this, the high complexity of the heuristic for the general case of axial line placement meant that the test cases could not be very large.

7.3 Explanation of data in tables

This section contains an explanation of the tables that are presented in this section. There are two types of tables used in this chapter and they are:

- tables that show the average number of lines found by each method and
- tables that show the differences in the number of lines found by each method.

The first type of table is straightforward. However, the second type may require some explanation. An example of the second type of table is given as table 7.1.

	Sparse			Dense		
Condition	#	average diff	greatest diff	#	average diff	greatest diff
method A $>$ method B	100	1.00	1	200	1.50	3
method A $=$ method B	200	0	0	300	0	0
method A $<$ method B	700	3.00	5	500	5.00	7
Total		-2.00		-2.20		

Table 7.1: Example of a table showing the differences in method A and method B

The condition column in table 7.1 specifies which data the row pertains to. The tables compare the number of lines found by the various methods, and the condition indicates which methods are being compared in that row, and what the relation between the solutions found was. For example, a condition labelled “Method A $<$ Method B” indicates that the configurations in that row conform to the condition that method A yields fewer lines than method B.

The first row of the first and fourth columns in table 7.1 shows how often the number of lines found by the specified method was greater than the number of lines found by another specified method. The second and third columns operate similarly, showing how often the number of lines found by the method is respectively equal to or less than the number of lines found by the other method. For example, the element at the first row of the first column of table 7.1 indicates that the number of lines found by the method A was greater than the number of lines found by the method B 100 times.

The first row of the second and fifth columns of table 7.1 gives an indication of the scale of the difference between the solutions found by the methods. It does this by presenting the average difference between the number of lines found by the methods when the specified condition holds. For example, the number at the first row of the second column of table 7.1 indicates that the average difference between the number of lines found by method A and method B was 1.00 *when* method A found more lines than method B. Note that the average differences in rows one and three cannot be less than 1 because the number of lines is a whole number, and the cases where the difference is equal to 0 would be included in row two.

The first row of the third and sixth columns of table 7.1 show the greatest, positive difference between the number of lines found by method A and the number of lines found by method B. Similarly, the first row of the third and sixth columns of table 7.1 show the greatest, positive difference between the number of lines found by method B and the number of lines found by method A. Note that the greatest element should not be greater than 1 if the corresponding average is equal to one, but this can happen because of round off errors while calculating the average.

The fourth row in tables 7.1 show the average difference between the number of lines found by the specified method and the other specified method for *all* the test cases. For example, the element at row four, column two in table 7.1 indicates that the average difference between the number of lines found by method A and method B was -2.00. This indicates that method B found more lines in the sparse configurations than method A did, suggesting that method A works better in sparse configurations than method B does.

7.4 Testing the starting point methods

These are the three starting point methods that were tested.

1. Pick a random polygon as the start set.
2. Pick the polygon with the least number of adjacencies that has the adjacency with smallest the x coordinate.
3. Detect any networks of stars embedded in the configuration then use the network of stars heuristic from chapter 5 on the networks.

Section 6.3 says that method 3 was expected to be the best and method 1 would be the worst. However, the results presented in this section show there is little difference between methods 1 and 2 and both are better than method 3. Possible explanations are given in this section.

To test the three starting points discussed in section 6.3 each starting point was run on 900 sparse and 900 dense configurations of 100 convex polygons each.

	Sparse	Dense
Random	20.80	20.34
Least Adj.	20.83	20.33
Networks	22.08	22.15

Table 7.2: Average lines found for each starting point method in configurations of 100 polygons each.

The average number of lines found by each method in the sparse and the dense configurations is shown in table 7.2. This table already suggests that the random and least adjacency methods are superior to the network method. Tables 7.3 and 7.4 and 7.5 compare the differences in the number of lines found by each method as explained in section 7.3. The following section compares the random method with the least adjacency method.

7.4.1 Random versus least adjacency

The data comparing the random method with the least adjacency method is shown in table 7.3. These results were generated using the line heuristic with $l = n^2$ as the limit passed to the `search` function. The data shows that the number of times the random method found more lines than the least adjacency method is very close to the number of times that the least adjacency method found more for both dense and sparse cases. In fact, both methods found the *same* number of lines in a large proportion of the test cases. Furthermore, the average difference in both cases is between 1 and 1.3, indicating that most of the differences were 1. This, coupled with the fact that the *total* average differences are very close to 0, strongly suggests that there is very little advantage to using either the random method or the least adjacency method.

What is interesting about these results is the difference between the sparse and dense configurations. The methods found the same number of lines much more often in sparse configurations than in dense configurations. This indicates that a dense configuration is more sensitive to the choice of initial polygon than a sparse configuration is. However, the test was not designed to test this and will not be discussed further.

	Sparse			Dense		
Condition	#	average diff	greatest diff	#	average diff	greatest diff
Random $>$ Least Adj.	205	1.16	4	275	1.26	3
Random $=$ Least Adj.	463	0	0	342	0	0
Random $<$ Least Adj.	232	1.16	3	283	1.22	3
Total		-0.03			0.0001	

Table 7.3: Results from starting point tests for 900 configurations of 100 polygons each for the random method versus the least adjacency method.

7.4.2 Networks versus random and least adjacency

The network method was expected to be the best performing method but the results in this section show otherwise. Since the random and least adjacency methods are shown to be of similar value above, the networks method is compared to both of them here.

	Sparse			Dense		
Condition	#	average diff	greatest diff	#	average diff	greatest diff
Random $>$ Networks	99	1.26	3	63	1.30	3
Random $=$ Networks	176	0	0	153	0	0
Random $<$ Networks	625	2.05	7	684	2.51	10
Total		-1.28			-1.81	

Table 7.4: Results from starting point tests for 900 configurations of 100 polygons each for the random method versus the network method.

	Sparse			Dense		
Condition	#	average diff	greatest diff	#	average diff	greatest diff
Least Adj. > Networks	99	1.28	3	49	1.08	2
Least Adj. = Networks	193	0	0	188	0	0
Least Adj. < Networks	608	2.06	7	633	2.54	8
Total		-1.25			-1.82	

Table 7.5: Results from starting point tests for 900 configurations of 100 polygons each for the least adjacency method versus the network method.

Tables 7.4 and 7.5 show that the network method found more lines than the least adjacency and random methods for more than two thirds of all of the configurations. The configuration shown in figure 7.5 will be investigated to explain why this happens. The darkly shaded polygons are polygons that are detected as networks of stars. The configuration in figure 7.5 is a dense configuration where the difference in the number of lines found by the network and the least adjacency methods was the greatest. In this case the network method found 26 lines and the least adjacency method found 18. So the difference is 8, as is indicated in table 7.5.

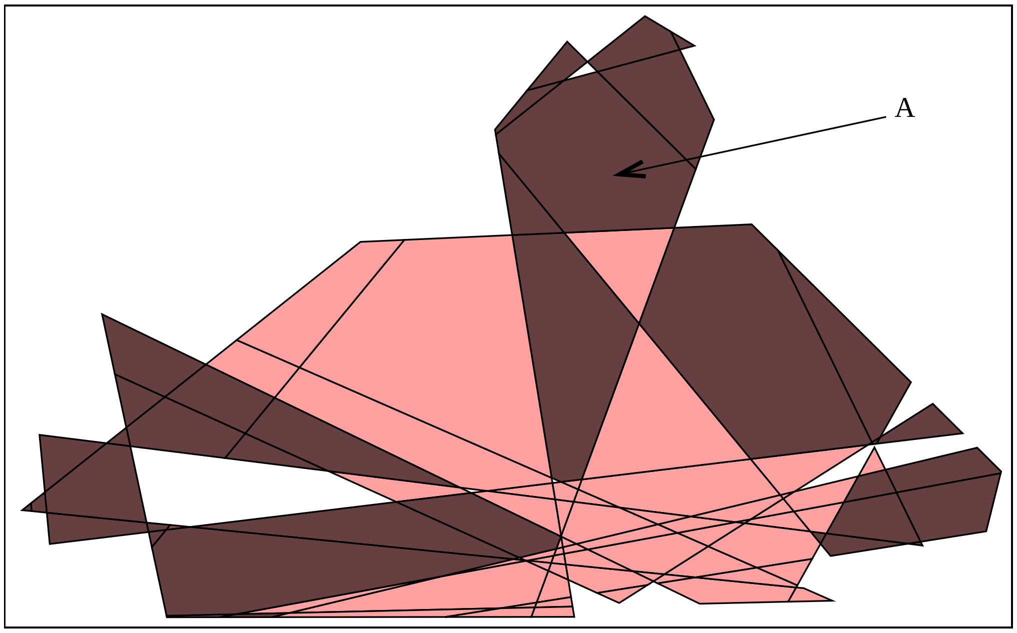


Figure 7.5: Example of a dense configuration where the network method worked badly.

The problem with the case in figure 7.5 is that many of the stars detected are too "open", in the sense that many of the adjacencies in the star are visible to the same parts of the configuration in many different ways. The star marked as A in the example is a good example of this. *All* of the adjacencies in the polygon are visible to the large cluster of small polygons at the bottom of the configuration. Many lines can come from the cluster at the bottom and cross the adjacencies in the star but don't because they have been crossed already meaning that the lines go in other directions with more uncrossed adjacencies.

Figure 7.6 shows some of the lines that the least adjacency method placed and figure 7.7 shows some of the lines that the network method placed. Some wavy axial lines are used because some pass through narrow parts or may seem to be co-linear with some adjacencies. The network method placed a line that crossed five adjacencies at the top of the configuration, marked a. All of these adjacencies were crossed by other lines placed by the least adjacency method (shown in figure 7.6) and these lines crossed many other adjacencies. Line f in figure 7.6 crosses many other adjacencies that are part of the star to the left but it was not found by the network method because most of those adjacencies were crossed by other lines placed by the network algorithm and so that line would be considered too short by the heuristic. This suggests that the network of stars heuristic may not be compatible with the line heuristic for dense configurations of polygons.

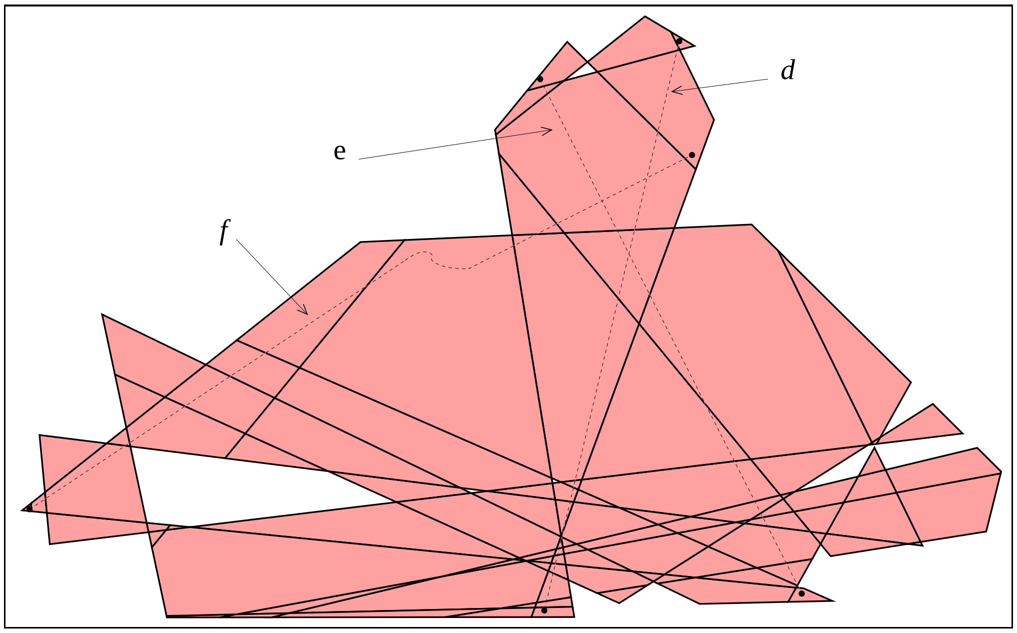


Figure 7.6: Some lines placed by the least adjacency method.

Another point about the network method is also shown in figure 7.7. The two adjacencies crossed by line b in figure 7.7 are not crossed again by the line heuristic meaning that it does not get thrown away. Figure 7.6 illustrates that this line could have been extended to cross many more adjacencies. This problem can be fixed by extending all of the lines using the `search` function from section 6.4 on all of the lines found by the network of stars algorithm.

A sparse configuration where the network method worked badly is shown in figure 7.8. The polygons that were detected as parts of networks of stars are the darkly shaded polygons. Figure 7.8 shows that a large proportion of the configuration is considered parts of networks of stars but the heuristic implementing the network of stars heuristic found

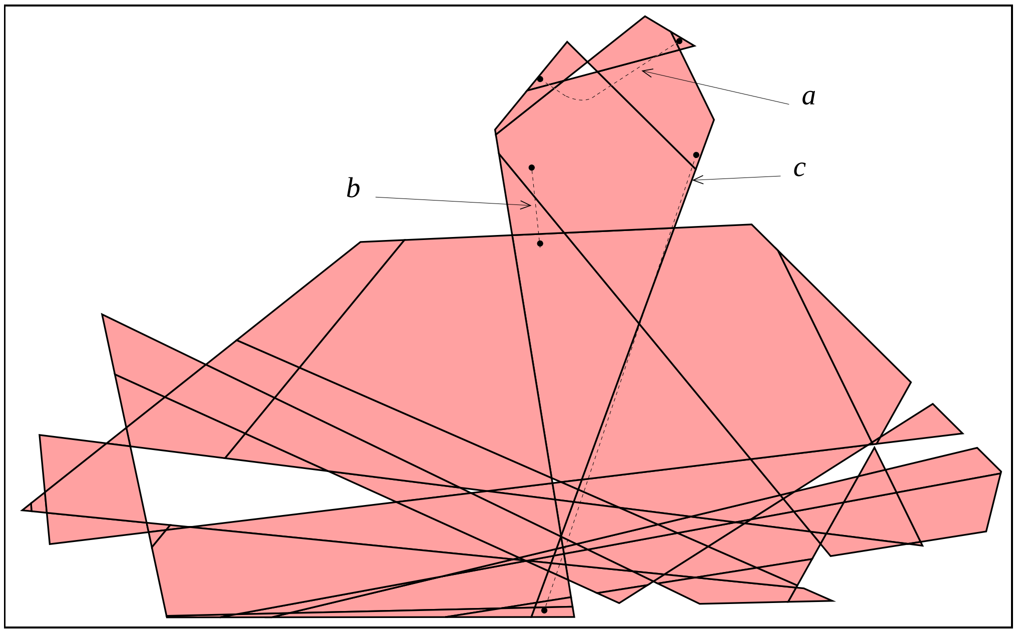


Figure 7.7: Some lines placed by the network method.

27 lines, where as the least adjacency method found 20. This indicates that the network heuristic is bad or it isn't compatible with the line heuristic. To test this, the polygons that are not part of the network of stars were removed from the configuration in figure 7.8 and both methods were tested. Both methods found 15 lines so the network of stars heuristic is not to blame for the difference, so the network heuristic may not be compatible with the line heuristic for sparse cases either. This conclusion is supported by the rest of the results.

One of the cases where the network method worked well is shown in figure 7.9. The network method produced 21 lines and the least adjacency method produced 24. In this case most of the configuration is considered a network of stars, much like the configuration in figure 7.8. However, this configuration differs because there are not so many spaces between stars that are not part of a network. This configuration seems to indicate that the network heuristic works well when applied to configurations that are only networks but is not compatible with general configurations. However, no conclusions can be drawn from this due to the small number of cases where the network did work well and there are no tests done in this research to test the network heuristic directly.

This section has shown that the random and the least adjacency methods are tied for the best starting point method so far, but the least adjacency method is deterministic so that will be the method used in any further experiments. It is believed that the network method can be improved but that is left for future work.

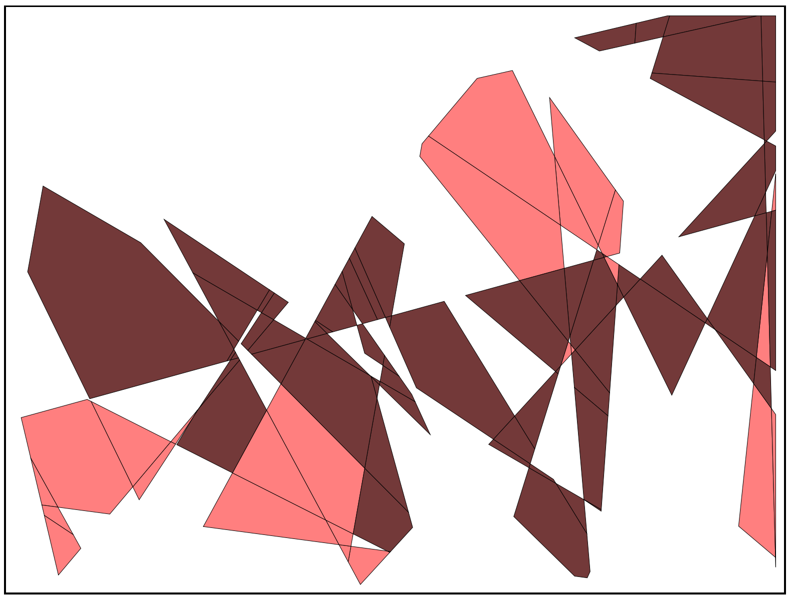


Figure 7.8: A configuration where the network method worked badly.

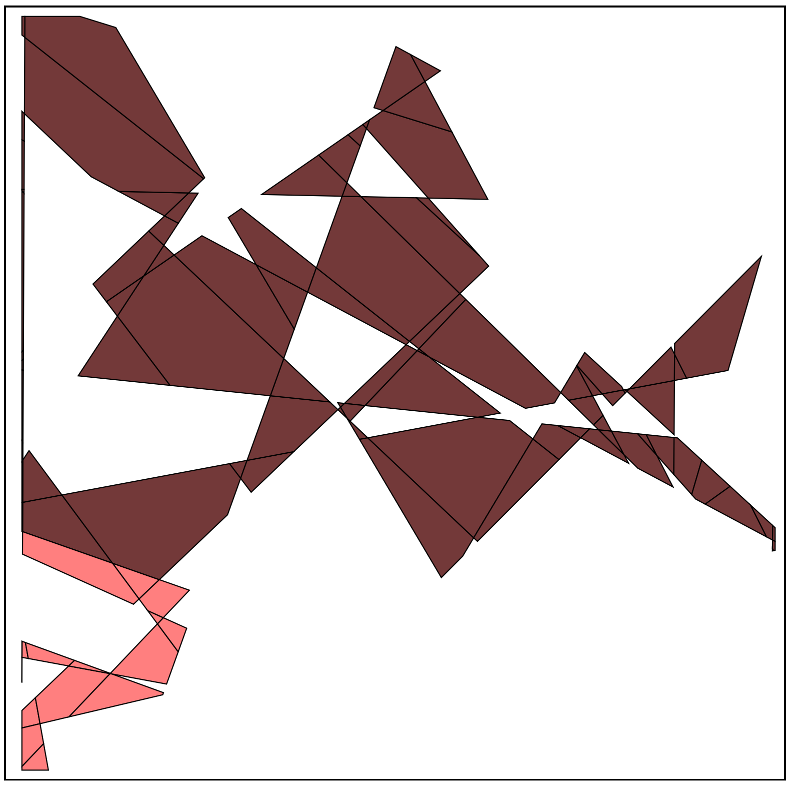


Figure 7.9: A configuration where the network method worked well.

7.5 Testing the limits on longest line generation

The results in this section were generated using three variations of an implementation of the line heuristic on 900 dense and 900 sparse configurations of convex polygons using the least adjacency method to find a starting point. The difference between each variation is the limits passed to the `search` function. These limits were $\log_2 n$, n and n^2 where n is the number of adjacencies in a configuration. The reason for making the limit a function of n is so the results can easily relate to the size of the configuration. Furthermore, the limit was made in terms of the number of adjacencies instead of the number of polygons because it is conjectured that the limit is in terms of the number of adjacencies more than that in terms of the number of polygons. However, the number of adjacencies is in the order of the number of polygons.

The results in the section show how the limit affects the number of lines in a configuration and shows the value of choosing the longest possible lines to cross adjacencies. More specifically, the results show that quadratic and linear limits produce similar results but logarithmic limits produce a much greater number of lines for dense cases. However, the results do not show how the difference in the number of lines grows as the number of polygons increases for the various limits. Results that show how often the `search` function reaches the specified limit are also shown.

limit	Sparse	Dense
$\log_2 n$	20.84	22.07
n	20.82	19.72
n^2	20.82	19.67

Table 7.6: Average lines found for each value of *limit* in 900 configurations of 100 polygons each.

The average number of lines found with each limit for each type of configuration is shown in table 7.6. This table shows that the difference in the number of lines found in sparse configurations for each of the three limits is so small that it can be ignored. However, the difference becomes apparent with the dense configurations. Tables 7.7, 7.10 and 7.9 go further to show this difference.

Table 7.8 shows the number of times the `search` function reached the specified limit as a percentage of the number of times `search` was called. The limit was divided at certain points in the `search` function, so the first and second columns show the limit data for the unchanged limit and the third and fourth columns show the data for the divided limit.

7.5.1 Limits and sparse confi gurations

Table 7.7 shows that there is no difference in the number of lines found in sparse cases using either the quadratic or linear limits. This indicates that there are relatively few lines from any adjacency in a sparse configuration. This is supported by table 7.8 which shows that the

	Sparse			Dense		
Condition	#	average diff	greatest diff	#	average diff	greatest diff
$n > n^2$	0	n/a	n/a	98	1.09	2
$n = n^2$	900	0	0	744	0	0
$n < n^2$	0	n/a	n/a	58	1.16	2
Total		0			0.05	

Table 7.7: Results from limit tests with limits equal to n and n^2.

limit	undivided			divided	
	sparse	dense		sparse	dense
$\log_2 n$	10.01%	82.19%		47.00%	99.65%
n	0.00%	3.38%		0.01%	36.28%
n^2	0.00%	0.00%		0.00%	0.01%

Table 7.8: Average number of times each limit was reached with each type of configuration expressed as a percentage.

limit was *never* reached for the undivided limit and only reached the divided limit 0.01% of the time for 900 configurations. The fact that the heuristic hardly ever reached either limit means that the heuristic was picking lines from the same sets, which explains why the heuristic with quadratic and linear limits found the same number of lines.

For the logarithmic limits, tables 7.9 and 7.10 show that the heuristic with the logarithmic limit found the same number of lines as the heuristic with the other limits 846 times out of 900 for sparse configurations. Furthermore, the instances where there were differences in the number of lines, the differences were small. The biggest differences in the lines were 2 in either direction and the total average difference was 0.02, which is inconsequential. Also, table 7.8 shows that the undivided, logarithmic limit was reached only 10.01% of the time. Therefore, there is very little difference between the logarithmic limit and the other two for sparse configurations.

Taking the above into account means that there is very little difference between any of the limits in sparse configurations. However, the fact that the undivided limits were seldom reached means that the time difference would be small for each of the limits. Unfortunately, no timing data was taken so this theory will not be empirically verified but it is easy to see this is so because the limit is part of the complexity function. However, these results do show that the algorithm will work relatively efficiently for sparse configurations.

7.5.2 Limits and dense configurations

Changing the limit seems to have very little effect on solutions when dealing with sparse configurations but the results show otherwise when dense configurations are considered.

	Sparse			Dense		
Condition	#	average diff	greatest diff	#	average diff	greatest diff
$\log_2 n > n$	33	1.09	2	807	2.65	7
$\log_2 n = n$	846	0	0	70	0	0
$\log_2 n < n$	21	1.05	2	23	1.13	2
Total		0.02			2.35	

Table 7.9: Results from limit tests with limits equal to $\log_2 n$ and n.

	Sparse			Dense		
Condition	#	average diff	greatest diff	#	average diff	greatest diff
$\log_2 n > n^2$	33	1.09	2	815	2.68	7
$\log_2 n = n^2$	846	0	0	62	0	0
$\log_2 n < n^2$	21	1.05	2	23	1.13	2
Total		0.02			2.40	

Table 7.10: Results from limit tests with limits equal to $\log_2 n$ and n^2.

Tables 7.9 and 7.10 show that there is a considerable difference between solutions found using the logarithmic limits and solutions found using the other limits for dense configurations. The heuristic using the logarithmic limit found more lines than the heuristic using the other limits more than 800 times each, and the average differences were 2.35 and 2.40. The average number of lines found using any limit was between 19 and 22 so differences of 2.35 and 2.40 are considered meaningful. Table 7.8 reached shows that the undivided limit was reached 82.19% of the time, which accounts for this great discrepancy.

The differences between the linear and the quadratic limits are not so large. Table 7.7 shows that both limits produced the same number of lines 744 times out of 900 for dense configurations and the average difference was an inconsequential 0.05. Table 7.8 shows that the undivided limit was reached only 3.38% of the time for the linear limit for dense configurations, meaning that using the linear limit as opposed to the quadratic limit gives similar sets of lines to choose from. Therefore, there is no notable benefit to using either the linear or the quadratic.

An interesting aspect of these results is that the 3.38% for the linear limit is less than the 10.01% for the logrithmic, undivided limit in sparse configurations but the number of times that the heuristic with linear and quadratic limits found the same number of lines is 100 instances less than the number of times the heuristic with the logrithmic limit found the same number of lines in sparse configurations. This suggests that the dense configurations are more sensitive to changes in limit than sparse configurations are. However, this is beyond the scope of this research and will not be discussed further.

These results show that the logarithmic limit should not be used when dense configurations are being considered if accuracy is a requirement. Furthermore, there is very little difference between using the quadratic and linear limits in the heuristic for both sparse and dense configurations. However, the undivided, linear limit was only reached 3.38% of the time so the unlimited algorithm may work with a limit close to the number of the adjacencies. However, only one size of configuration was considered and to show any conclusive evidence of the changes in the differences as configurations get bigger more tests need to be run. Unfortunately, time constraints did not allow for these tests to be done.

Table 7.8 shows that the undivided, quadratic limit was never reached and the divided limit was only reached 0.01% of the time for dense configurations. Therefore, the limit is implied to be n^2. However, this may change as the configurations get bigger.

7.6 Comparing the line heuristic to the exact solution

This is probably the most important section in this chapter because it evaluates the heuristic in general. The line heuristic is shown to work quite well when compared to the exact solution. However, the exact solution takes an unreasonably long time to compute so only small configurations are considered. As small cases are only considered, cases where the largest differences occur will be investigated to see where the heuristic went wrong and if these cases scale up to larger configurations.

The results shown in this section compare the exact solutions to the solutions found by the line heuristic with a quadratic limit and the least adjacency starting point method in 900 dense and 900 sparse configurations of 20 convex polygons. The average number of lines found by each algorithm is shown in table 7.11 and the differences in the number of lines found by each algorithm is shown in figure 7.12. The row that shows the number of times the heuristic found fewer lines than the exact solution is omitted for obvious reasons.

	Sparse	Packed
Heuristic	4.75	5.47
Exact	4.51	5.07

Table 7.11: Average lines found in the exact solutions and solutions found by the heuristic in configurations of 20 polygons each.

Table 7.11 shows that the average number of lines found by both algorithms is relatively small so any difference in the number of lines in the solutions is considered meaningful. Though, it is a heuristic so some differences should be tolerated.

The data in table 7.12 show that the heuristic found an exact solution more than two thirds of the time in sparse configurations and just less than that for dense configurations. The heuristic found only one more line than the exact solution in 212 out of 900 configurations and two more lines in only *two* out of 900 configurations for the sparse configurations. Similar results hold for the dense configurations. This indicates that the heuristic works well. However the test cases were quite small so the extreme cases are now investigated.

	Sparse			Dense		
Condition	#	average diff	greatest diff	#	average diff	greatest diff
Heuristic > exact	214	1.01	2	347	1.03	2
Heuristic = exact	686	0.00	0	553	0.00	0
Total		0.24			0.40	

Table 7.12: Results from exact tests for 900 dense configurations of 20 polygons each.

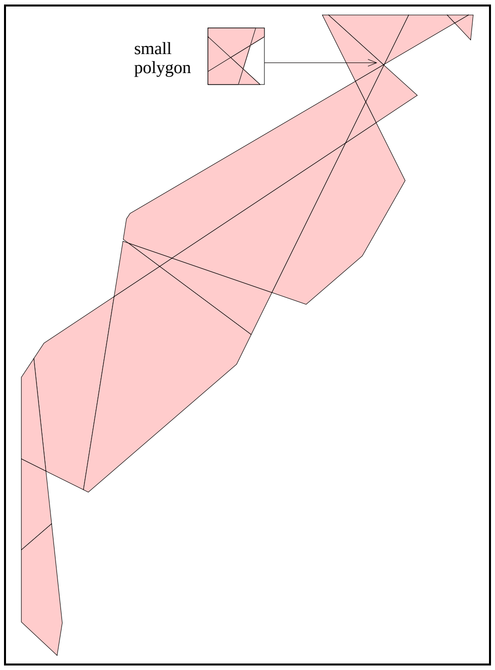


Figure 7.10: A configuration where the heuristic worked relatively badly.

Figure 7.11 shows a configuration where the heuristic found 7 lines but the exact solution contained 5 lines. The square marked "small polygon" is a magnification of the area that the arrow is pointing to. The arrow points to an area where there is a very small polygon that cannot be seen. The configuration shown has ends that are similar to a chain so any attempt at finding the axial lines should start at the ends because axial lines are guaranteed to start there. In this case, the least adjacency method picks the polygon at the bottom, so this condition is partly satisfied. Next, the exact and heuristic solutions will be compared to see where extra lines were added and why.

Figure 7.11 shows the solutions found by the heuristic and the exact solution. Some of the lines *seem* to run parallel to some adjacencies so they are represented by wavy lines. Furthermore, the figures might become confusing if all of the lines were extended as far as they could go, so the extensions are not shown where possible. The lines in the heuristic solution are labelled in the order that they were placed by the heuristic and the lines in the

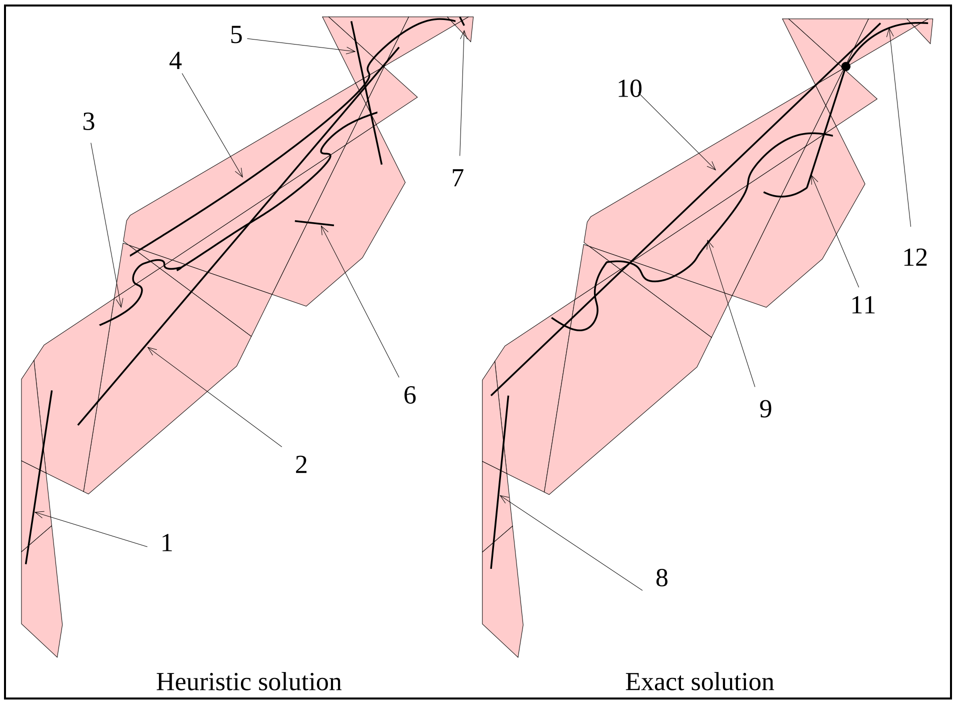


Figure 7.11: The exact and heuristic solutions for a configuration where the heuristic worked relatively badly.

exact solution are only labelled for convenience.

The figure shows that line 1 extended from the bottom polygon then line 2 extended from the closest uncrossed adjacency through the tiny polygon that is shown in figure 7.10. Line 10 in the exact solution is similar to line 2 except it doesn't pass through the small polygon. Line 10 and line 2 are the same length so either would be a candidate for the second line chosen by the heuristic but line 2 was chosen. In this case line 2 was the wrong choice because line 12 could have crossed the adjacencies in the small polygon. Instead, line 5 is used to cross some adjacencies that would have been crossed by line 10, leaving adjacencies that will be crossed by line 7 uncrossed. Line 5 could have been used to cross adjacencies that will be crossed by line 7 but the current orientation is chosen because it is longer. This is a case where a shorter line would have been a better choice than a longer line.

The fact that line 5 is higher than it should be means that it doesn't extend far enough, causing line 6 to be chosen. All of this indicates that line 6 and line 7 are the unnecessary lines that are added by the heuristic. Choosing line 2 instead of line 12 and choosing line 5 instead of a line to cross adjacencies crossed by line 7 caused these lines to be added.

These problems may have been solved by implementing a policy that makes lines that extend into chains have a higher probability of being chosen than lines that extend into an open part of the polygon. However, the heuristic seems to work slightly worse for dense cases. This is because most of the sparse cases are very much like chains and more cases like the above example would occur in dense configurations. In spite of this, the heuristic still finds the exact solution on more than half of the test cases. Furthermore, the differences are relatively small where they do occur.

7.7 Conclusion

This chapter has shown that the best form of the line heuristic out of all of the options considered in this research has the following configuration.

- The least adjacency method or the random method should be used to find a starting point.
- A quadratic limit should be used.

The network method was a surprising failure but some suggestions for improvements are given in the next chapter. It was also shown that a quadratic limit should be used but only if efficiency is not a consideration. Also shown, was that the limit is actually closer to logarithmic for sparse configurations and close to a linear for dense configurations. However, it would be interesting to see how these limits are affected when the configurations grow, as only configurations of one size where tested.

The heuristic was compared to the exact solution and shown to work quite well. The next chapter suggests methods to improve the algorithm in the places where it doesn't work well.

Chapter 8

Future work

8.1 Introduction

This document presents algorithms for a few different areas in axial line placement in convex polygons. However, there is room for improvement in many of these algorithms. This chapter discusses some ideas for improvements to these algorithms as well as some ideas that were not researched due to time constraints or scope.

The first topic for discussion in section 8.2 is a linear algorithm for visibility through a chain of convex polygons. This algorithm is used to determine if an axial line can cross a certain sequence of adjacencies, so it is used in all areas of axial line placement in convex polygons.

The chain algorithm developed in chapter 3 finds a smallest cardinality set of lines to cross the adjacencies in a configuration of convex polygons. However, the sum of the length of the lines in the subset is not guaranteed to be maximal. Section 8.3 gives some direction to any research that tackles this subject.

Section 8.4 discusses the origin of the star of convex polygons and poses it as a topic for future work as well as some related configurations.

Section 8.5 follows with a discussion on the usage of *essential lines* for another starting point method for the heuristic for general configurations of convex polygons.

Section 8.6 discusses possible explanations for the bad performance of the network heuristic as a starting point method. In addition to this, some methods are suggested that would greatly improve its use as a starting point method.

Finally, section 8.7 discusses some improvements to the heuristic. These suggestions may improve the efficiency of the heuristic and may decrease the number of lines in the heuristic.

8.2 Linear visibility algorithm for axial line placement

All of the algorithms for axial line placement would be more efficient if a more efficient algorithm for placing an axial line was found. The algorithm for partial visibility between two edges in a simple polygon is linear. However, in order to place an axial line in a configuration of convex polygons each adjacency that that line crosses must be tested with the first adjacency as shown by the progression in figure 8.1. This means that the algorithm for placing an axial line is quadratic in the number of adjacencies.

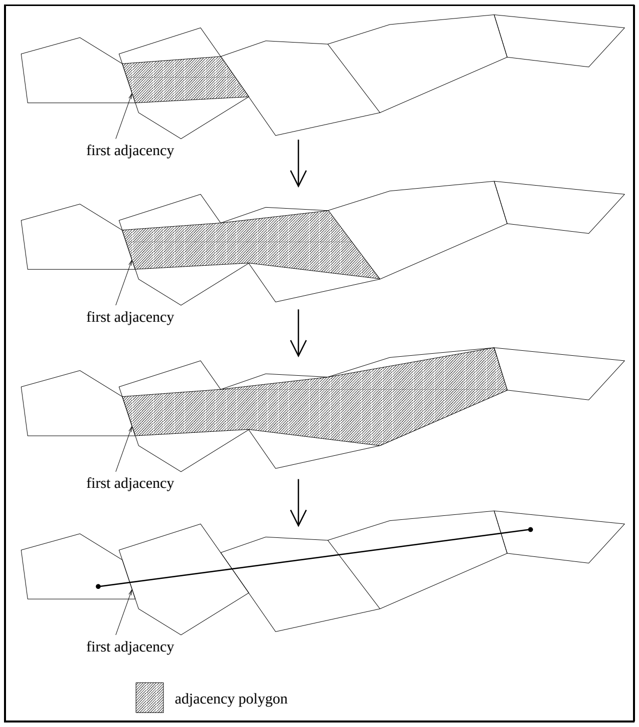


Figure 8.1: The process of placing an axial line.

In van Antwerpen [2002] a linear algorithm is presented that places axial lines in chains of orthogonal rectangles. The algorithm is an adaptation of an algorithm to fit straight lines through data ranges [O'Rourke 1981]. The main problem with the algorithm, other than that it does not work for convex polygons, is that it only works if each adjacency in the chain has no x value less than the adjacencies that came before it in the chain as depicted in figure 8.2. The algorithm cannot be applied to b because adjacencies 5, 6 and 7 have x values smaller than an x value in adjacency 4.

If this algorithm can be adapted to convex polygons then it can be used to make the `search` function from section 6.4 much more efficient because adjacencies can be added one at a time. However, the states would have to be saved to facilitate back stepping through the configuration.

If this algorithm cannot be adapted for convex polygons then it may be possible to do some sort of binary search. This would require a candidate chain such as in figure 8.3. Placing a line that crosses 1 and as many other adjacencies as possible can be done by using the standard visibility algorithm to check if adjacency 1 is partially visible to adjacency 5 through 2, 3 and 4. If it is, then adjacencies 1 and 7 should be checked for partial visibility, otherwise 1 and 3 should be checked and so on.

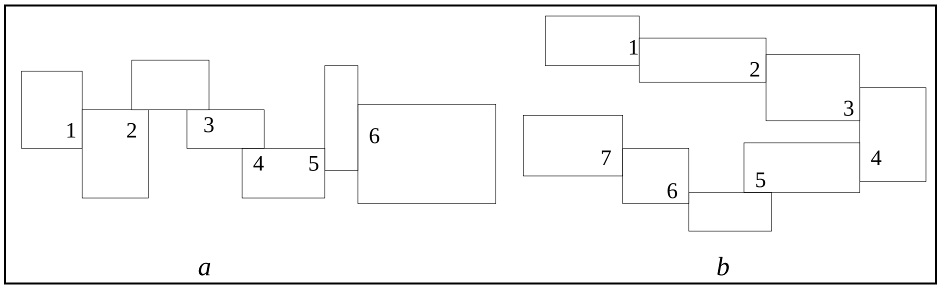


Figure 8.2: Two chains of convex polygons with the adjacencies labelled in order. The linear chain algorithm can be applied to a but not to b.

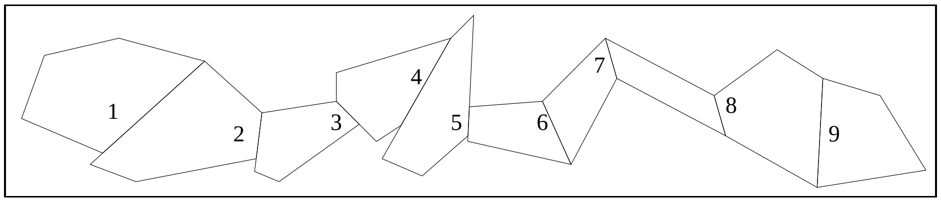


Figure 8.3: A chain with adjacencies labelled 1 to 9.

This method would guarantee an $O(n \log n)$ running time. However, its uses would be restricted because a candidate chain must be given. The `search` function only builds paths by checking visibility each time a node is added. However, this may be solved by finding chains of adjacencies that a line cannot cross or can only extend to a certain depth. This will be discussed in more detail in section 8.7 which discusses ways to improve the heuristic itself.

8.3 Maximising the lines in a chain

The definition of axial line placement calls for the lines in any solution to be as long as possible, but it does not require that the solution be the one with the longest lines. In other words it is not required to find the minimum solution where the sum of the length of the lines is maximal. Finding such a solution is much harder than finding a solution matching the original criteria. However, Space Syntax requires that the lines be as long as possible so such a solution would be better than the current one because the lines could be longer. This section discusses this problem for chains of convex polygons and goes to show that this problem definition is much harder to solve for than the definition used in this research.

Previous research (Sanders *et al.* [2000b] and Phillips [2001]) has solved the problem of axial line placement in chains of polygons by using an approach that first finds the set of forward[1] lines and then the set of backward lines then merging these to get a solution. The

[1]The forward direction is considered as going from left to right and right to left for backwards

forward lines are generated by starting at the first adjacency in the chain and extending a line as far forward (chainwise) as possible. Where that line ended a new line is started at the next uncrossed adjacency and extended as far forward as possible. This process continues until the last adjacency is crossed. This process yields a minimum cardinality set of lines is the same as algorithm 3 with the **while** loop removed. The set of backward lines is formed by performing the same operation used to find the forward lines but starting at the last adjacency and extending lines in the opposite direction. This approach is used in this section to suggest a way of finding the solution where the lines are of maximal length and shows why this problem is difficult.

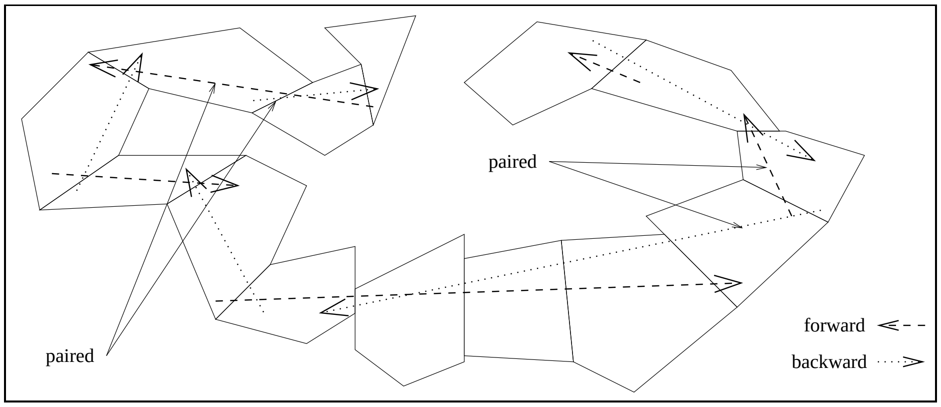


Figure 8.4: Forward and backward lines.

The chain algorithm (algorithm 3 from chapter 3) finds the forward set of lines then extends them backwards to maximise them. This could be modified to produce longer lines if both backward and forward sets of lines where found and where extended. The solution with the longest lines could then be picked. Figure 8.4 shows an example of a forward set and a backward set of lines. If this procedure was applied to the example in figure 8.4 then the solution shown in figure 8.5 would be found where the lines cross a total of 15 adjacencies. However, this is not the solution where the lines cross the greatest number of adjacencies. Such a solution is shown in figure 8.6. This solution takes a combination of the forward and backward lines. However, the following discussion will show that a solution that has the longest lines may contain a line that cannot be derived from either a forward or a backward line but must lie somewhere in between a pair or them.

Both the forward and the backward set have the same number of lines because can be generated by the chain algorithm, so it is possible to match each line in one set with a line in the other. The first line from the forward set will be paired with the last line in the backward set. The second line in the forward set is paired with the second last line in the backward set and so on. These pairs are the lines that are "merged" if the previous technique of placing axial lines in chains of rectangles is used.

Figure 8.7 shows a set of forward and backward lines on a section of a chain. Note that the figure is just an abstract representation and not a real chain. Once a forward line and

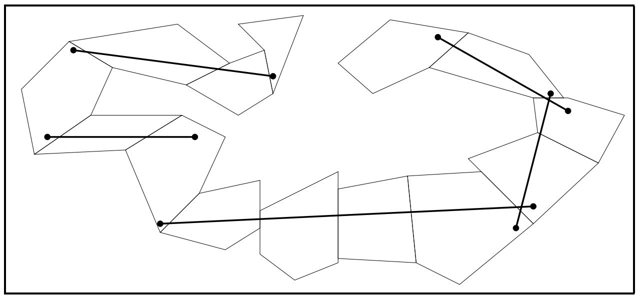


Figure 8.5: Solution from chain algorithm

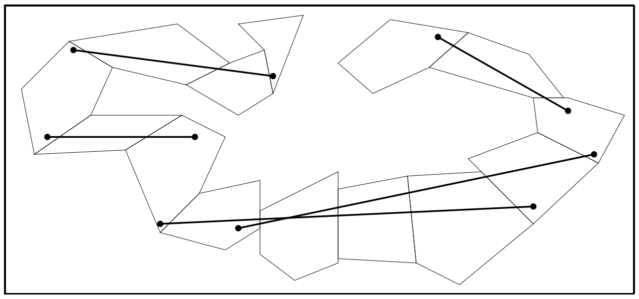


Figure 8.6: Set of lines that cross the greatest number of adjacencies

a backward line are paired, the adjacencies that this pair spans forms a range in which an axial line must occur. This is because the right end of the forward lines are as far right as the lines can possibly go while still retaining the minimal solution. Similarly, the backward lines are the lines where their left point is as far left as possible. The ranges in figure 8.7 are labelled a, b and c. The relevant adjacencies are labelled 0 to 21.

A *single* axial line *must* cross the adjacencies that both backward and forward lines cross in a pair. For example, consider range b in figure 8.7. The forward and the backward lines that form this range both cross adjacencies 9 to 13. This range is called an "essentail range" because an axial line must cross these adjacencies in order for it to be in the minimum cardinality solution. The other "essential" ranges where axial lines must occur in figure 8.7 are 17 to 20 and adjacency 5 on its own. Clearly, if an axial line was chosen to *only* cross adjacencies 2 to 5 then lines from the neighbouring pairs would have to be extended to cross the uncrossed adjacencies, meaning that the local choice of line would affect the choice of lines for other pairs.

Consider range b. Suppose that adjacency 7 is partially visible to 15 but 6 is not partially visible to 16. It is relatively easy to determine that the longest line that crosses the essential range of b starts at 7 and ends at 15. However, suppose that adjacency 4 cannot see adjacency 6, then the line from 7 to 15 would affect the choice of line at range a because the forward line would have to be picked where the longer line in range a would be the backward line. If the forward line is chosen in this case then the choice of line for the range left of a would

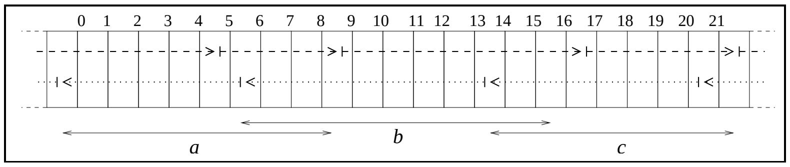


Figure 8.7: Representation of pairs on a section of chain.

also be affected, but it not possible to say whether this would be for the better at this point. Therefore, the choice at the local stage affects the choice at other stages but it is seemingly not possible to say how the local choice affects other choices. Figure 8.8 shows how this situation can occur in an actual chain.

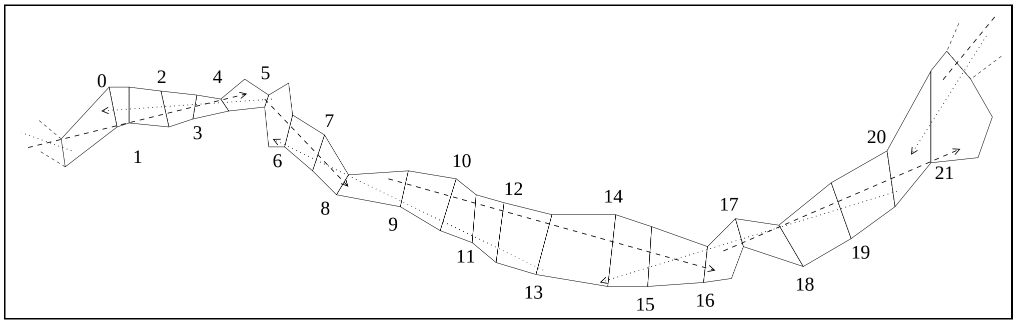


Figure 8.8: A real example of figure 8.7

Finding an algorithm to find the truly maximal solution or proving that the problem is NP-complete is left for future work.

8.4 Configurations related to the star of convex polygons

The star of convex polygons originated from another special case which has not been solved, but investigating it produced the algorithm to solve the star. This section discusses how the star relates to this special case as well some ideas that could lead to a solution. Naming this special case is also left for future work. An example of the original problem is given in figure 8.9.

The special case is made of a main chain and a number of other “offshoot” chains. In figure 8.9 the main chain is made up of the polygons that are shaded. Each polygon in the main chain must be adjacent to at most one polygon that is not part of the main chain and begins (or ends) another chain of convex polygons. No other polygon in this “offshoot” chain can be adjacent to any other polygon that is not part of this chain.

This special case could be useful because of its similarity to a road map. The main chain would represent a major thoroughfare and the offshoot chains represent roads that intersect with it.

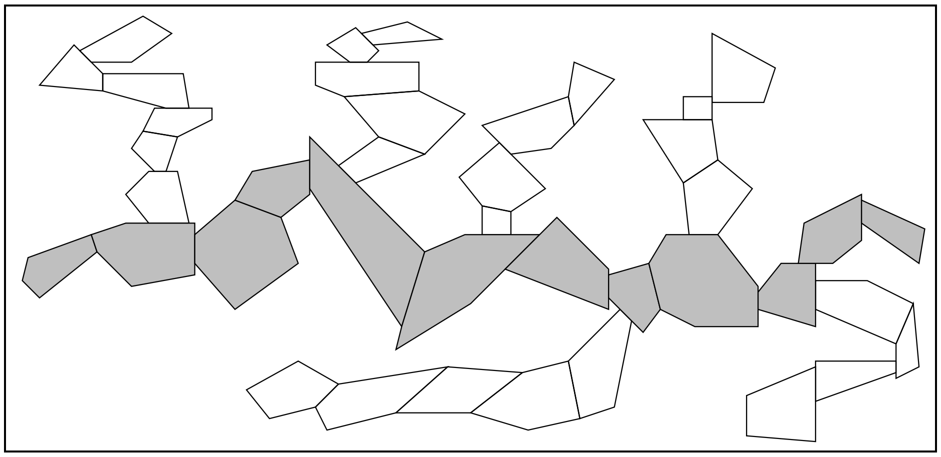


Figure 8.9: Origin of the star

It probably is not obvious what this problem has to do with the star of convex polygons but figure 8.10 shows a rather bizarre special case of this special case which shows the connection. The shaded polygons form the offshoot chains and the rest form the main chain. The axial lines that cross the adjacencies in the figure are all the possible merges between the chains across the configuration.

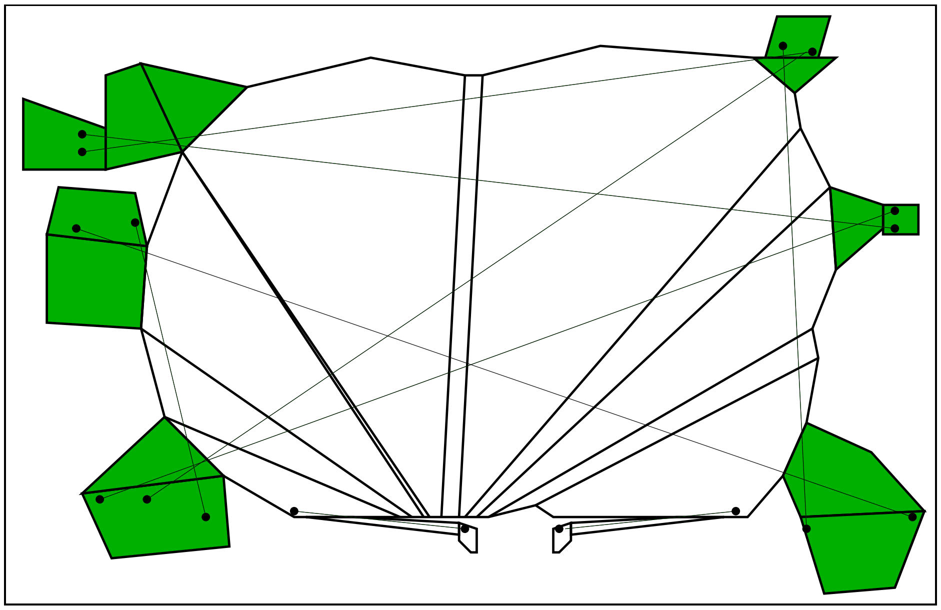


Figure 8.10: A bizarre special case

Using the maximum cardinality matching algorithm on these lines will solve this particular case but it is possible that the matching could leave some of the adjacencies in the main chain uncrossed such as the arrangement shown in figure 8.11. In figure 8.11, either the vertical lines or the horizontal lines would be chosen by the matching algorithm so it would be desirable to bias the matching towards the lines that cross the most adjacencies. This can be done by using the weighted, maximum cardinality matching algorithm where the number of adjacencies each axial line crosses is used for the weight. This alone still does not solve the whole problem alone because there are adjacencies that can be left uncrossed.

Once the weighted matching is done, it is possible that the solution with the smallest number of axial lines can be found by just crossing the uncrossed adjacencies using the chain algorithm. However, there may be many configurations like the one shown in figure 8.10 that are linked together by the main chain. In that case, the way that the matching is done in one part of the configuration may affect the matching done in another because the main chain may produce lines for merging.

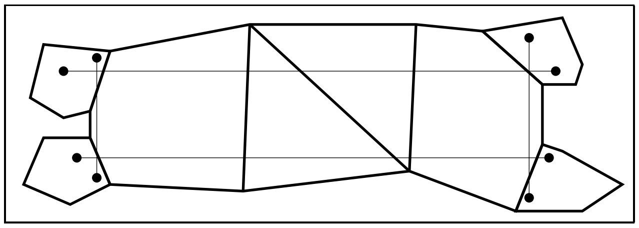


Figure 8.11: Weighted maximum cardinality matching must be used

It is suggested that any attempt at a solution to this problem should start with applying algorithm 4 to the offshoot chains so the axial line that crosses the adjacency between the last polygon in the offshoot chain and the main chain is as short as possible. The chain that begins the main chain and the chain that ends the main chain should be considered as offshoot chains as well as demonstrated in figure 8.12. Figure 8.10 suggests that the problem should be transformed to some sort of matching problem in much the same way that the star of convex polygons was transformed by merging the lines across the main chain.

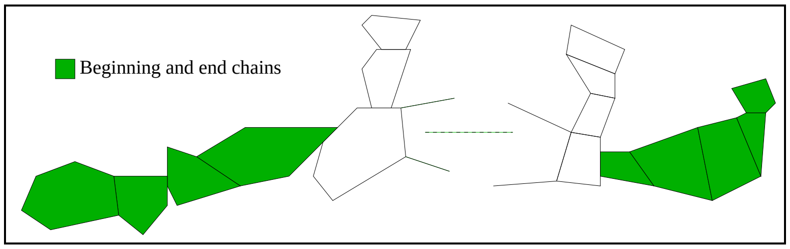


Figure 8.12: Considering the beginning chain as an offshoot

The next step would be to identify all the possible merges across the main chain and separate the merges that would not have common vertices in the corresponding graph. At this point a greedy approach may fail because it is difficult to say how separate matchings would affect others. Once a matching has been done there are more lines present in the main chain which present new opportunities for matching so the merges may have to be redone.

The problem presented here is more difficult than the star and may be NP-complete, but it opens up a new area of special cases. Additional special cases can be formed by replacing the central polygon with other arrangements. This particular problem can be seen as a star with a chain replacing the central polygon. The more general the replacement for the central polygon becomes, the more likely the configuration will occur in general configurations and the more useful the special case would be in a heuristic.

8.5 Another starting point method – essential lines

Three methods have been given to find a starting point for the heuristic. The random and least adjacency methods are desirable because they do not interfere with the workings of the line heuristic. The network method is desirable because it solves a part of the configuration but may not merge well with the rest. A method that would combine the desirable characteristics of both of these methods would be to find the essential lines.

Essential lines are defined in section 2.3 as lines that cross an adjacency that cannot be crossed by another line that is not wholly contained in the essential line (i.e. redundant lines are excluded). These are lines that must be in a solution with the least number of longest lines. If these are found then the line heuristic would begin to pick lines that extend from lines that are known to be in the minimal solution.

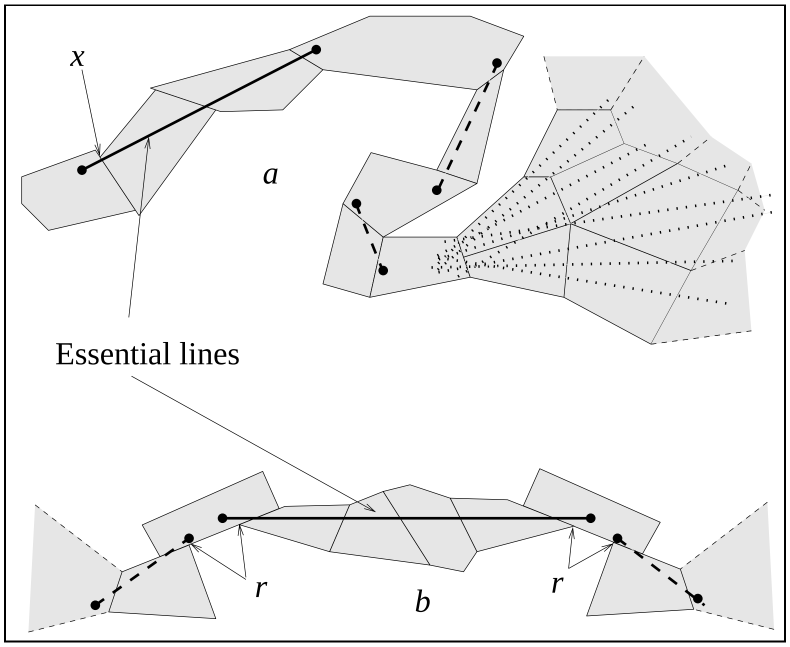


Figure 8.13: Essential line starting point method.

Figure 8.13 shows two configurations where essential lines occur. Configuration a shows a chain that is attached to a general configuration on one end and the other end is not attached. The line that extends from the unattached side is essential because it is the only non redundant line to cross adjacency x. This case can easily be identified by finding all the chains with an unattached polygon and extending a line from the unattached polygon as far into the chain as it can go. If the line cannot extend into the general part of the configuration then that line is essential.

The essential line in configuration a is the same line that the chain algorithm would place. If this line was used as a starting point then the broken lines in configuration a would be found but they could also be found by using the chain algorithm from the unattached edge. However, the line heuristic would find the same result but may be inefficient so the chain algorithm would probably be better.

At this point, the broken lines can be seen as essential but they do not meet the definition given in section 2.3 because other lines that are not redundant can cross all the adjacencies that the broken lines cross. This can be rectified by changing the definition of redundant to the following.

- Redundant - each of the adjacencies crossed by this line is already crossed by one choice line *and one or more essential lines*.

This definition means that the broken line closest (chain wise) to the essential line indicated becomes essential, which, in turn, means that the other broken line is also essential. However, a line that is made redundant by this definition may still have been in a minimum cardinality solution because the choice line that excluded it may not have been chosen because all of the adjacencies crossed by this choice line may be crossed by the redundant line and another choice line. This only means that the solution without the redundant line might be a solution where the total length of the lines is less than that of the total length of the lines in the solution with the redundant lines but the number of lines would still be the same in both solutions because all the adjacencies crossed by the redundant lines are crossed by the choice line and some essential lines.

Configuration b shows a case where an essential line occurs somewhere in the middle of a chain. This may be a very restrictive case because it seems to occur only when two polygons are adjacent to the same edge of another polygon in two different places in a chain. The adjacencies, indicated as r in figure 8.13, satisfy this condition.

Other lines can be placed based on the position of this essential line by using the chain algorithm starting at the polygons that contain the ends of the essential line and continuing outwards. This case is not yet fully understood so future work may investigate the requirements and detection of this special case.

Other configurations may exist that can be identified as having essential lines so future work may be dedicated to finding these configurations or proving that none exist.

8.6 Improvements to the network of stars heuristic

Chapter 7 shows that the network of stars algorithm works relatively badly as a starting point method. This may be because the heuristic itself works badly. Unfortunately, the empirical analysis focused on testing the line heuristic itself. It would be desirable for future work to improve and compare the network heuristic to this heuristic and the exact solution. The conclusion to chapter 5 has already suggested that the network of stars heuristic can be improved by transforming it to some graph problem.

However, the stars of convex polygons algorithm from chapter 4 forms most of the network heuristic and always finds the minimal solution. Therefore, it is not likely that it works worse than the line heuristic in networks of stars. This means that the solution produced by the network algorithm is not compatible with the solution found by the line heuristic.

The most probable reason for this incompatibility is that the lines in the network are not extended into the rest of the configuration where possible. This means that there can be at most one extra line added for each unextended line from the network. However, a line found by the line heuristic could extend into the network and may cross all the adjacencies crossed by an unextended line, then the unextended line would be considered redundant and removed. This is not a very good solution because there is no guarantee that the line heuristic would find such a line.

The easiest way to solve the problem would be to plug the lines that are found by the heuristic and have ends in the rest of the configuration into the `search` function from section 6.4 and picking the longest line. This was not done because, by the time this had been discovered, the test cases had already been run. Time was running short and it would have been impractical to re-run the test cases. However, there are other ways this could be done that may be more effective.

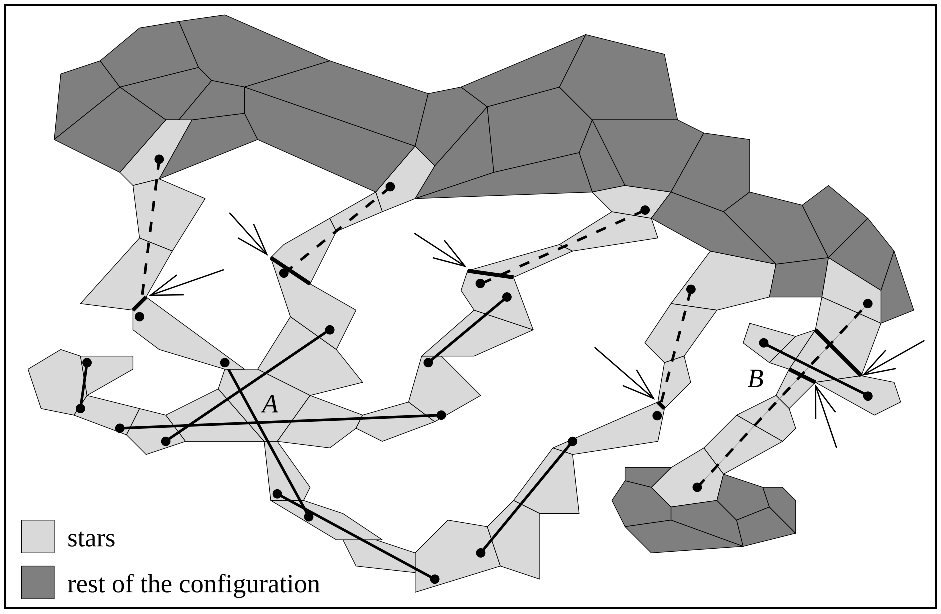


Figure 8.14: A general configuration with stars A and B

In figure 8.14, the broken lines are lines that can be extended into the rest of the configuration. Instead of extending these lines into the rest of the configuration they can be deleted. The remaining lines can be used as a starting point, meaning that the adjacencies that are indicated by the arrows in figure 8.14 are marked as origins.

This method gives more freedom of choice in cases like star B in the figure, where a line from the rest of the configuration can cross a star back into the rest of the configuration.

The network heuristic can also be redone so that the broken lines are as short as possible by using the `reduceChain` function described in the heuristic. This would mean that the lines that extend from the star can cross as many adjacencies in the rest of the configuration as possible.

8.7 Improvements to the line heuristic

There are two areas where the heuristic can be improved.

- Reducing the complexity
- Reducing the number of lines found

The first attempt at reducing the complexity would be to find a sub-quadratic algorithm for visibility. The visibility algorithm is already linear, however, only one adjacency is tested at a time so the algorithm for a whole chain of adjacencies becomes quadratic. The algorithm would have to be an online algorithm because it cannot be known which adjacencies are going to be added in the future. Finding such an algorithm would make *any* heuristic more efficient because the visibility algorithm has to be used when doing axial line placement.

Another improvement in the efficiency could come from improving the data structure. Some balanced tree structures could be used to improve the efficiency of the insertion and removal algorithms. As the data structure stands, some of the lines are repeated because they come from different origin adjacencies. An efficient method of finding repeats may be beneficial.

If the limit on the number of polygons is shown to be not polynomial in the number of polygons, then the method of finding all the lines from an adjacency can be improved by using a search that only searches along lines where an axial line is likely to be placed. Figure 8.15 demonstrates this concept. Searching in the *right* direction will most likely give longer lines than searching in the *wrong* direction. Some angular sweep method may prove to be useful but may prove to be just as inefficient. This may also decrease the number of lines because the search algorithm in this document may terminate before the *right* directions have been searched, resulting in much shorter lines than would otherwise be possible.

This method of choosing the right direction can improve the approximation for the length of a chain by using the method of finding chains of adjacencies that a line cannot cross or can only extend to a certain depth, mentioned briefly in section 8.2. This uses the idea of “kinks” used in Sanders [2002] to develop heuristics for ALP-ALOR. For *rectangles*,

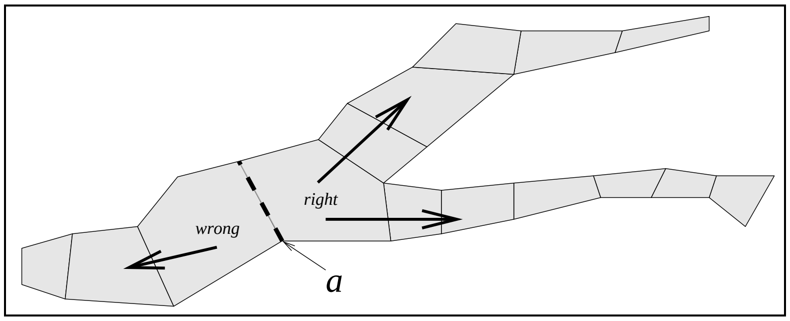


Figure 8.15: Searching in the right direction.

a kink occurs when two polygons in a chain are adjacent to the same side of another polygon in the same chain. Figure 8.16 shows an example of a kink in a chain.

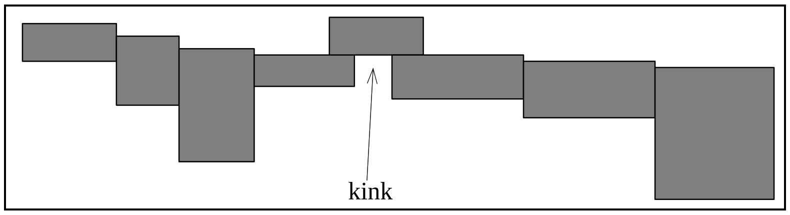


Figure 8.16: An example of a kink in a chain of rectangles.

If a kink occurs in a chain then it is impossible for one axial line to cross all the adjacencies in the chain. The search function can have a preprocessing stage where chains are searched for using a depth first search. The search stops descending when a kink is discovered.

The definition of kinks given for rectangles works for convex polygons, however, they occur infrequently, but there are other methods of identifying kinks in a chain. Once such method is demonstrated in figure 8.17.

Assume that the search is adding adjacency C. To identify the kinks, consider the lines A_tB_b, B_bC_t, A_bB_t and B_tC_b, and form the angles α and β. If $\alpha \geq \pi$ or $\beta \geq \pi$ then an axial line cannot cross adjacency A, B and C. Identifying other methods is left for future work. This method can be used in conjunction with the angular sweep method to get a better idea of the length of a line if it was extended in a certain direction.

The number of lines found by the heuristic can be reduced by implementing the essential lines method for finding a starting point. This would start the heuristic in a polygon where a line in the minimal solution is known to start. However, there may be no essential lines in the configuration so another method will have to be used.

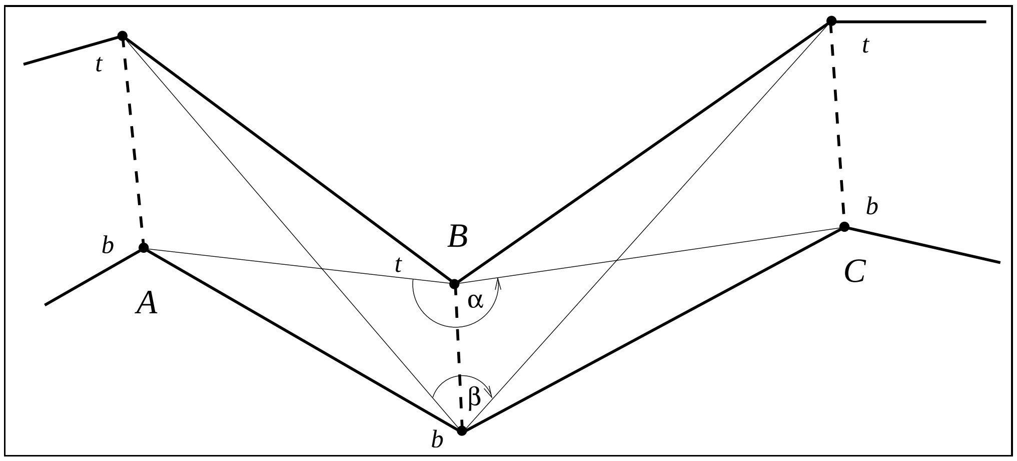


Figure 8.17: An example of a kink in a chain of convex polygons.

8.8 Conclusion

This chapter shows that the subject of axial line placement in convex polygons is far from closed. Many more special cases that have polynomial solutions may exist and more efficient heuristics need to be found that find better solutions. However, this document has made a large contribution to the field by introducing a new heuristic and finding polynomial time algorithms for two special cases.

Chapter 9

Conclusion

9.1 Introduction

The research presented in this document has covered a few different areas in axial line placement in convex polygons to develop the heuristic for the general case. This chapter gives some final remarks on the various elements that have been brought together to find a good heuristic and the heuristic itself is remarked upon too.

The first three sections discuss the special cases that were studied in this research. These special cases where only used as preprocessing for the heuristic for the general case but they have applications in other areas of axial line placement. The chains of convex polygons are the most basic special case considered and can be considered an extension of the work done in visibility. The resolution of the conflict between stars of convex polygons resulted in the network of stars which is similar to a town plan, and the study of town plans is a major application area of axial line placement.

The heuristic for the general case of axial line placement is discussed in section 9.5 where the results are reviewed. The algorithm is the first attempt at producing a heuristic for finding a minimum cardinality set of lines that cross the adjacencies in a configuration of convex polygons, so the research is considered a success even though the heuristic has a high complexity.

9.2 Chains of convex polygons

The study of chains of polygons started with placing orthogonal lines in configurations of orthogonal rectangles and was then expanded to allow the placement of lines of arbitrary orientation. This document has taken the next step, which is to develop an algorithm for placing lines of arbitrary orientation in configurations of convex polygons.

The algorithm is proven to find the minimum cardinality set of lines to cross the adjacencies in a chain and is $O(n^2)$ where n is the number of adjacencies.

The algorithm makes a large step towards completing the work on chains of polygons because convex polygons are the most general type of polygon allowed by the problem of axial line placement. The only area that may need to be researched is finding the solution where the sum of the lines is maximal (discussed in chapter 8). However, this requires a restatement of axial line placement, so this may never be necessary. If this is the case, then this document completes the study of axial line placement in chains of polygons.

Chains of convex polygons can be viewed as a fundamental part of axial line placement since they can be seen as an extension of the visibility algorithm. The `search` algorithm from chapter 6 finds a chain with adjacencies that can be crossed by one axial line and are built in a depth first manner by using the visibility algorithm. The chain algorithm could be used in the same manner with more advanced path finding strategies to build a better heuristic.

9.3 Stars of convex polygons

The star of convex polygons is a new special configuration of convex polygons and is made up of a central polygon with a number of chains connected to it. The future work chapter has suggested that future research should attempt to replace the central polygon with other configurations such as a chain. However, the problem of a star of convex polygons transformed into maximum cardinality matching which is a well known graph theory problem. In light of this, it may be possible that more special cases can be found that transform into other graph theory problems.

This document has presented a quadratic algorithm to find the least number of axial lines to cross the adjacencies in a star of convex polygons. This is a good result as it uses the chain algorithm multiple times yet manages to have the same complexity as the chain algorithm itself.

9.4 Networks of stars of convex polygons

The purpose behind investigating special cases is to integrate them into a heuristic. In this research they were detected in a configuration and solved using the appropriate algorithm. If this is done with stars then conflicts happen between stars that share the same chains. To resolve this conflict, the network of stars special case was developed.

The network of stars differs from the other two special cases because it is unlikely that there exists a polynomial time algorithm to place axial lines across its adjacencies. However, town planning is an application of axial line placement and the network of stars resembles a town plan. This means that the algorithm developed is relevant besides its application to the heuristic so further research would be useful.

9.5 The heuristic for general configurations of convex polygons

The greedy heuristic for general configurations of convex polygons is an algorithm that can be applied to any configuration that will be considered in the area of axial line placement.

Chapter 7 shows that the heuristic works well, at least for small cases.

At each step, the heuristic chooses a line based upon the lines already chosen. This means that some initial set of lines needs to be found. Various starting point methods were developed and are presented in this document. These starting point methods include the network of stars algorithm which is shown to work badly relative to the other starting point methods presented. However, this is conjectured to be because the solution found by the network of stars algorithm is not merged well with the solution found by the heuristic rather than because the solution found by the heuristic is bad itself. This document conjectures that further work on merging the two solutions will probably show that this approach produces better solutions than the other starting point methods.

9.6 Conclusion

This document gives solutions to many different areas of axial line placement in convex polygons. The research nearly completes the work on chains and introduces new types of special cases that may have polynomial time solutions. Additionally, the special cases are much more suited for development than trees of convex polygons.

The heuristic presented in this document is not very efficient but it is the first algorithm that can be used to find a set of lines to cross the adjacencies in a set of convex polygons. This is a good result because the problem of axial line placement is NP-complete and an algorithm now exists that can be applied to any configuration of polygons that will be considered in the area of axial line placement.

Additionally, previous research has focused on developing heuristics for and finding special cases of configurations of rectangles. This meant that the solutions tended to focus on the geometry of rectangles rather than the nature of the problem of axial line placement. Attempted to abstract away from the geometry as much as possible. This produced a special case that is solvable in polynomial time and a good heuristic which may pave the way for further research.

Appendix A

Terminology

Various terms are defined here that are used throughout the document. Almost all of the terms are defined as in Sanders [2002] but exist in many other resources.

- A *point* p is represented by a pair of coordinates (x, y) in Euclidean space.
- A *line* is represented by two points p and q (which can be any two distinct points on the line) and is denoted $-pq-$.
- A line is *orthogonal* if it is parallel to one of the Cartesian axes.
- A *line segment* is represented by a pair of points p and q where the points are the endpoints of the line segment; and the line segment is denoted by pq.
- A line segment is *orthogonal* if it is parallel to one of the Cartesian axes.
- A *path* is a sequence of points $p_1, p_2, ..., p_n$ and the line segments joining them (see fig A.1).
- The line segments in a path are called *edges*.
- A *closed path* is a path whose last point is the same as its first point (see fig A.1).
- A closed path is also called a *polygon*.
- The points defining the polygon are called the *vertices* of the polygon (see fig A.1).
- A *polygon* P can also be defined as a collection of n vertices, $v_1, v_2, ..., v_n$, and n edges, $v_1v_2, v_2v_3, ..., v_{n-1}v_n, v_nv_1$.
- A *simple polygon* is a polygon where no two non-consecutive edges intersect (see fig A.1).
- The set of points enclosed by a simple polygon forms the *interior* of the polygon.

- A *convex polygon* is a simple polygon whose internal angles are all less than or equal to 180 degrees. Alternatively, if a car was driving along the boundary of a convex polygon then it would only make left turns at the vertices (or right turns depending on which way it was going) (see fig A.1).

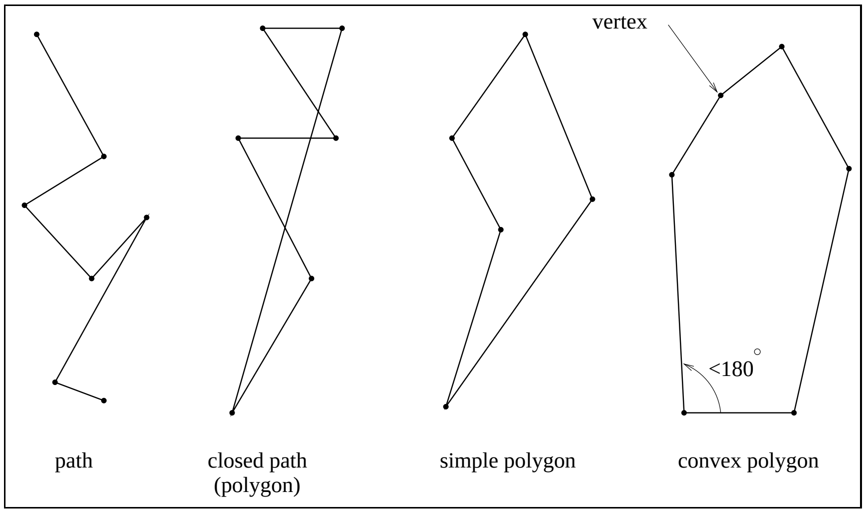


Figure A.1: Polygons

- Let A and B be two polygons that do not overlap. If an edge from A and an edge from B share a line segment, this line segment is an *adjacency* (see fig A.2).

- A *chain of convex polygons* is a configuration of convex polygons where all polygons are adjacent to at most two other polygons. Two polygons in the chain must only be adjacent to one other polygon. These will form the beginning and the end of the chain. This ensures that the chain is not a loop (see fig A.2).

- If P is a polygon with or without holes, a *partition* of P is the set of convex polygons that covers the entire area of P without overlapping. Figure A.3 shows an example of a partition with the original polygon on the right. The number of polygons in a partition should be minimal but finding such a partition has been shown to be NP-hard [Lingas 1982].

- *Axial lines* are one dimensional extensions of the sight lines from particular spaces [Hillier *et al.* 1983]. In this document, an axial line is defined by the set of adjencies it crosses.

- *ALP-OLOR* is the problem of placing lines that are orthogonal to the Cartesian axes upon adjacencies between orthogonal rectangles. Where orthogonal rectangles are rectangles with edges orthogonal to the Cartesian axes. This is discussed further in section 2.3.1.

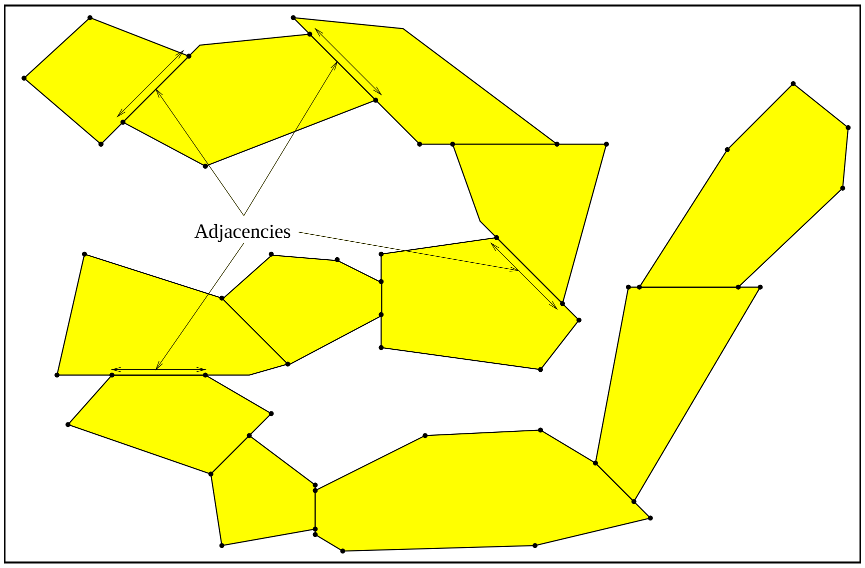


Figure A.2: A chain of convex polygons

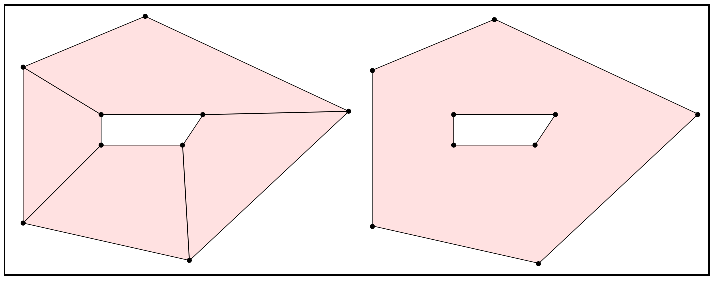


Figure A.3: A partition with the original polygon on the right

- *ALP-ALOR* is the problem of placing lines of arbitrary orientation upon adjacencies between orthogonal rectangles. ALP-ALOR is discussed further in section 2.3.2.
- *ALP-ALCP* is the problem of placing lines of arbitrary orientation upon adjacencies in collections of convex polygons. A special case of ALP-ALCP is discussed in section 2.3.4.

Appendix B

A property of a minimum cardinality set of lines for ALP-ALCP in chains of convex polygons.

Lemma 3 *No minumum cardinality solution to axial line placement in chains of convex polygons can have three or more lines that cross a single adjacency.*

Proof – by contradiction

Assume that there are three lines, a, b and c in a minimal solution to axial line placement in chains of convex polygons that cross adjacency q. For this proof, a line is considered to end after another line begins if it ends on the same adjacency as the other begins. There are four cases to consider below which are illustrated in figures B.1, B.2, B.3 and B.4. Without loss of generality, assume that a starts before b and c, and b starts before c for the first three cases.

1. a ends after b and c
2. b ends after a and c
3. c ends after a and b
4. any of the lines starts (ends) at the same point as another starts (ends).

Case 1 in figure B.1: In this case, a begins before the two other lines and ends after the two other lines. By lemma 1 a crosses all the adjacencies that b and c cross making them redundant, meaning they cannot be in a solution with the least amount of lines that contains a.

Case 2 in figure B.2: b begins before c and ends after c, therefore, by lemma 1 c is redundant and cannot be in any minimal solution containing a and b.

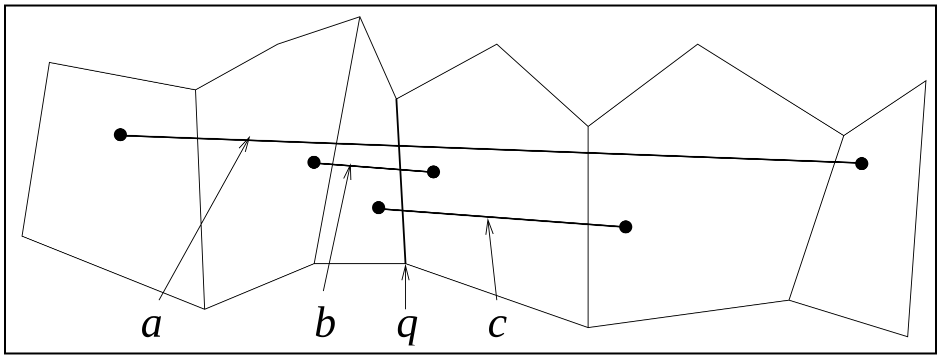


Figure B.1: Case 1: b and c are redundant

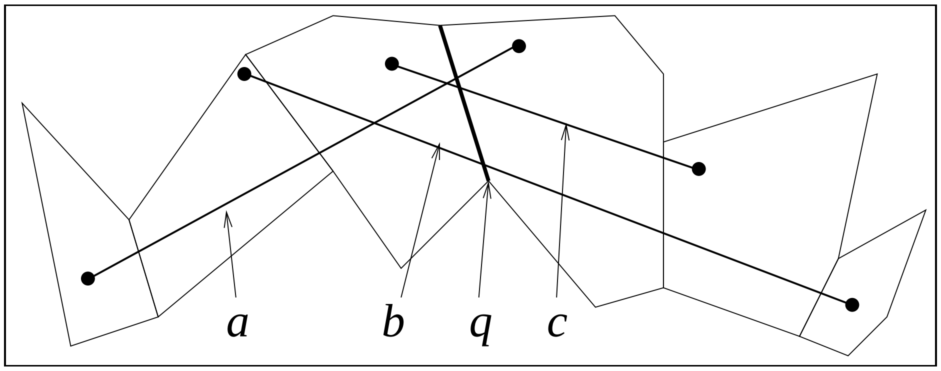


Figure B.2: Case 2: c is redundant

Case 3 in figure B.3 : Here, a begins before the start of line b and ends at some adjacency after q, so all adjacencies crossed by b, before q are crossed by a, by lemma 1. Similarly, c ends after b ends, and c starts before q so c crosses all the adjacencies after q that b crosses, by lemma 1. Now, both a and c cross q so a together with c cross all the adjacencies that b crosses making b redundant. Therefore b is not in any minimal solution that contains a and c.

Case 4 in figure B.4: If any of the lines start at the same point as any of the other lines, then one of the lines must cross the adjacencies that the other crosses, by lemma 1, so one of the lines cannot be in the minimal solution. Similarly, proved when they end at the same point. $\square$

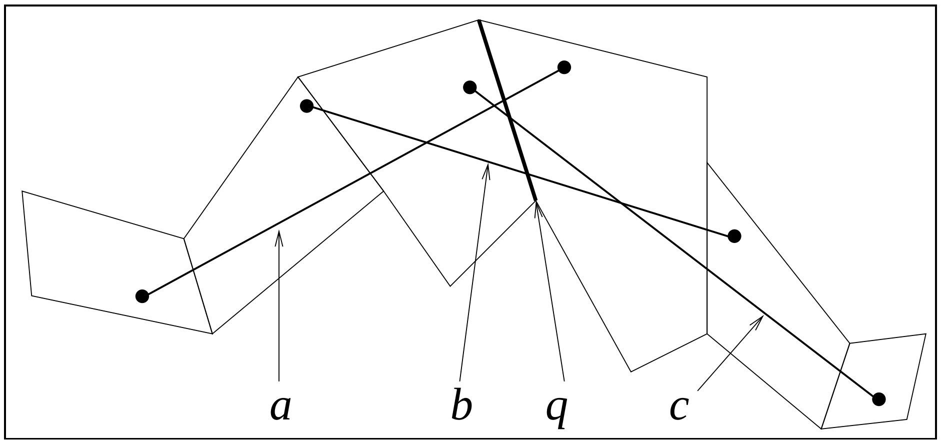


Figure B.3: Case 3: b is redundant

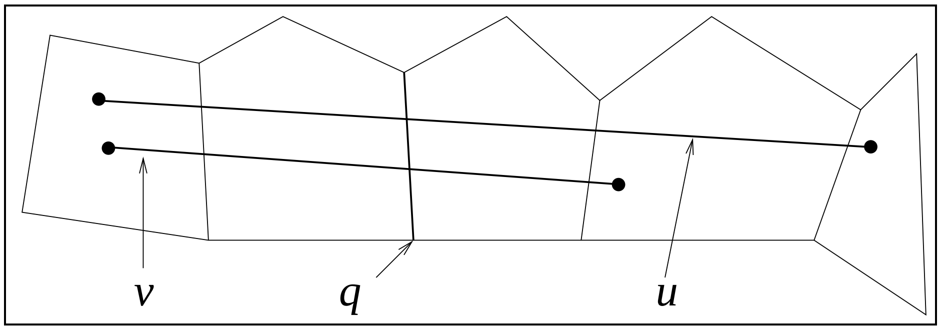


Figure B.4: Case 4: v is redundant

References

[Adler *et al.* 2001] J. Adler, G.D. Christelis, J.A. Deneys, G.D. Konidaris, G.Lewis, A.G. Lipson, R.L. Phillips, D.K. Scott-Dawkins, D.A. Shell, B.V. Strydom, W.M. Trakman, and L.D. Van Gool. Finding adjacencies in non-overlapping polygons. In *Electronic Proceedings of South African Institute of Computer Scientists and Information Technologists Annual Research Symposium*, 2001.

[Asano *et al.* 1999] T. Asano, S.K. Ghosh, and T.C. Shermer. *Visibility in the Plane, Handbook on Computational Geometry*, chapter 19. Elsevier Science, 1999.

[Avis and Toussaint 1981] D. Avis and G. T. Toussaint. An optimal algorithm for determining the visibility of a polygon from an edge. *IEEE Transactions on Computers*, C-30:910–914, December 1981.

[Avis and Wenger 1987] D. Avis and R. Wenger. Algorithms for line stabbers in space. In *Third ACM conference on computational geometry*, pages 300–307, 1987.

[Avis *et al.* 1981] D. Avis, T. Gum, and G. Toussaint. Visibility between two edges of a simple polygon. *The Visual Computer*, 2:342–357, 1981.

[Berge 1957] C. Berge. Two Theorems in Graph Theory. *Proceedings of the National Academy of Sciences*, 43:842–844, 1957.

[du Plessis and Sanders 2000] N. du Plessis and I. D. Sanders. Partial Edge Visibility in Chains of Orthogonal Rectangles. Technical Report TR-Wits-CS-2000-15, Department of Computer Science, University of the Witwatersrand, September 2000.

[Edelsbrunner *et al.* 1982] H. Edelsbrunner, H.A. Maurer, F.P. Preparata, A.L. Rosenberg, E. Welzl, and D. Wood. Stabbing line segments. *BIT*, 22:274–281, 1982.

[Edmonds 1965] J. Edmonds. Paths, trees and flowers. *Canadian journal of Mathematics*, 17:449–497, 1965.

[El Gindy and Avis 1981] H. A. El Gindy and D. Avis. A Linear Algorithm for Computing the Visibility Polygon from a Point. *Journal of Algorithms*, 2:186–197, 1981.

[Gabow and Tarjan 1983] H. N. Gabow and R. E. Tarjan. A linear time algorithm for a special case of disjoint set union. In *the Proceedings of the Annual ACM Symposium of the Theory of Computing*, volume 15, pages 246–251, Boston, 1983.

[Gabow 1976] H. N. Gabow. An efficient implementation of Edmond's maximum matching algorithm. *Journal of the ACM*, 23:221–234, 1976.

[Galil 1986] Z. Galil. Efficient Algorithms for Finding Maximum Matching in Graphs. *Computing Surveys*, 18(1):23–36, March 1986.

[Gewali and Ntafos 1993] L. Gewali and S. Ntafos. Covering grids and orthogonal polygons with periscope guards. *Computational Geometry: Theory and Applications*, 2:309–334, 1993.

[Guibas *et al.* 1986] Leonidas J. Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert E. Tarjan. Linear Time Algorithms for Visibility and Shortest Path Problems Inside Simple Polygons. In *Proceedings of the Second Annual Symposium on Computational Geometry*, pages 1–13, Yorktown Heights, NY, June 2–4 1986. ACM Press.

[Hillier *et al.* 1983] B. Hillier, J. Hanson, J. Peponis, J. Hudson, and R. Burdett. Space syntax, a different perspective. *Architects' Journal*, 178:47–63, 1983.

[Joe and Simpson 1987] B. Joe and R. B. Simpson. Corrections to Lee's visibility polygon algorithm. *BIT*, 27(4):458–473, 1987. See Lee [1983].

[Joe 1990] B. Joe. On the correctness of a linear-time visibility polygon algorithm. *International Journal of Computer Mathematics*, 32:155–172, 1990.

[Karger *et al.* 1997] David R. Karger, Rajeev Motwani, and G. D. S. Ramkumar. On Approximating the Longest Path in a Graph. *Algorithmica*, 18(1):82–98, May 1997.

[Konidaris and Sanders 2002] G.D. Konidaris and I.D. Sanders. *Axial Line Placement in Deformed Urban Grids*. Technical Report TR-Wits-CS-2002-4, School of Computer Science, University of the Witwatersrand, April 2002.

[Konidaris *et al.* 2003] G. D. Konidaris, K. Mehlhorn, and D. A. Shell. An Optimal Algorithm for Finding Edge Segement Adjacencies in Configurations of Convex Polygons, December 2003.

[Lee 1983] D. T. Lee. Visibility of a simple polygon. *Computer Vision, Graphics, and Image Processing*, 22(2):207–221, May 1983. See corrections Joe and Simpson [1987].

[Lichtenstein 1982] D. Lichtenstein. Planar formula and their uses. *SIAM Journal of Computing*, 11(2), May 1982.

[Lingas 1982] A. Lingas. The power of non-rectilinear holes. In *The 9th International Colloquim on Automata, Languages and Programming*, number 140 in Lecture Notes in Computer Science 140, pages 369–383, New York, 1982. Springer-Verlag.

[Micali and Vazirani 1980] S. Micali and V. Vazirani. An $O(\sqrt{V}E)$ algorithm for finding maximum matching in general graphs. In *the Proceedings of the Annual IEEE Symposium of the Foundations of Computer Science,* volume 21, pages 17–25, Syracuse, 1980.

[Mirzaian 2002] A. Mirzaian. Interval cover problem. `http://www.cs.yorku.ca/~andy/courses/3101/lecture-notes/LN5.html`, 2002.

[Norman and Rabin 1959] R. Z. Norman and M. O. Rabin. An algorithm for a minimum cover of a graph. *Proceedings of the American Mathematical Society*, 10:315–319, 1959.

[O'Rourke 1981] J O'Rourke. An on-line algorithm for fitting straight lines between data ranges. *Communications of the ACM*, 24(9):574–578, September 1981.

[O'Rourke 1987] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, New York, NY, 1987.

[O'Rourke 1995] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press,Cambridge,UK, 1995.

[O'Rourke 1998] J. O'Rourke. Open problems in the combinatorics of visibility and illumination; in Advances in Discrete and Computational Geometry. `citeseer.nj.nec.com/49118.html`, 1998.

[O'Rourke 2002] J. O'Rourke. Comments on Sanders [2002], 2002.

[Pellegrini and Shor 1992] M. Pellegrini and P.W. Shor. Finding stabbing lines in 3-space. *IEEE Transactions on Computers*, 8:191–208, 1992.

[Pellegrini 1993] M. Pellegrini. Lower bounds on stabbinglines in 3-space. *Computational Geometry: Theory and Applications*, 3:53–58, 1993.

[Phillips 2001] R. Phillips. *Special Cases for Axial Line Placement in Orthogonal Rectangles*. Honours research report, School of Computer Science, University of Witwatersrand, October 2001.

[Sanders and Kenny 2001] I. D. Sanders and L. Kenny. Heuristics for placing non-orthogonal axial lines to cross the adjacencies between orthogonal rectangles. Technical Report TR-Wits-CS-2001-6, School of Computer Science, University of the Witwatersrand, August 2001. (16 pages).

[Sanders *et al.* 2000a] I. D. Sanders, D. C. Watts, and A. D. Hall. Orthogonal axial line placement in chains and trees of orthogonal rectangles. *South African Computer Journal*, 25:56–67, 2000.

[Sanders *et al.* 2000b] I.D. Sanders, D.C. Watts, and A. Hall. *Orthogonal Axial Line Placement in Chains and Trees of Rectangles*. Technical Report TR-Wits-CS-2000-8, Department of Computer Science, University of the Witwatersrand, June 2000.

[Sanders 1999] I. D. Sanders. Non-orthogonal ray guarding. In J. Snoeyink, editor, *Abstracts for the Eleventh Canadian Conference on Computational Geometry*, pages 80–83. University of British Columbia, Vancouver, August 1999. The full paper is available in the electronic proceedings – `http://www.cs.ubc.ca/conferences/CCCG/elec\_proc/elecproc.html`.

[Sanders 2002] I.D. Sanders. *The Axial Line Placement Problem*. PhD Thesis, Department of Computer Science, University of Pretoria, June 2002.

[Shermer 1992] T.C. Shermer. Recent results in art galleries. In *Proceedings of the IEEE*, volume 80, pages 1384–1399, September 1992.

[Urrutia 1999] J. Urrutia. *Art Gallery Illumination Problems, Handbook on Computational Geometry*, chapter 22. Elsevier Science, 1999.

[van Antwerpen 2002] K. van Antwerpen. *Using Duality to Place Lines through Adjacencies between Orthogonal Rectangles*. Honours research report, School of Computer Science, University of Witwatersrand, November 2002.

[Vazirani 1989] V. Vazirani. A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph matching algorithm. Technical Report TR-89-1035, Cornell University, 1989.

[Wilkins and Sanders 2004] D. Wilkins and I. D. Sanders. Axial Line Placement in Deformed Urban Grids. Technical Report TR-Wits-CS-2004-2, School of Computer Science, University of the Witwatersrand, April 2004.