TU WIEN Informatics

Exact and Heuristic Recognition of Monotone Lobster Graphs on a Grid

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Peter Neubauer
Registration Number 00725263

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Dipl.-Inform. Dr. rer. nat. Martin Nöllenburg
Assistance: Dipl.-Ing. Soeren Terziadis, B.A.

Vienna, 16th March, 2023

Peter Neubauer	Martin Nöllenburg

Erklärung zur Verfassung der Arbeit

Peter Neubauer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 16. März 2023

Peter Neubauer

Acknowledgements

I finished this thesis only thanks to the endless patience and support of my girlfriend Simone Marxer, my family and friends. I am grateful that you are here for me.

I extend my sincere thanks to Dr. rer. nat. Simon Naarmann for being interested in my thesis and proofreading it, and for always remaining an enduring friend even over long distance and low contact.

I also extend my gratitude to my supervisor Professor Dr. rer. nat. Martin Nöllenburg and project assistant Dipl.-Ing. Soeren Terziadis, whose frequent and knowledgeable feedback made my thesis into a true learning experience.

Kurzfassung

Unit Disk Graphen sind Graphen, die man in der Ebene derart zeichnen kann, dass jede Scheibe (disk) einen Knoten und der Kontakt zwischen den Scheiben die Kanten repräsentiert. Unit Disk Graphen und ihre vielfältigen Varianten erfreuen sich eines lebhaften Forschungsinteresses auf dem Gebiet der Algorithmischen Geometrie. In dieser Arbeit beschäftigen wir uns mit Erkennungsproblemen hinsichtlich Scheibenkontakt in eingeschränkten Klassen von Unit Disk Graphen, spezifisch Caterpillar und Lobster Graphen. Diese beschränkten Klassen sind, gewisse offene Fragen vorbehalten, einfacher zu erkennen als allgemeine planare Graphen und sogar Bäume.

Wir formalisieren und diskutieren den aktuellen Wissensstand über die betreffenden Problemstellungen und ihre algorithmische Komplexität. Wir stellen Annahmen vor, die unser Kernproblem erleichtern. Außerdem formulieren wir zwei Algorithmen zur Entscheidung des Erkennungsproblems, ob ein gegebener Lobster Graph eine x-monotone schwache Unit Disk Repräsentation auf einem Dreiecksraster zulässt. Zum einen handelt es sich um einen exakt entscheidenden Algorithmus mittels Dynamischer Programmierung aus der Literatur, zuzüglich ein paar Anpassungen für verbesserte Laufzeit. Zum anderen stellen wir eine schnelle, greedy Heuristik zum Vergleich vor.

Beide Algorithmen sind in dem beiliegenden Softwareprojekt umgesetzt. Wir führen mit diesem Programm eine experimentelle Analyse der vorgestellten Ansätze durch. Die Untersuchung umfasst die vollständige Aufzählung aller relevanten Lobster bis zu einer Gratlänge von 7 Knoten. Sie vergleicht die Laufzeit beider Implementierungen und die Treffsicherheit der Heuristik, wobei sich zeigt, dass die Heuristik bis zu 250× schneller läuft und, trotz abnehmender Zuverlässigkeit bei größeren Instanzen, etwa 70% der Ja-Instanzen mit Gratlänge 7 korrekt entscheidet.

Abstract

Disk contact graphs are graphs which can be drawn in the plane such that every disk represents a vertex, and disk contact represents graph edges. These graphs and their variants form are a lively research topic in the field of Algorithmic Geometry. This work focuses on recognition problems of unit disk contact on restricted graph classes, specifically caterpillars and lobsters. These restricted classes are easier to recognize than general planar graphs or even trees, with some caveats.

We formally review the related problems and their complexity, as far as we currently know. We present assumptions which make our core problem easier. Then we formulate two algorithms to decide, in particular, whether a given lobster admits an x-monotone weak unit disk contact representation on a triangular coordinate grid. One is a dynamic programming algorithm from the literature with some performance modifications, which decides the problem exactly. The other is a fast, greedy heuristic approach.

Both algorithms are implemented in an accompanying software project. We employ this program for experimental analysis of the presented approaches. The examination covers a complete enumeration of all relevant lobsters up to a spine length of 7 nodes. It regards the run time of both implementations and accuracy of the heuristic, showing that the heuristic algorithm is up to 250× faster and, though generally less reliable on larger instances, correctly decides about 70% of yes-instances with spine length 7.

Contents

CHAPTER 1

Introduction

A disk contact graph is a graph for which there exists a mapping of all vertices to disks on the plane such that, if two vertices are connected, their respective disks are in contact with each other. Since the formulation of the circle packing theorem [Koe36], we know that the set of disk contact graphs is exactly the set of planar graphs.

Various practical applications, such as wireless communication towers [Hal80], require us to consider the restricted case of *unit disk graphs* (also known as *penny graphs*), in which all disks in the graph's planar embedding are of one unit size. Figure 1.1 shows some drawings of a unit disk graph.

Not every planar graph is a unit disk contact graph. Naturally, we have to consider the associated recognition problem: Given a planar graph G, does it admit an embedding in the plane using unit disks? This problem is NP-hard [BK98]. The same complexity persists even when restricting the problem to only trees [Bow+15].

If even trees are too complex to decide in polynomial time, then where is the line of

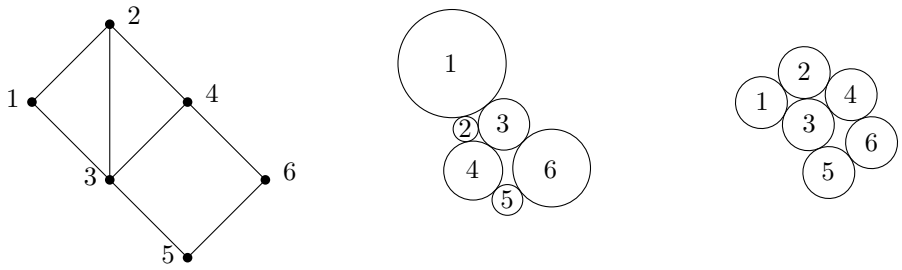

Figure 1.1: The same graph can be drawn in different ways. For example, compare the node-link diagram on the left, the disk contact representation in the middle and the unit disk contact representation on the right.

tractability? Coming at the problem from the other direction, we can identify graphs for which this problem is definitely in P.

A caterpillar is a tree that consists only of a string of connected vertices called a "spine" or "backbone", and an arbitrary number of leaf nodes connected to the spine. It is possible to decide the problem for caterpillars in linear time. Klemz et al. [KNP15] show this result under the "proper" or "strict" mode of disk contact, where the interior-disjoint disks are in contact if and only if there is an edge between their nodes. Cleve [Cle20] uses the slightly modified notion of "weak" disk contact, in which the corresponding disks may touch even between unconnected vertices. Our approach uses the "weak" disk contact notion.

We conclude that the line between graph classes for which the recognition problem is tractable and those for which it remains intractable lies somewhere between caterpillars and trees. Perhaps there is some pertinent quality about them that determines the problem's complexity.

This thesis is concerned with the complexity of the recognition problem on *lobsters*, one such in-between class. A lobster is a tree which, similar to the caterpillar, has a connected vertex string for a spine. The spine vertices may further be connected to subtrees with a depth of at most two, i.e. "expanding" the caterpillar concept by one step from the leaves.

Our main question is whether the lobster, like the caterpillar, can be recognized in linear time. We also aim to provide a practical method of constructing a unit disk contact embedding for a given instance. Additionally, we are curious to know how fast and how accurate our solution is.

It is conjectured that lobsters admit x-monotone unit disk representations, i.e. that any unit disk contact lobster can be embedded without any "u-turns" in the spine [Bho+21]. With the `udcrgen` software developed in conjunction with this thesis, we provide a tool which can empirically evaluate such x-monotone lobsters through exhaustive enumeration and testing of small lobsters.

If lobsters are indeed x-monotone and another assumption on the alignment of the embedding on a triangular grid also holds, it follows that the dynamic program implemented by `udcrgen` decides the problem in linear time [Bho+21]. Beyond the asymptotic time constraints given by theory, our implementation uses prudent shortcuts and methods to achieve good performance in practice.

Furthermore, we present a heuristic to embed x-monotone lobsters. Both approaches run in linear time, but the dynamic program requires us to consider an impractically large set of partial solutions. We examine both approaches with regards to correctness and run time.

Our results narrow a gap in our understanding of the complexities of the unit disk contact problem. They offer tools for answering this question for lobsters in particular, and a step towards further research.

The further structure of this thesis is as follows. In Chapter 2, we introduce all necessary terminology and formal definitions, building up to the embedding problems which we approach in the later chapters. Chapter 3 delves into the literature which forms the basis of this work and other interesting related works. Chapter 4 explains our reliable approach for deciding the unit disk contact recognition problem for any given lobster in linear time. It covers theoretical and implementation aspects. Chapter 5 explains our faster, but not always correct, approach to the same problem. Chapter 6 describes our empirical evaluation of the two approaches. Using the implementation written in conjunction with this thesis, we exhaustively cover lobsters up to spine length 7. We compare our algorithms with regards to run time and correctness. In Chapter 7, we summarize the results and consider open questions for future work.

CHAPTER 2

Preliminaries

Starting from the basics, this chapter introduces the definitions and notation used in this work.

2.1 Graph Classes

A graph $G = (V, E)$ is defined by its vertices $V = \{v_0, \ldots, v_n\}$ and edges $E \subseteq V \times V$. Though all our graphs are undirected, we use the edge notation (v, w) for readability without implying a direction.

A *drawing* of a graph is a representation of the vertices and edges of G in the plane. A drawing contains an implied mapping of every graph vertex to a planar x-/y-coordinate, and of every edge to a curve such that the endpoints of the curve are the planar coordinates of the incident vertices.

A graph G is *planar* if there exists a drawing of G such that none of the curves, which represent the edges, intersect or overlap. All *trees*—cycle-free connected graphs—are planar.

A *spined graph* is a tree $G = (S \dot\cup \mathcal{T}, E)$, where $S = \{s_1, \ldots, s_k\}$ is the set of *spine vertices* and $\mathcal{T}$ are the *subtree* vertices. The spine is a connected string: $(s_i, s_{i+1}) \in E$. The *depth* d of G is defined as the maximum path distance from any $t \in \mathcal{T}$ to any spine in S. With unbounded depth and unbounded spine length, spined graphs are merely trees with a path designated as the spine. However, by bounding d by a constant, we treat ourselves to a new class of graph, simpler than the general tree. Refer to Figure 2.1 for illustration.

A *caterpillar* is a spined graph $(S \dot\cup L, E)$ with depth $d = 1$, where L is the set of *leaves*: vertices at distance 1 from the spine. Every leaf $l \in L$ is connected to its parent spine vertex $p(l) \in S$: $(l, p(l)) \in E$.

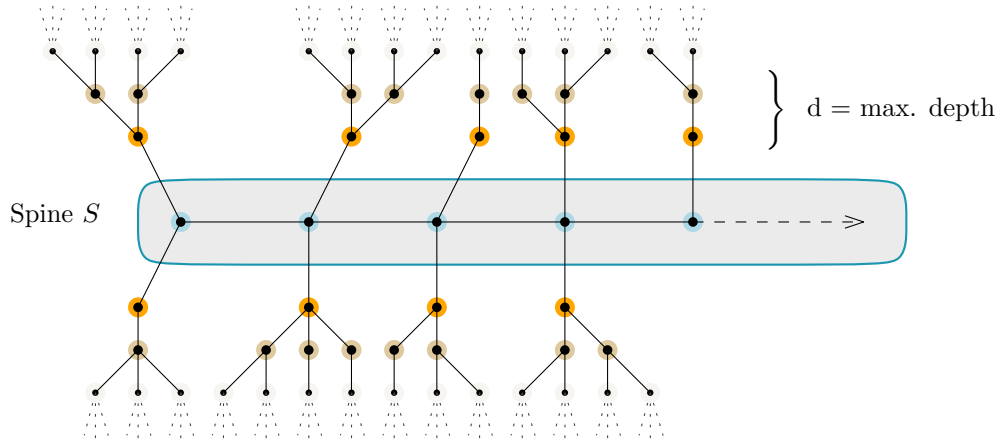

Figure 2.1: A spined graph consists of a string of vertices identified as the *spine*, here marked in blue. Connected to the spine vertices, we find the subtrees rooted in the orange vertices. All brown vertices have a distance of two to the spine.

We are now prepared to introduce the graph class which is the core subject of our examination.

Definition 1 (Lobster)**.** *A lobster is a spined graph* $(S \dot\cup B \dot\cup L, E)$ *with depth* $d = 2$*, where* B *is the set of* branches *and* L *is the set of* leaves*. Branches have distance* 1 *from the spine, leaves have distance* 2*.*

Every branch $b \in B$ *is connected to its parent spine vertex* $p(b) \in S$*:* $(b, p(b)) \in E$*. Every leaf* $l \in L$ *is connected to its parent branch* $p(l) \in B$*:* $(l, p(l)) \in E$*.*

Figure 2.2 displays a caterpillar and a lobster.

2.2 Representations

A *combinatorial embedding* (or just *embedding*) of a graph in the plane encodes the topological properties of the graph. For each vertex, the embedding defines the clockwise-ordered permutation of its neighbors. The embedding is independent of any specific

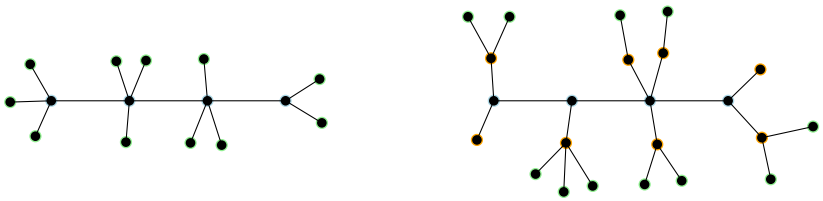

Figure 2.2: The caterpillar (left) and lobster (right) are the only kinds of spined graphs which we explore in this thesis.

planar coordinates for vertices and edges, but every drawing of a planar graph induces an embedding.

In this thesis, we define the *embedding function* d over G—not to be confused with the embedding, see above—as a bijection $d\colon V \to \mathbb{R}^2$, which maps vertices to coordinates in the plane. In the common node-link diagram representation, we would draw straight lines to represent the edges. The notation d is short for "disk" since we use it to define centers of disks.

The *Euclidean norm* $||v||, v = (v_x, v_y)$, is the length of the vector $v \in \mathbb{R}^2$, calculated as $||v|| = \sqrt{v_x^2 + v_y^2}$. We denote the distance between the points v_1 and v_2 by $||v_1 - v_2||$.

The most common notions of graph drawings represent vertices as points. Numerous alternative representations exist. We are particularly interested in drawings with vertices represented as disks.

Definition 2 (Disk Contact Representation)**.** *Let $G = (V, E)$ be a graph and d be an embedding function over G. Further let $w\colon V \to \mathbb{R}$ be a* weight function *for the vertices of G such that*

$$\begin{aligned} \|d(v_1) - d(v_2)\| &= \frac{1}{2}(w(v_1) + w(v_2)) && \text{if } (v_1, v_2) \in E \text{ and} \\ \|d(v_1) - d(v_2)\| &> \frac{1}{2}(w(v_1) + w(v_2)) && \text{otherwise.} \end{aligned}$$

Then $D = (d, w)$ is a disk contact representation *of G.*

Note that this definition uses strict inequality for non-edges. Disks of disconnected vertices do not touch. In the next definition, the inequality is relaxed. Disks may touch even if there is no edge between them.

Definition 3 (Weak Disk Contact Representation)**.** *Let $G = (V, E)$ be a graph, d be an embedding function over G and w be a weight function for V such that*

$$\begin{aligned} \|d(v_1) - d(v_2)\| &= \frac{1}{2}(w(v_1) + w(v_2)) && \text{if } (v_1, v_2) \in E \text{ and} \\ \|d(v_1) - d(v_2)\| &\geq \frac{1}{2}(w(v_1) + w(v_2)) && \text{otherwise.} \end{aligned}$$

Then $D = (d, w)$ is a weak disk contact representation *of G.*

When we want to explicitly refer to the contact notion from Definition 2, in contrast to the weak contact from Definition 3, we call it *strict* or *proper* contact. In the rest of this thesis, we mainly concern ourselves with weak contact. Thus, the weak contact notion is

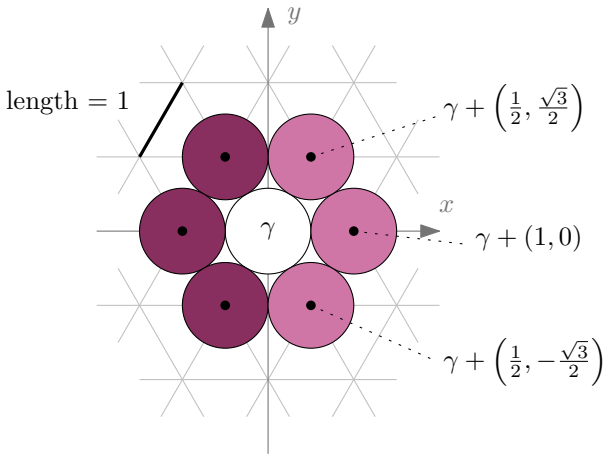

Figure 2.3: The neighborhood $\Gamma(\gamma)$ on the triangular grid contains the center coordinates of all purple disks. The x-monotone neighborhood $\Gamma^{x+}(\gamma)$ contains only the centers of the light purple disks. All grid distances are unit.

assumed as the default in every context from now on unless explicitly specified otherwise. Regardless, the further definitions involving disk contact can be used both with weak and strict notions of contact.

G is a *disk contact graph* if it admits a disk contact representation. A disk contact graph can be drawn by drawing for each vertex v a disk centered at $d(v)$ with diameter $w(v)$. The disk of v precisely touches the disks of neighbouring vertices. In a drawing using disks, there are no curves to represent the edges beyond the disks being in contact or not. Note that every strict disk contact graph is also a weak disk contact graph.

Let $G = (S \dot\cup \mathcal{T}, E), S = \{s_1, \ldots, s_n\}$ be a spined graph and $D = (d, w)$ be a disk contact representation. D is *x-monotone* if $\forall i > 1 : d_x(s_{i+1}) > d_x(s_i)$, where $d_x(v)$ is the x-component of the embedded coordinates $d(v)$. Informally, the spine "goes from left to right".

If it holds for the weight function w of a disk contact representation of $G = (V, E)$ that $\forall v \in V : w(v) = 1$, then we call it a *unit disk contact representation*, shortening "unit disk contact" to UDC for readability. G is a UDC graph or UDCG.

Let $D = (d, w)$ be a weak disk contact representation in which d maps all vertices to points from the set $\{(x + \frac{y}{2}, \frac{\sqrt{3}}{2}y) \mid x, y \in \mathbb{Z}\}$. This set describes the intersection points of a *triangular grid* in the plane, and we call D a *tri-grid representation*. Every such grid point γ has a *neighborhood*. Using the notation $(\cdot)_{\mathbb{R}^2} \colon \mathbb{C} \to \mathbb{R}^2$ for the bijection $(x + yi)_{\mathbb{R}^2} = (x, y)$, we define the neighborhood of γ as the set of points offset from γ by the sixth unit roots in $\mathbb{C}$.

$$\Gamma(\gamma) = \{\gamma + (x + yi)_{\mathbb{R}^2} \mid (x + yi)^6 = 1\}$$

The more restricted *x-monotone neighborhood*

$$\Gamma^{x+}(\gamma) = \{\gamma + (x + yi)_{\mathbb{R}^2} \mid (x + yi)^6 = 1 \wedge x > 0\}$$

includes only those neighbors with a larger x-coordinate than γ. Figure 2.3 illustrates both.

The triangular grid lends itself to UDC representations because the distance between two neighbor coordinates on the triangular grid is exactly 1. Consequently, unit disks embedded at neighboring tri-grid points are in contact. So, if $G = (V, E)$ is a disk contact graph, $D = (d, w)$ is a UDC representation, $v \in V, \gamma = d(v)$ and $(v, u) \in E$, then $d(u) \in \Gamma(\gamma)$.

2.3 Problems

A *recognition problem* is a computational problem in which the goal is to find an algorithm that decides, for a given input from a larger domain, whether that input belongs to a certain smaller class or set in the domain. We say that the algorithm *recognizes* the set.

The broadest problem that we are interested in discussing here is, in the following two variants:

Problem 1 (Strict UDC Recognition)**.** *Given a graph G, does G admit a strict UDC representation?*

Problem 2 (Weak UDC Recognition)**.** *Given a graph G, does G admit a weak UDC representation?*

These problems are successors to the original disk contact problem, answered by the above mentioned circle packing theorem. Now restricted to unit size disks, solutions to these problems already exist for the restricted graph class of caterpillars [KNP15; Cle20], which are discussed in Chapter 3.

As promised in the introduction, we now tie into current research by focusing our attention on the subclass of lobsters. Under the current state of knowledge, we have no algorithm to answer Problems 1 or 2 in polynomial time, even when the input is restricted to lobsters. We can, however, make some concessions in the form of assumptions to pull the problem into our analytic reach [Bho+21].

Problem 3 (Tri-Grid X-Monotone UDC Recognition for Lobsters)**.** *Given a lobster G, does G admit a weak x-monotone UDC representation on the triangular grid?*

Unit disks on the triangular grid are packed decently dense. Disk packing is an entirely own field of research. It is not obvious that every lobster which admits a weak UDC also admits a weak UDC on the triangular grid. This remains an open question. Regardless,

the convenience of the triangular grid is "unlocked" by and motivates our use of the weak contact notion.

Likewise, we allow ourselves the restriction to x-monotone solutions based on the unproven assumption that a given lobster G admits an x-monotone UDC representation if it admits any UDC representation[1]. As there is no combinatorial embedding prescribed for G, no branch necessarily has to be above or below the spine. Yet, to refute our x-monotonicity assumption, a counter-example lobster would have to have some configuration of branches and leaves such that all its possible UDC representations enforce either an acute ($60°$) "bend" in the spine, or two consecutive obtuse ($120°$) bends in the same direction. By superficial experimentation, no such configuration has been found so far. The lobster drawn in Figure 2.4 does admit a tri-grid x-monotone representation.

If our assumptions should hold, then solving Problem 3 is equivalent to solving Problem 2. Hence we focus our aims on Problem 3.

With the subject problem properly outlined, we explore the current state of research on the following questions:

Question 1: Can we decide the weak UDC recognition problem for lobsters of size n in time $O(n)$?

Question 2: Can we decide the tri-grid x-monotone UDC recognition problem for lobsters of size n in time $O(n)$?

[1] This assumption is the same as in the the proof of linear time for monotone weak UDCs by Bhore et al. [Bho+21].

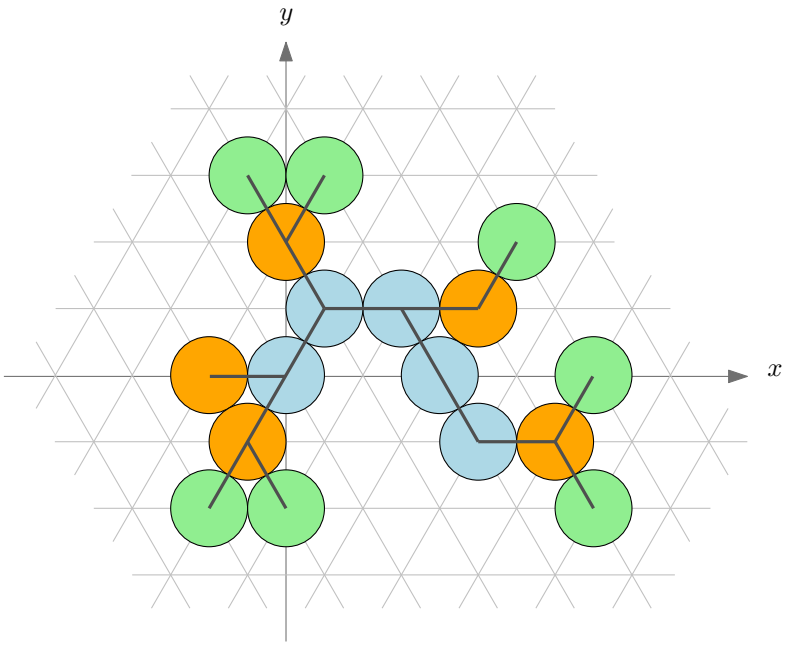

Figure 2.4: A tri-grid x-monotone UDC representation of a lobster. Spine, branch and leaf vertices are mapped to the blue, orange and green disks respectively. The spine, although it bends twice, is embedded on strictly increasing x-coordinates.

CHAPTER 3

Related Work

The aforementioned circle packing theorem [Koe36] should be considered as a fundamental result from which various research into contact graphs derives. We can categorize the topic by different aspects. Consider as an exemplary overview:

- by graph class: planar, outerplanar, tree, lobster, caterpillar, ...
- by embeddedness: free (only the graph itself is given), embedded (an embedding is prescribed)
- by shape: disk, (rectilinear) polygon and many more
- by notion of contact: intersecting, strict, weak
- by problem/question and associated complexity

Just to illustrate some of the topical variations being researched, consider these examples. Any 4-connected, internally triangulated graph admits a contact representation using rectangles [KK85]. For some shapes, like rectilinear polygons, the problem is not so much in the representation of any planar graph under the shape constraints. Instead, we want to know the maximum complexity of the shape that is necessary to represent the graph. If the graph is planar triangulated, it admits a weighed contact representation using rectilinear polygons with at most 8 vertices: "T-shapes" [Ala+13].

Breu and Kirkpatrick [BK98] showed the first complexity result that specifically applies to unit disk graphs. They introduce the topic by showing that recognition of *intersection* graphs, in which adjacent disks' interiors may overlap, is NP-hard. They also state that their results extend to unit disk contact graphs, which are thus also NP-hard to recognize in general. Hardness of the recognition problem on intersection graphs does not stop in NP. McDiarmid and Müller [MM13] proved that there exist unit disk intersection graphs

which do not have a polynomial certificate—required in NP—due to the exponential number of bits required to represent an embedding function for them. The problem is complete for the class $\exists\mathbb{R}$ (existential theory of reals), even harder than NP.

On the other hand, a previous result by Klemz et al. [KNP15] that the recognition problem for caterpillars under the strict contact notion can be decided in linear time has since been weakened to a conjecture due to a missing proof for one of its steps [KNP22]. This result is based on an observation that if two inner spine vertices of degree 5 are not separated by a spine vertex with degree at most 3 between them, the leaves are simply "too many" and will not fit. It is then quite easy to check whether or not a particular caterpillar has this property.

With the findings of Cleve [Cle20], the author turns his attention to the weak contact notion and the triangular grid. Under this notion, caterpillars can definitely be recognized in polynomial time[1], and the recognition problem remains NP-hard for trees.

We continue from these results with this thesis, going hand in hand with a new paper by Bhore et al. [Bho+21]. They take the next step from the caterpillar to the slightly more complex lobster. Our particular attention concerns the proof that, assuming the x-monotonicity and triangular grid layout, these graphs can also be recognized in linear time. The method is more difficult than for caterpillars, as it requires a constructive algorithm with a dynamic programming approach—the same approach which we discuss in Chapter 4 and implement.

Interestingly, the recognition problem becomes harder if an embedding is prescribed in addition to the graph itself. It is even NP-hard to recognize embedded unit disk contact caterpillars under weak contact [CCN19]. The given embedding robs us of our choice to place the subtrees such that the spine cannot "turn in on itself". While the complexity of strict UDC for trees is yet unknown, if the tree is embedded, it remains NP-hard [Bow+15].

[1] Although the original source claims linear time, this proof has since been retracted. Instead, caterpillars can be recognized in quadratic time at worst (Jonas Cleve, personal communication to Soeren Terziadis).

CHAPTER 4

Dynamic Program

We describe two algorithms that aim to decide Problem 3 on a lobster instance of size n in time linear in n. The *dynamic program* is an implementation of the algorithm conceptualized in the literature [Bho+21], with a few refinements for practical performance. It solves Problem 3 exactly, i.e. if and only if the lobster admits a tri-grid x-monotone embedding, the dynamic program decides on "yes" as its output.

Dynamic programming is a technique whereby a problem is solved by dividing it into smaller sub-problems and combining their solutions. A key feature of this approach is the reuse of solutions to sub-problems which emerge in different problem divisions.

4.1 Problem Definition

To apply the dynamic programming idea to Problem 3, we must extend its definition to that of the *partial recognition problem* which we introduce in this section.

Let $G = (V, E), V = S \dot\cup B \dot\cup L$ be a lobster.

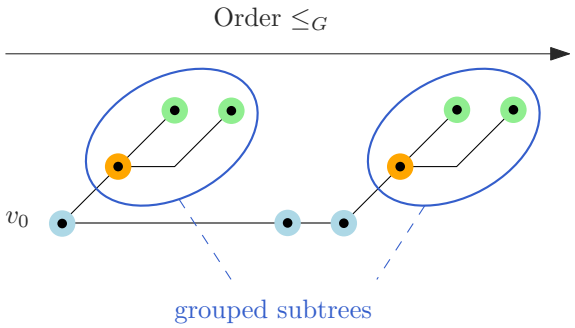

Figure 4.1: An admissible order over V, from left to right.

The *embedding order* $\leq_G$ for G is a total order over V. It dictates the order in which the algorithm assigns coordinates to vertices in its exploration of the partial problem space. Starting from the initial problem, where no vertices are embedded, the minimum vertex under $\leq_G$ is assigned coordinates first. The final coordinate assignment to the maximum vertex implies an affirmative answer to the original problem instance.

An embedding order $\leq_G$ is *admissible* if it fulfills the following criteria.

- $\leq_G$ is a total order. It is reflexive, transitive and antisymmetric.
- $\leq_G$ orders the spine.
  - There exists a minimal $v_0 \in S$, i.e. $\forall v \in V : v_0 \leq_G v$, which is connected to at most one other spine.
  - Let $s_1 \neq s_2 \neq s_3 \in S, s_1 \leq_G s_2, s_1 \leq_G s_3$ and $(s_1, s_3) \in E$. Then $s_3 \leq_G s_2$.
- $\leq_G$ groups branches together with their parent spine. Let $s_1, s_2 \in S, b \in B, s_1 = p(b)$ and $s_1 \leq_G s_2$. Then $b \leq_G s_2$.
- $\leq_G$ groups subtrees together. Let $l \in L, b = p(l) \in B, v \in V, (b, v) \notin E$ and $b \leq_G v$. Then $l \leq_G v$.
- Parent vertices come first. Let $v_1, v_2 \in V$. If $v_1 = p(v_2)$, then $v_1 \leq_G v_2$.

Figure 4.1 shows a possible embed order $\leq_G$, which adheres to these criteria.

For a given lobster, there may be multiple possible admissible orders. Whichever one we choose, the dynamic program requires that the chosen order definition remains fixed during the execution of the algorithm across all partial problem instances.

The *depth* δ of a partial problem is the number of embedded vertices, i.e. the δ smallest vertices under $\leq_G$. It takes δ subdivisions from the initial partial problem to arrive at a (set of) depth δ problems.

Let $V = V_d \dot\cup V_r$ be a bipartition of V ordered under an order $\leq_G$, where the *prefix* V_d contains the smaller vertices until some cutoff vertex and V_r contains the remaining larger vertices:

$$\forall v_d \in V_d, v_r \in V_r : v_d \leq_G v_r.$$

We call V_d the set of "embedded" vertices, meaning that we have, before fully defining the entire embedding function $d\colon V \to \mathbb{R}^2$, already decided on some $d(v)$ for every $v \in V_d$. V_r are then the remaining vertices yet to be embedded. For a partial problem with depth δ, $|V_d| = \delta$.

The *spine head* γ_s is the coordinates of the last embedded spine vertex and the *branch head* γ_b is the coordinates of the last embedded branch vertex. At depth δ,

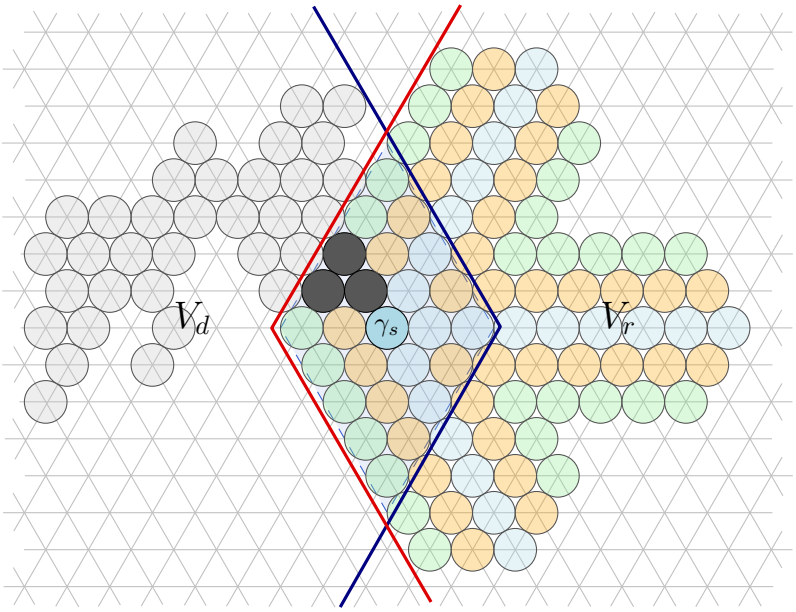

Figure 4.2: Motivation for the fundament F. We can imagine vertices from V_d embedded in parent problems, illustrated here in translucent grey. Future V_r coordinates may lie anywhere to the right, with some exemplary coordinates shown for the spine laid out only upwards, downwards or straight. The possible areas for past and future vertex coordinates are delineated in blue and red respectively, and the intersection of the areas is the fundamental neighborhood $\Phi(\gamma_s)$. Vertices from V_d within the fundamental neighborhood form the fundament, illustrated in solid grey. We must keep track of them for potential conflicts in coordinate assignment.

$$\gamma_s = \max_{\leq_G}(v \in V_d \cap S) \text{ and}$$
$$\gamma_b = \max_{\leq_G}(v \in V_d \cap B).$$

Let γ be the tri-grid embedding coordinates of a spine vertex. The *fundamental neighborhood* of a partial problem

$$\Phi(\gamma_s) = \left\{ \gamma_s + \left(x + \frac{y}{2}, \frac{\sqrt{3}}{2} y \right) \middle| \, |x + y| \leq 2; x, y \in \mathbb{Z} \right\}$$

is a region of interest for avoiding potentially conflicting assignments of coordinates to vertices in V_r. It defines a "perceivable range" inside of which the algorithm remembers the embedding coordinates of vertices in V_d and are also potential embedding coordinates of vertices in V_R.

The *fundament* $F \subseteq \Phi(\gamma_s)$ is a subset of local grid locations which are unavailable due to being reserved for vertices in V_d. Figure 4.2 visualizes this reasoning.

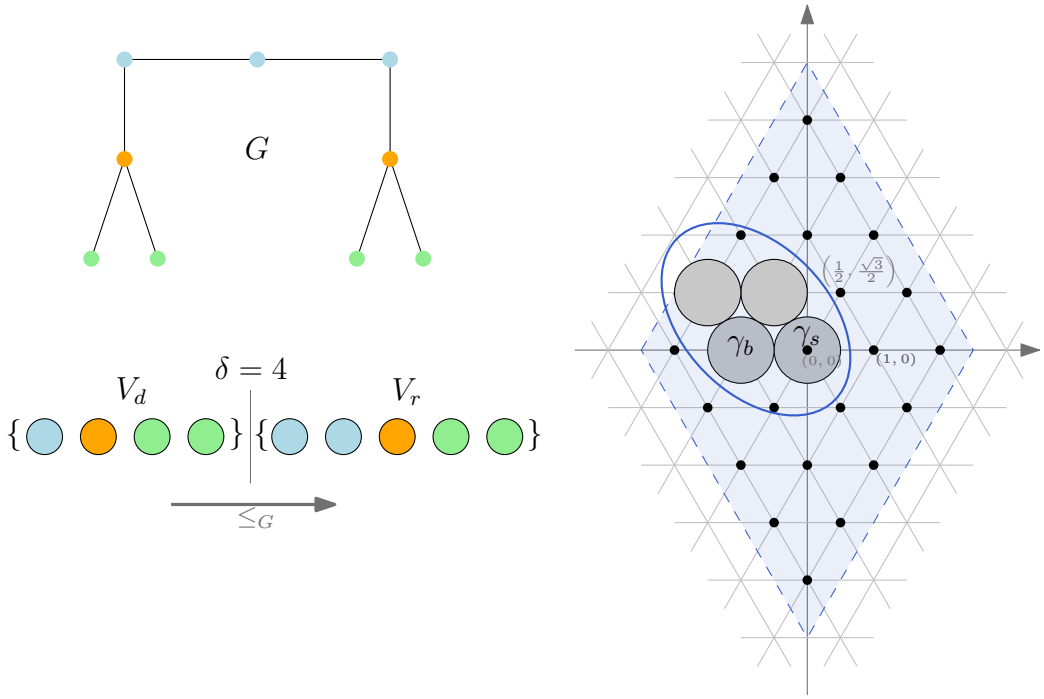

Figure 4.3: An instance of Problem 4. The task is to decide the lobster depicted in the top left as a node-link diagram. Its vertices, illustrated as disks in the bottom left, are partitioned into the prefix V_d and the remainder V_r, with the second spine vertex to be embedded next. On the right side, the fundamental neighborhood $\Phi(\gamma_s)$ is marked with dots on the coordinates and enclosed with a dashed line. The coordinates contained in the fundament F are represented by grey disks, marked with the blue ellipse. These locations are earmarked for the embedding of the vertices in V_d and no longer available for any vertex in V_r.

The *signature* ς of a partial problem is the tuple $(\delta, \gamma_s, \gamma_b, F)$, specifying the depth, heads and fundament of the problem.

Problem 4 (Partial UDC Recognition for Lobsters)**.** *Given a Lobster G, an embedding order $\leq_G$ and a signature ς, does G admit a partial tri-grid x-monotone embedding of the remaining vertices under the constraints defined by ς?*

Refer to Figure 4.3, which visualizes a partial problem instance, for an illustration of all the above concepts.

We are now prepared to define the dynamic programming algorithm on the extended problem definition.

If we can construct any partial problem instance π at the maximum depth $\delta = |V|$, there are no further vertices to embed and the answer is always “yes” (the input lobster

Input: lobster $G = (V, E), V = S \dot\cup B \dot\cup L$
Output: "yes" if G admits a tri-grid x-monotone embedding, "no" otherwise
$n \leftarrow |V|$;
$\mathsf{V} \leftarrow \texttt{sort}(V)$; `// establish some admissible order` $\leq_G$
$\mathsf{S}_0 \leftarrow \{\text{initial signature: } (\delta = 0, \gamma_s = \mathsf{undef}, \gamma_b = \mathsf{undef}, \mathsf{F} = \emptyset)\}$;
for $\delta = 1$ **to** n **do**
  $\mathsf{S}_\delta \leftarrow \bigcup_{\varsigma \in \mathsf{S}_{\delta-1}} \texttt{divide}(\mathsf{V}, \varsigma)$;
if $\mathsf{S}_n = \emptyset$ **then**
  return *"no"*;
else
  return *"yes"*;

Algorithm 4.1: The simplified dynamic program. This scheme does not implement the run-time efficiency improvements discussed later in this chapter, but it serves to illustrate the dynamic programming formula and the reasoning behind its linear-time complexity.

does admit a tri-grid x-monotone embedding). From the set S_δ of all partial problem instances at depth $\delta < |V|$ reachable by subdivision, the answer is "yes" if there exists a sub-problem yes-instance of some $\pi \in \mathsf{S}_\delta$, and "no" only if no descendant instance reaches the maximum depth. We therefore subdivide partial problems until we find a maximum-depth instance or until the search space is exhausted. The algorithm will be defined such that the combined assignments of coordinates during division of the instance define a total embedding function $d(G)$.

Algorithm 4.1 shows an illustrative implementation. Unlike this pseudo-code, the common conception of a dynamic program involves a "combination" of solved sub-problems into a solution of the super-problem. Because of the above conditions on yes- and no-instances, we can short-circuit the recombination step. See Subsection 4.3.1 below.

Algorithm 4.2 describes the subroutine by which descendant partial problems are derived. It returns fewer than six partial embedding sub-instances (with the negligible sole exception of $\delta = 1$).

Figure 4.4 shows how the dynamic program might arrive at a decision given the same lobster as in Figure 4.3. The arrows represent `divide` operations, yielding different paths forward. The colored disk marks the chosen candidate for the current embedding coordinates. At each point, the local knowledge of the fundament exactly suffices to prevent us from considering invalid duplicate assignments of coordinates.

Given a lobster $G = (V, E)$, we can construct an initial partial UDC problem signature ς as illustrated in Algorithm 4.1. The dynamic programming algorithm answers Problem 3 on G.[Bho+21]

```
Function divide(V, ς)
    Input: ordered vertices V = S ∪̇ B ∪̇ L, signature ς = (δ, γ_s, γ_b, F)
    Output: Set of signatures S′
    v ← V[δ]
    if δ = 0 then                                   // first vertex ⟹ v ∈ S
        γ′_s ← (0, 0);
        F′ ← {γ′_s};
        return {(1, γ′_s, undef, F′)};
    S′ ← ∅;
    if v ∈ S then                                   // spine vertex
        K ← Γ^{x+}(γ_s) \ F;
        foreach κ ∈ K do
            F′ ← (F ∩ Φ(κ)) ∪ {κ};
            ς ← (δ + 1, κ, undef, F′);
            S′ ← S′ ∪ {ς};
    else if v ∈ B then                              // branch
        K ← Γ(γ_s) \ F;
        foreach κ ∈ K do
            F′ ← F ∪ {κ};
            ς ← (δ + 1, γ_s, κ, F′);
            S′ ← S′ ∪ {ς};
    else                                            // leaf
        K ← Γ(γ_b) \ F;
        foreach κ ∈ K do
            F′ ← F ∪ {κ};
            ς ← (δ + 1, γ_s, γ_b, F′);
            S′ ← S′ ∪ {ς};
    return S′;
```

Algorithm 4.2: Subdivision of partial problem instances. We deal with partial problems in terms of their signatures, which contain the distinguishing properties of the instance. Dividing an instance consists of assigning the next vertex in order to one of the *candidate coordinates* in K relative to the appropriate (spine- or branch-) head. Each possibility becomes a sub-instance. We maintain the fundament F and the heads γ_s and γ_b as appropriate, depending on the type of assigned vertex. Recall that $\Gamma(\gamma)$ and $\Gamma^{x+}(\gamma)$ define the tri-grid (x-monotone) neighborhood of γ.

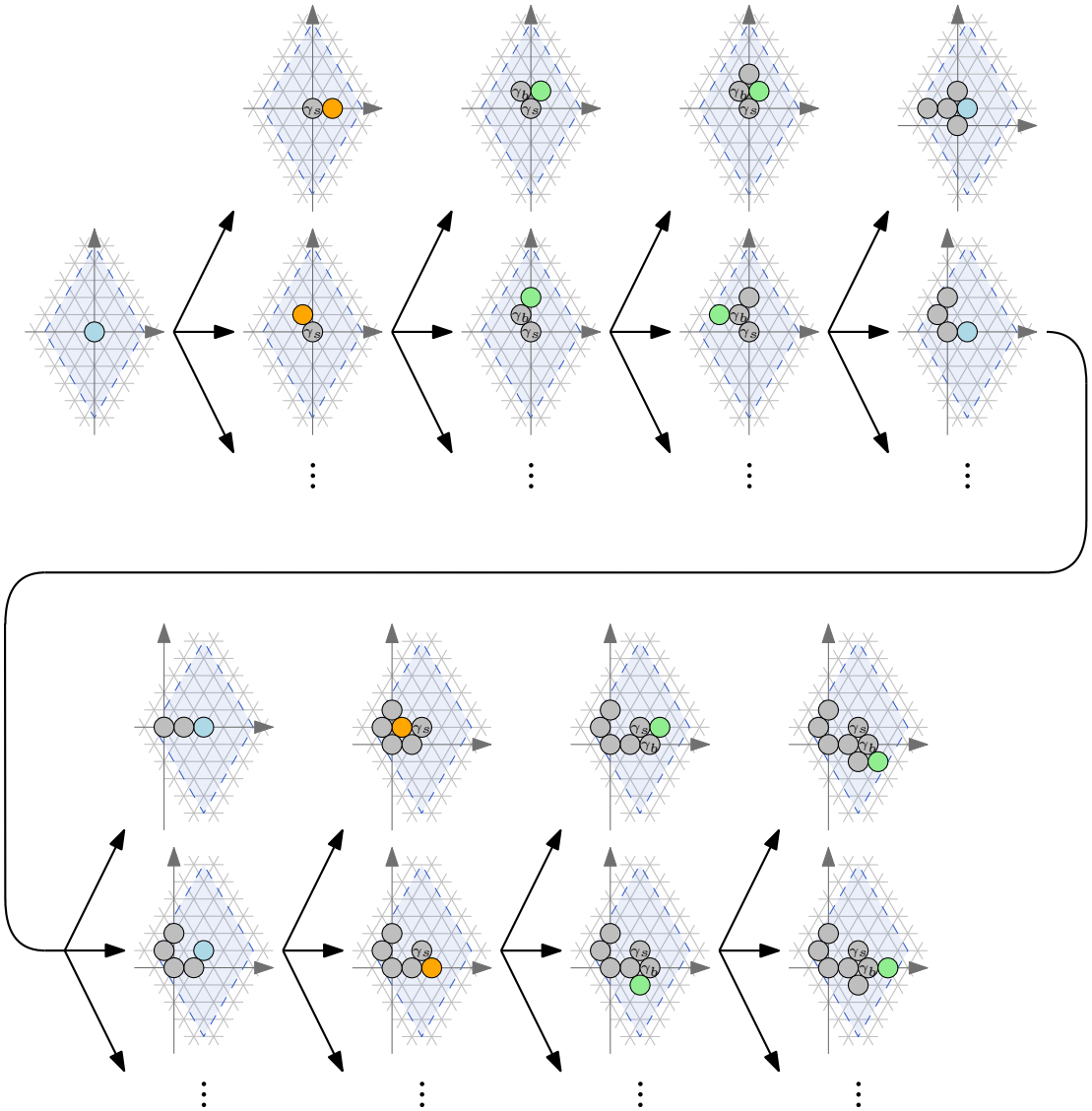

Figure 4.4: This might be the path through the search space visited by the dynamic program to solve a lobster. Each of the small drawings represents one signature. The fundamental neighborhood $\Phi(\gamma_s)$ is enclosed in dashed blue. The disk of the most recently assigned vertex bears the color of its type: blue for spines, orange for branches and green for leaves. Gray disks represent coordinates in the fundament F. The spine and branch heads γ_s and γ_b are also labeled, if relevant and not implied by color. The view always centers on γ_s, the center of $\Phi(\gamma_s)$. The black arrows signify construction of the partial problem associated with the target signature by subdivision of the source problem.

4.2 Complexity

Theorem 1. *Let $G = (V, E)$ be a lobster with $|V| = n$. Algorithm 4.1 runs in time $O(n)$.*

Proof. The determination of $\leq_G$ and the ordered vertex set V is possible in time $O(|V|)$ simply by traversing G from a suitable starting spine v_0 of our choice (front or back).

The algorithm constructs an initial partial UDC problem signature, from which all sub-problem signatures are derived. Each problem is either `divide`-d further or outright solved in constant time simply by having maximum depth. Also, due to the memory of problems already encountered, no problem is processed more than once. It therefore suffices to show that the number of signatures under consideration is bound by $O(n)$.

In fact, the size of the set of signatures S_δ that may possibly occur at any particular depth $0 \leq \delta \leq n$ is bound, due to our representation of signatures, by a constant. Consider the constituent components of the signatures at depth δ.

- The spine head γ_s does not contribute to the quantity of instances. The problem is translation-invariant. We can always derive an equivalent instance by translating γ_s to the origin point, and γ_b and F with it.
- The branch head γ_b is relevant only when the vertex to be embedded next is a leaf, due to the subtree grouping property of $\leq_G$. Even in that case, its position is limited to at most 6 coordinates relative to γ_s.
- F is limited to be a subset of $\Phi(\gamma_s)$ and therefore to the constant amount of $2^{|\Phi(\gamma_s)|} = 2^{25}$ values.

Since all constituent components of signatures at depth δ are limited to a constant-bound set of values and we discard duplicate signatures, the number of signatures at depth δ is also constant-bound and the total number of sub-problems is in $O(n)$. A single problem (signature) takes a constant amount of time to handle, putting the total run time in $O(n)$ as well. □

This result superficially seems to declare Problem 3 as tractable. However, the constant bound C for the number of equi-depth instances is too high to ignore for practical applications. On its face, it may be as high as

$$C = 5 \cdot 2^{24} = 83,886,080$$

In this first estimate, we consider 5 locations for the branch head relative to the spine head and the spine disk before it, which eliminates one coordinates option. At depth $d > 0$, F always contains γ_s, leaving us with just 24 potentially unavailable coordinates.

4.3 Run-Time Improvements

Consider the high constant bound C of sub-problem signatures at some arbitrary depth. Upon further consideration of the algorithm definition, not every value from the domain of γ_b and F can occur in real problems. Our estimate for the value of C should drop accordingly. We also discuss several improvements to the algorithm which reduce the run time in practice. Some of them are based on recognizing *equivalence* of signatures.

Let π_1 with signature $\varsigma_1 = (\delta, \gamma_s^1, \gamma_b^1, F^1)$ and π_2 with signature $\varsigma_2 = (\delta, \gamma_s^2, \gamma_b^2, F^2)$ be two partial recognition problem instances on the lobster $G = (V, E), |V| = n$. Let $V_r \subseteq V$ be the vertices remaining without coordinates assignment—equivalently, the largest $n - \delta$ vertices under the order $\leq_G$.

π_1 and π_2 are *equivalent* if and only if for any witness embedding function $d_1 \colon V_r \to \mathbb{R}^2$ that certifies a yes-answer to π_1, we can transform it into a yes-witness d_2 for π_2 and vice-versa by translation and mirroring of the coordinates.

The global *breadth* $\overline{m}$ of the graph class of lobsters is the maximum number of equivalence classes that occur at any depth over all lobsters by application of Algorithm 4.1. This is important because the run time in practice is bound by $\overline{m} \cdot n$.

The improvements described here feature in Algorithm 4.3, which corresponds to the actual implementation used for our experiments in Chapter 6.

4.3.1 Early Exit

Our definition of the partial problem includes that the combined answer is "yes" iff any sub-problem answer is "yes". Once we find a positive instance, we can instantly skip over all other sub-problems and declare the original decision problem instance solved in the affirmative.

We encourage the fast determination of this result by always prioritizing sub-problems of higher depth before sub-problems of lower depth. We augment Algorithm 4.1 with one lookup data structure of encountered ("closed") signatures at every depth and a priority queue of pending ("open") signatures. This allows us to customize the order of processed signatures while retaining the dynamic program benefit of not duplicating computations.

The early exit strategy does not improve the time to solve a negative instance. The algorithm is forced to check every possibility until it confirms the negative answer.

4.3.2 Mirror Instance

Let π_1 and π_2 be two problem instances with signatures which differ only in the branch heads γ_b^1 and γ_b^2 and in the fundaments F_1 and F_2. Without loss of generality, we assume that $\gamma_s^1 = \gamma_s^2 = (0, 0)$. Let $m((x, y)) = (x, -y)$ be the function that mirrors coordinates along the x-axis. Then π_1 is equivalent to π_2 if $\gamma_b^1 = m(\gamma_b^2)$ and $F_1 = \{m(\gamma) \mid \gamma \in F_2\}$. This effectively cuts $\overline{m}$ in half.

4.3.3 Reachability

Define the *reach* R of the partial problem instance π as the set of all possible future embedding coordinates—in other words, the union of all candidate sets K of all descendent sub-problem instances.

Then the unreachable coordinates $\Phi(\gamma_s) \setminus R$ have no more bearing on the answer of the problem. We may as well disregard all unreachable coordinates in $\Phi(\gamma_s)$ by just assuming that they are in the fundament, i.e. unavailable.

Construct an instance π' equal to π, except for the fundament $F' = F \cup (\Phi(\gamma_s) \setminus R)$. Then π is equivalent to π', and so are any other instances that differ only in the availability of unreachable coordinates.

4.3.4 Domination

Let π_1 and π_2 be two otherwise equal problem instances with fundaments F_1 and F_2. If $F_1 \subseteq F_2$, then π_1 allows as many or more options for future coordinates candidates of vertices in V_R than π_2. π_1 is easier to recognize than π_2, and we say that π_1 *dominates* π_2.

We can use the resulting hierarchy of instances to instantly disregard a sub-problem under consideration if, looking at the memory of known solutions with equal depth and equal $(\gamma_b - \gamma_s)$, we find that we have already decided a dominating instance as a "no". Attempting the harder instance will never result in a "yes"-answer.

```
Input: lobster G = (V, E), V = S ∪̇ B ∪̇ L
Output: "yes" if G admits a tri-grid x-monotone embedding, "no" otherwise
Data: Priority queue of open signatures S
Data: Set of closed signatures C
n ← |V|;
V ← sort (V) ;            // establish some admissible order ≤_G
S ← new Queue (priority_predicate) ;    // prefer higher depth
C[1], ..., C[n] ← ∅;
S.insert (initial signature: (δ = 0, γ_s = undef, γ_b = undef, F = ∅));
while ¬S.empty() do
    ς ← S.pop ();
    if ς.δ = n then
        return "yes" ;                                  // early exit
    foreach ς' ∈ divide (V, ς) do
        apply_reachability (ς');
        demirror (ς');
        d ← ς'.δ;
        if ¬C[d].contains_dominating(ς') then
            S.insert (ς');
            C[d].insert (ς');
return "no";
```

Algorithm 4.3: The improved dynamic program. The open queue S prioritizes promising signatures, while the closed sets C block duplicates. The algorithm preprocesses signatures to combine equivalent problems.

CHAPTER 5

Heuristic

We present a fast approach to decide problem 3 on a lobster instance of size n. Like the dynamic program from the previous chapter, it runs in time linear in n. The basic idea, to impose an order on the graph nodes and pick embedding coordinates step by step, remains the same. However it eliminates the large constant factor in the number of sub-problems considered.

This is a greedy algorithm: Instead of branching into different sub-problems, we immediately commit to a final placement decision at every step. This decision is based only on information that is obtainable in a short constant amount of time. Through this shortcut, we process only one sub-problem at each depth $0 \leq d \leq n$.

The price for the resultant speedup is correctness. In some cases, the heuristic decision prematurely commits us to embed a vertex at coordinates that are incompatible with any valid embedding, even if some valid embedding exists. The heuristic algorithm yields false negative answers in these cases.

5.1 Algorithm Definition

The heuristic algorithm decides problem 3 on a lobster $G = (V, E)$ by incrementally constructing a series of partial embedding functions $d_0, \ldots, d_n \colon V \to \mathbb{R}^2$.

We reuse some concepts from the dynamic programming approach as defined in Section 4.1. Again, the algorithm assigns coordinates to vertices in a globally determined admissible embedding order $\leq_G$. We also call this the *depth-first* order, explicitly denoted as $\leq_G^D$, to distinguish it from the variant in Section 5.3. When we construct d_{i+1} from d_i, we refer to the bipartition $V = V_d \dot{\cup} V_r$. The prefix V_d is the same as the domain of the partial embedding function d_i. We consider V to be an ordered set, explicitly referred to as $V_{\leq_G}$, and denote by v_i the i-th vertex in the ordered set.

The algorithm starts from $d_0(v)$ defined on the empty domain, with no coordinates assignment to any v. It iteratively derives the successor function d_{i+1} by considering v_{i+1}, the next vertex in order. The partial function expands to include the embedding coordinates for v_{i+1} determined by the heuristic function:

$$d_{i+1}(v) := \begin{cases} \texttt{heuristic}(v, d_i) & \text{if } v = v_{i+1}, \\ d_i(v) & \text{otherwise.} \end{cases}$$

The result is "yes" if the algorithm completes the embedding function $d = d_n$, and "no" if the heuristic function fails to find valid coordinates for some v_i.

5.2 Heuristic Function

Given the vertex v_{i+1} and partial embedding function d_i, the heuristic function selects the next embedding coordinates

$$\kappa \in \Gamma(d_i(p(v_{i+1}))) \setminus \{\gamma \mid \exists v : d_i(v) = \gamma\},$$

where $p(v_{i+1})$ is the parent vertex of v_{i+1}. In other words, κ represents a disk adjacent to the disk of the parent vertex which does not overlap any previously selected coordinates. The heuristic function should attempt to pick the most promising of the at most 5 candidates in this set.

Our design goals for a heuristic which produces promising candidates are:

1. Spine disks should, within the constraints of x-monotonicity, be appended such that there is as much free space around them as possible. The relevant area around them extends up to two units of distance, where leaves may be assigned.
2. Branch and leaf disks should not block vital coordinates for later vertices. Since we know that the spine will be x-monotone, going in some forward direction, it makes sense to embed the current vertex far back to keep it out of the way.
3. Bends in the spine might be unavoidable. Yet, by placing branches on both sides in a balanced way, the spine naturally flows in the in-between forward direction. This avoids conflicts with the x-monotonicity restriction.
4. A branch should always have space to fit its leaves in the neighborhood. Knowing that fewer blocked coordinates and therefore more free space can be expected in the forward direction, we should avoid placing a branch so far back that the surrounding space will be obviously insufficient.

Before we define criteria realizing these design goals, we need some supporting concepts.

Let γ be a tri-grid x-/y-coordinate. The set of *two-step neighbors*

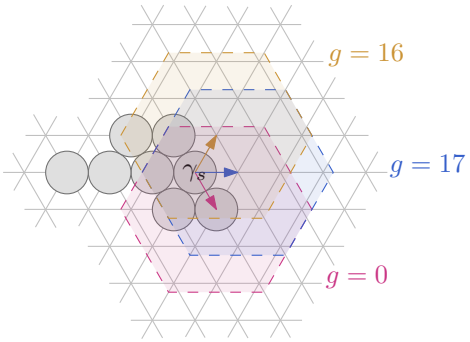

Figure 5.1: The principal direction is chosen from one of three x-monotone options, here illustrated in different colors. The arrows represent the direction α. The colored circles cover the two-step neighborhood $\Gamma^2(\gamma_s + \alpha)$. From the previously assigned coordinates in d_i—the grey disks—within that area follows $g = g(\alpha)$.

$$\Gamma^2(\gamma) = \bigcup_{c \in \Gamma(\gamma)} \Gamma(c) \setminus \{\gamma\}$$

is the set of all tri-grid coordinates at which another vertex v of graph distance ≤ 2 to γ could generally be placed. These are all vertices v such that, for any lobster $G = (V, E)$, $(\gamma, v) \in E$ or $\exists u : (\gamma, u) \in E \wedge (u, v) \in E$.

Let d_i be a partial embedding function which assigns coordinates to the first i vertices in $V_{\leq_G}$. We define the *free space* $f(i, \cdot)$ as a function counting unassigned tri-grid coordinates in d_i. Free space may apply to a particular x-/y-coordinate a or a set of coordinates A.

$$f(i, a) = \begin{cases} 1 & \text{if } \nexists v_k \in V_{\leq_G} : k \leq i \wedge d_i(v_k) = a \\ 0 & \text{otherwise} \end{cases}$$
$$f(i, A) = \sum_{a \in A} f(i, a)$$

We represent a *direction* by a unit vector at angle β relative to the x-axis, denoted with $\overrightarrow{e}(\beta) = (\cos\beta, \sin\beta)$.

Let $d_i, i > 0$ be a partial embedding function and $\gamma_s = d_i(v_j) = (x_s, y_s)$ be the coordinates of the maximum embedded spine vertex v_j, with index $j \leq i$. Then $\alpha \in \{\overrightarrow{e}(-60°), \overrightarrow{e}(0°), \overrightarrow{e}(60°)\}$ is the *principal direction* in which we want to extend the spine.

The principal direction is determined by the free space around $\gamma_s + \alpha$, weighed by distance according to a weighting function $g(i, \alpha)$. Refer to Figure 5.1 together with the following definition of g:

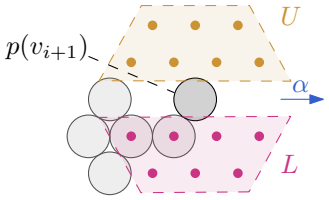

Figure 5.2: The affinity biases the heuristic towards the upper or lower area, whichever contains more (weighted) free space. In this example, the coordinates marked by dots are part of the colored upper and lower area, which are determined from the position of the parent vertex disk in emphasized black and the principal direction α.

$$g(i,\alpha) = \begin{cases} 1 & \text{if } i = 0 \text{ and } \alpha = \overrightarrow{e}(0°), \\ f(i,\Gamma^2(\gamma_s+\alpha)) + f(i,\Gamma(\gamma_s+\alpha)) & \text{if } i > 0 \text{ and } f(i,\gamma_s+\alpha) = 1, \\ 0 & \text{otherwise.} \end{cases}$$

The weight function g evaluates a direction α as "no space" if it is not possible to embed the next spine vertex in direction α because the coordinates are immediately blocked. Among the free two-step neighbors, it gives more weight to the immediately adjacent free coordinates by counting them twice (recall that $\Gamma(\gamma) \subset \Gamma^2(\gamma)$). These coordinates are available for both branch and leaf disks, rather than those suitable only for leaves.

At the first step only, when there is no maximum embedded spine vertex v_j, the principal direction is fixed to "straight right" without loss of generality. The principal direction is evaluated only when embedding a spine vertex. The same principal direction then applies for embedding all adjacent branches and their leaves. Therefore, the direction which maximises the free space at index i is the principal direction, i.e.

$$\alpha = \begin{cases} \arg\max_\alpha g(i,\alpha) & \text{if } v_i \in S \\ \arg\max_\alpha g(j-1,\alpha) & \text{otherwise} \end{cases}$$

Let $\gamma^- = (x^-, y^-) = d_i(p(v_{i+1}))$ be the assigned x- and y-coordinate of the parent of the next vertex in order $\leq_G$, and let $\alpha = \overrightarrow{e}(\beta)$ be the principal direction.

An x- and y-coordinate $\gamma = (x^- + x, y^- + y)$ are *upper coordinates* if $x \cdot \sin(-\beta) + y \cdot \cos(-\beta) > 0$. They are *lower coordinates* if $x \cdot \sin(-\beta) + y \cdot \cos(-\beta) < 0$. Refer to Figure 5.2 for a visualization. The set $U = \{\gamma \in \Gamma^2(\gamma^-) \mid \gamma \text{ are upper coordinates}\}$ is the *upper area* and $L = \{\gamma \in \Gamma^2(\gamma^-) \mid \gamma \text{ are lower coordinates}\}$ is the *lower area*. The *affinity* $a \in \{-1, 1\}$ at step i of the algorithm guides its decision for embedding of branches and leaves. It is defined as

$$a = \begin{cases} 1 & \text{if } i = 0 \text{ or } f(i,L) \leq f(i,U), \\ -1 & \text{otherwise.} \end{cases}$$

Affinity compares available space "above" and "below" a line imagined through the parent coordinates γ^- along the principal direction. The heuristic thereby gains a tendency to embed a branch on the more spacious side of the spine, overall balancing disks on both sides. Coordinates on the side indicated by a are preferred for assignment as described later in this chapter. Although our definition includes the special case of $i = 0$, in which there is no parent vertex, this default has no bearing in practice because decisions on spine vertices are not affected by affinity.

We are now ready to formulate the criteria which apply in the selection of the `heuristic` function output to determine our heuristic strategy.

1. *Bend heuristic*: When deciding the embedding coordinates for a spine vertex, we take a step in the principal direction from the previous spine vertex coordinates. This implements design goal 1.
2. *Packing heuristic*: For branches and leaves, we prefer coordinates which lie further in the opposite of the principal direction. This implements design goal 2.
3. *Balance heuristic*: For branches and leaves, we strictly prefer coordinates matching the affinity. This implements design goal 3.
4. *Space criterion*: Coordinates with too few free neighboring coordinates to fit all the leaves are not valid branch candidates. This implements design goal 4.

Among these decision criteria, higher precedence is given akin to lexicographical order. For branches and leaves in particular, the space criterion trumps balance, which trumps packing.

The pseudocode in Algorithm 5.1 describes how the elements of the heuristic function come together. The spine bends in the principal direction, which promises the most free space. We evaluate candidate coordinates in back-to-front order to encourage tight packing. We distribute branches evenly above and below the spine by affinity, which tends to preserve the x-monotone spine coordinates as unassigned. We embed branches only where there is enough space left for leaves.

Consider this example involving precedence of the different considerations. If the next vertex is a branch and there are "far-back" candidate coordinates at $\gamma_s + \overrightarrow{e}(-120°)$, when the affinity is $a = 1$, the upper area candidate at $\gamma_s + \overrightarrow{e}(60°)$ is preferred. If there are 4 leaves on the branch and not enough free space in the neighborhood of the preferred candidate, $\gamma_s + \overrightarrow{e}(0°)$ is chosen instead, and so on until some candidate satisfies the space criterion.

5.3 Breadth-First Order

We also examine a variant of the heuristic algorithm that uses a different vertex embed order, denoted $\leq_G^B$. The original definition for $\leq_G^D$ permits the case that, while embedding

a particular subtree (branch and leaves), the algorithm might embed a leaf at critical coordinates that would have been required as a free space for embedding some later branch vertex. Instead of embedding all the leaves on a branch directly after the branch vertex, the variant definition orders all branch vertices with the same parent spine to be embedded directly following that spine. Only then does it consider all the adjacent leaves.

We call this variant order the *breadth-first* order to distinguish it from the previously defined depth-first order. A breadth-first embedding order $\leq_G^B$ is *admissible* if it fulfills the following criteria, analogous to admissibility for a depth-first order.

- $\leq_G^B$ is a total order. It is reflexive, transitive and antisymmetric.
- $\leq_G^B$ orders the spine like an admissible depth-first order.
- $\leq_G^B$ groups branches and leaves together with their parent spine. Let $s_1, s_2 \in S, b \in B, l \in L$ and $s_1 \leq_G^B s_2$. If $s_1 = p(b)$, then $b \leq_G^B s_2$. If $s_1 = p(p(l))$, then $l \leq_G^B s_2$.
- $\leq_G^B$ groups branches before leaves in a subtree. Let $s \in S, b \in B, l \in L, s = p(b)$ and $s = p(p(l))$. Then $b \leq_G^B l$.
- Parent vertices come first. Let $v_1, v_2 \in V$. If $v_1 = p(v_2)$, then $v_1 \leq_G^B v_2$.

Otherwise, the definition of the algorithm and heuristic function remain the same.

Function `heuristic` (v_{i+1}, d_i)
- **Input:** next vertex v_{i+1}, partial embedding function d_i
- **Output:** embedding coordinates for v_{i+1}
- **Data:** lobster $G = (S \dot\cup B \dot\cup L, E)$, candidate coordinates κ
- $\gamma_s \leftarrow \max(v \in S \wedge v \leq_G v_{i+1})$;
- $\alpha \leftarrow$ `principal_direction`(d_i, γ_s); `// bend heuristic`
- **if** $v_{i+1} \in S$ **then**
  - **return** $\gamma_s + \alpha$;
- $\gamma_p \leftarrow p(v_{i+1})$;
- $a \leftarrow$ `affinity`(d_i, γ_p, α); `// balance heuristic`
- **for** β *in* $180°, 120°, 60°, 0°, -60°, -120°$ **do** `// packing heuristic`
  - $\kappa \leftarrow \gamma_p + R(a \cdot \beta) \cdot \alpha$;
  - **if** $f(i, \kappa)$ **then**
    - $\mathcal{L} \leftarrow \{l \in L \mid (v_i, l) \in E\}$;
    - **if** $f(i, \Gamma(\kappa)) \geq |\mathcal{L}|$ **then** `// space criterion`
      - **return** κ;
- **fail**; `// no representation found`

Algorithm 5.1: The heuristic decision function. We use the embedding order $\leq_G$, the parent function p, the neighborhood function Γ and the free space function f. The rotation matrix $R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$ helps us orient candidates relative to the principal direction α.

CHAPTER 6

Evaluation

The two algorithms from Chapter 4 and Chapter 5 both decide Problem 3 for size n lobsters in time linear in n. In this chapter, we extend our understanding of their performance and correctness in practical application.

We provide a software implementation of the approaches. Then we measure the run time and answers on the exhaustively enumerated set of all lobster instances with spine length ranging from 2 to 7.

6.1 Software Implementation

Both algorithms are included in the functionality of the `udcrgen` ("unit disk contact representation generator") program, which was entirely written as part of this thesis. `udcrgen` is written in C++ using the standard library and no other dependencies—though its unit tests do depend on Google Test [Goo]. It uses the CMake [IC] build system. The source code is available online[1].

The program reads in a graph instance, runs one of the available algorithms on it and produces statistical output. This output contains, for every instance tested, the algorithm used, the instance size, spine count, recognition success (yes/no) and runtime. If the instance is found to admit an embedding, `udcrgen` can also output such an embedding as a drawing.

It additionally offers "benchmark mode", in which it does not read an instance as input. Instead, it generates an enumeration of possible lobsters within user-specified restrictions on spine count given as input parameters. All generated instances become inputs for the algorithms. Though the enumeration omits some instances to save time, it covers

[1] `https://github.com/Animiral/udcrgen`

enough to decide any lobster within the given limits. We describe the exact method in Section 6.2 below.

During execution, `udcrgen` produces diagnostic log messages at different levels of detail. The level and target log file can be configured by the user.

These algorithms are implemented in the program:

- The "strict contact" algorithm described by Klemz et al. [KNP15], with a configurable gap size for minimum distance of disks not in contact. It applies only to caterpillar graphs.
- The "dynamic program" described in Chapter 4.
- The "heuristic algorithm" described in Chapter 5. It features an "embed order" parameter, which selects either the depth-first or breadth-first order of vertices.

Input graphs may be given (if not generated) either

- in a *degree notation*—see Section 6.2—, which specifies the vertex degree of each spine vertex in a caterpillar, or
- as an *edge list*, which may describe any graph. The program recognizes a caterpillar or lobster, if possible, and converts it to an internal representation as described in Section 6.3. If provided with an unrecognized type of graph, the program terminates with an error.

The decision result and runtime measurement are recorded in a CSV file. The drawing(s) of yes-instances may be written to an `.ipe` file, the format of the IPE drawing editor [Che] (only in a single-instance mode), or in SVG format to an HTML file (also in benchmark mode).

`udcrgen` can also create an *archive* consisting of two directories: one to hold all the yes-instances encountered, and another one for the no-instances. The archive uses a degree list format.

6.2 Instance Enumeration

To exhaustively examine lobsters with n spine vertices, we must define a systematic method of enumerating them. Our idea is based on the degree notation for lobsters. In this notation, a lobster with n spine vertices is represented as an *identifier* string

$$d_{1,1}d_{1,2}d_{1,3}d_{1,4}d_{1,5}\_d_{2,1}d_{2,2}d_{2,3}d_{2,4}d_{2,5}\_\ldots\_d_{n,1}d_{n,2}d_{n,3}d_{n,4}d_{n,5},$$

where $d_{i,j} \in \{\mathtt{x}, 0, 1, 2, 3, 4, 5\}$ is the number of leaf vertices adjacent to the jth branch on the ith spine vertex, or `x` to signify "no branch at this index". We consider the `x`

Spine Length	Yes-Instances	No-Instances	Max. Vertex Count (Yes)
2	141	101	24
3	1107	757	29
4	9343	6297	34
5	80952	54336	39
6	698352	468667	44
7	6041183	4053036	49

Table 6.1: Through our enumeration, we discovered the listed number of lobsters. This enumeration is exhaustive for the given sizes, meaning that it accounts for every yes-instance for Problem 3 up to efficient omissions by branch order, orientation and dominance discussed in this chapter. The no-instances are not exhaustive. They are simply informative about the arbitrary number of data points which emerged from our method of "adding vertices until it fails". This even lead us to evaluate lobsters with one more vertex than the maximum space afforded by the spine length, e.g. $|S| = 3$ with $|V| = 30$.

equivalent to -1 for ordering purposes. Any lobster instances with $n \geq 2$ that contain too many branches or leaves to be represented in this way certainly do not admit a disk contact embedding.

For all $1 \leq i \leq n, 1 \leq j \leq 5$ we initialize $d_{i,j} = \mathtt{x}$, starting with a lobster consisting only of a spine without any branches. We instantiate the internal graph representation from the degree notation and test it with regards to the decision problem. We obtain the next identifier by interpreting it like a $5n$-digit base-7 number, which we increment. If the last digit overflows, we carry the increment to the next digit and so on.

By this naive method, the number of tests to run for size n is 7^{5n}. The actual number of instances which we evaluated in practice is listed in Table 6.1. We reduce the necessary work in several ways.

6.2.1 Branch Ordering

Since the minimum distance from a spine vertex is by definition the only distinguishing feature for any vertex in a lobster, we can always exchange any two branches of the same spine by swapping $d_{i,j}$ and $d_{i,k}$ in the representation. It still identifies the same lobster. We constrain our representation with the following *branch ordering criterium*:

$$d_{i,j} \leq d_{i,k} \text{ if } j < k.$$

This reduces the valid representations to $\binom{7+5-1}{5}^n = 462^n$, where $\binom{7+5-1}{5}$ is the number of 5-multisets from 7 elements.

6.2.2 Orientation

Let $s_1, \ldots, s_n$ be the sequence of spine sections—5 digits each. The sequence $s_n, \ldots, s_1$ then identifies the same lobster. We remove these duplicates from consideration by defining some arbitrary order $<_s$ over all valid spine constellations and enforce that the spine sequence must be *canonically oriented*. A sequence has canonical orientation if

- it is the empty sequence or
- $s_1 <_s s_n$ or
- $s_1 = s_n$ and $s_2, \ldots, s_{n-1}$ is canonically oriented.

6.2.3 Dominance

Let

$$A = a_{1,1}a_{1,2}a_{1,3}a_{1,4}a_{1,5}\_\ldots\_a_{n,1}a_{n,2}a_{n,3}a_{n,4}a_{n,5}$$

and

$$B = b_{1,1}b_{1,2}b_{1,3}b_{1,4}b_{1,5}\_\ldots\_b_{n,1}b_{n,2}b_{n,3}b_{n,4}b_{n,5}$$

be two lobster identifiers. Then A *dominates* B if for all $1 \leq i \leq n, 1 \leq j \leq 5, a_{i,j} \leq b_{i,j}$. Since A is easier than B, if we determine that A does not admit an embedding, we infer that B does not admit an embedding either and forgo testing the lobster described by B.

Similarly, if we would first determine that B does admit an embedding, we may conclude that A does as well. In practice, this does not help us because we examine our lobsters in strict incremental order.

6.2.4 Heuristic-Based Skip

The test on any given lobster may, depending on the user's choice, involve running both the heuristic algorithm and the dynamic program. If this is the case, and if we are only interested in learning the answer to the decision problem (not in comparing the performance of the algorithms), then the fast heuristic solution should be attempted first. If it succeeds in finding that the input is a disk contact lobster, the slower exact algorithm can be skipped.

Our experiment, described below in Section 6.4, does in fact compare algorithm performance. Thus we do not apply this particular optimization.

6.3 Graph Representation

The internal graph representation is the data structure which our algorithms operate on when they answer the disk contact problem. It must fulfill the following design criteria.

- Sparsity—the structure should not store more than $O(n)$ data points for n vertices.

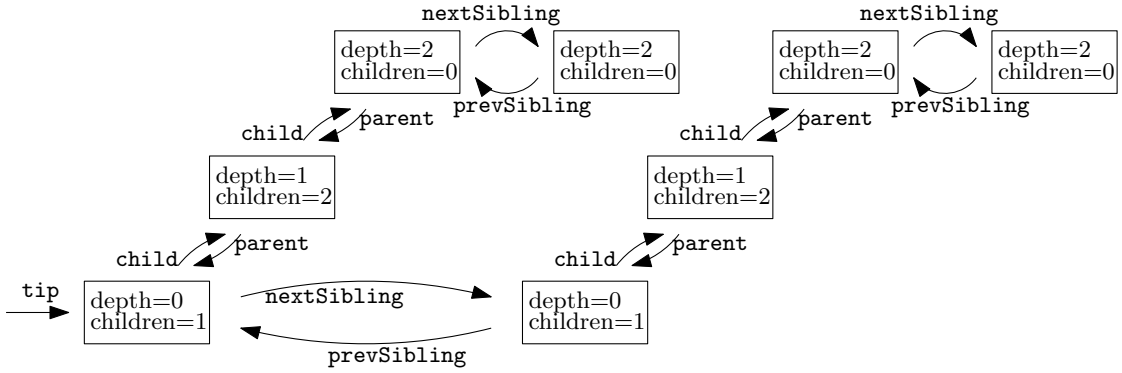

Figure 6.1: The `DiskGraph` data structure with `Disks`.

- Fast traversal must be supported under admissible depth-first and breadth-first orders as defined in Chapter 4 and Chapter 5.
- The query of relevant vertex properties must be fast:
  - the depth of the vertex—0 for spines, 1 for branches and 2 for leaves and
  - the number of children, i.e. vertices which have the queried vertex as their parent.
- The internal representation must be easily convertible from the (user-supplied and generated) input graph formats as well as to the supported embedding output formats.

In the implementation, this structure is named `DiskGraph`. It holds a collection of `Disks` and a pointer to the `tip`, which is the `Disk` that represents the first spine vertex. The `Disk` structure holds the vertex properties, including the constructed result of the (partial) embedding of that vertex. Figure 6.1 illustrates the structure. `Disks` are represented by boxes and pointers by arrows.

6.3.1 Format Conversion

To convert from and to the degree list representations to the internal representation and also to produce the drawing from the coordinate data is trivial. However, the implementation can also accept an input graph as an edge list, which can describe any graph. In that case, we remove all the leaves to see if there remains just a string of vertices. If so, the graph is a caterpillar. Otherwise, we again remove all the leaves from the remainder to see if there remains just a string of vertices. If so, the graph is a lobster. We can reconstruct the removed graph components in reverse order.

These steps run in time $O(n)$ for n input edges.

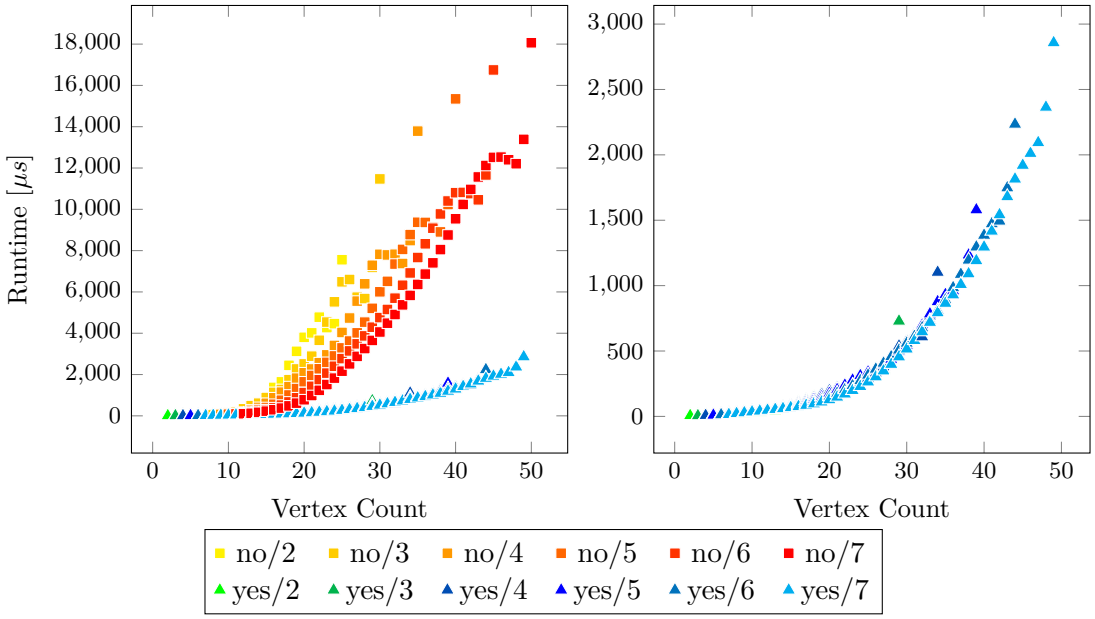

Figure 6.2: The runtime of the dynamic program is depicted here. Every plotted point indicates the mean across all data points from the experiment with the same vertex count, spine length and instance class. Marks are colored and shaped according to the spine length and instance class respectively. The second plot holds the same data, illustrating just the yes-instances. Note the increased runtime for instances around the maximum vertex count as listed in Table 6.1. These instances mark the decision boundary from both sides, where the available space is saturated.

6.4 Experiment

We ran the `udcrgen` software in benchmark mode with the dynamic programming algorithm, the heuristic algorithm with depth-first order and the heuristic algorithm with breadth-first order. The hardware specifications of the university-scale system used were: CPU: 2× Intel Xeon E5540, 2.53GHz Quad Core, RAM: 24GB. Note that the implementation is not parallelized.

The benchmark covers lobsters with at least 2 and at most 7 spine vertices enumerated using the scheme described in Section 6.2. In total, $11,414,272$ lobsters were evaluated.

6.4.1 Dynamic Program Runtime

In Figure 6.2 and Table 6.2, we see the resulting average execution time of the dynamic program on lobsters of different sizes.

Among same-sized instances, those with a longer spine run faster. This is because there are fewer candidate coordinates for a spine vertex than for a branch vertex, and fewer candidates for a branch than a leaf.

$\|V\|$	#yes	#no	$\varnothing\mu s$ DP/yes	$\varnothing\mu s$ DP/no	$\varnothing\mu s$ H/yes	$\varnothing\mu s$ H/no
3	2	0	5.50		7.50	
9	130	3	28.94	74.25	19.35	16.50
15	3272	385	75.37	543.69	34.49	34.22
21	31581	7028	188.50	2,257.75	29.20	28.95
27	151315	47442	378.25	4,337.38	40.04	39.35
33	351093	157841	750.49	6,783.19	43.00	42.60
39	402216	290471	1,355.00	9,797.65	52.00	50.85
45	203481	251466	1,919.84	14,631.98	59.37	56.57

Table 6.2: This excerpt of the experiment data shows the mean runtime of the dynamic program (under "DP") next to the heuristic with DFS order (under "H"). The data is aggregated from all instances of the specified vertex count $|V|$ and class as specified in the header ("yes"/"no"). The number of data points aggregated in the "DP" columns is the same as the number of yes- and no-instances ("#yes", "#no").

We can observe that starting from about 25 vertices, the runtime of the dynamic programming algorithm is linear in the size of the input. The reason for the sharper rise in smaller instances, especially among yes-instances, is that the proportion of leaf vertices, which cause the greatest number of sub-problems to consider, grows initially and later remains constant in longer lobsters.

No-instances in our benchmark generally take far longer to compute than yes-instances. One reason is the early exit strategy discussed in Section 4.3.1, which allows yes-instances to be answered without evaluating many of the generated sub-problems. Another reason is the way we enumerate instances, described above in Section 6.2. All our no-instances are at the brink of being "almost solvable". With these instances, we rarely encounter an early clog of too many vertices which would refute the instance. The algorithm is really forced to exhaust every possibility.

6.4.2 Heuristic Runtime

Figure 6.3 and Table 6.2 show the empirical runtime of the heuristic algorithm tested on the same set of instances as the dynamic program. The same data in direct comparison with the dynamic program data is shown in Figure 6.4. Overall, the heuristic offers up to about a 30× speedup compared to the dynamic program on similar-sized yes-instances and about 250× speedup compared to the dynamic program on our no-instances—which are, as mentioned before, generally unfavorable to the dynamic program.

While the dynamic program is faster on instances with more spines, the heuristic algorithm seems to require a small, fairly constant amount more time to place a spine in our implementation. The trend is linear in the vertex count of the instance, as expected. There is no performance difference between yes- and no-instances.

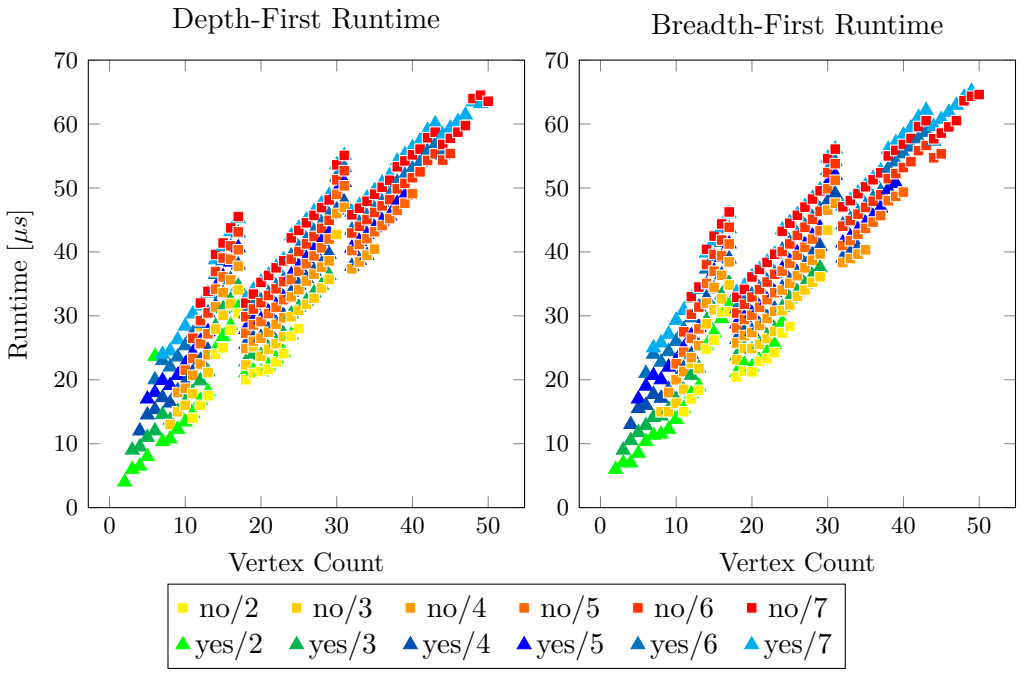

Figure 6.3: The runtime of the heuristic algorithm is depicted here both for depth-first and breadth-first order. Every plotted point indicates the mean across all data points from the experiment with the same vertex count, spine length and instance class. Marks are colored and shaped according to the spine length and instance class.

The data in Figure 6.3 also shows a consistent sudden increase in runtime around specific vertex counts, especially 15 and 31. Since these sizes are not related to any feature of the problem instances, we hypothesize that they can probably be attributed to quirks of the test system, like the `std::unordered_map` associative data structure from the C++ standard library implementation. This container holds the vertex coordinates in our heuristic algorithm implementation. The system used for the full experiment used GNU `libstdc++`. We test our hypothesis with an alternative experiment on lobsters of spine length 2–4, run on a Windows system using Microsoft's C++ standard library implementation. The results, depicted in Figure 6.5, show that the alternative experiment does not produce the same pattern of "jumps". The hypothesis is thus confirmed.

6.4.3 Heuristic Accuracy

Our heuristic approach with depth-first order answers between 70%–90% of all yes-instances of spine length 2–7 correctly. Table 6.3 accompanied by Figure 6.6 shows the percentages for the depth-first vertex order and compares it to the breadth-first variant. The variant is shown to perform consistently worse by around 20 percentage points.

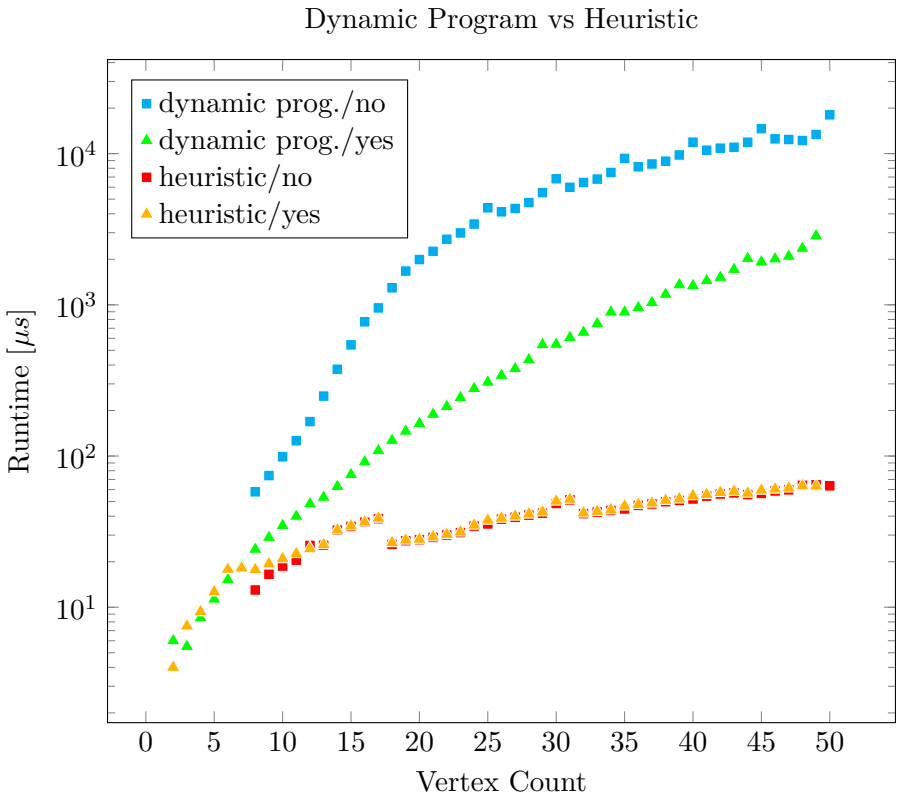

Figure 6.4: This is a direct comparison of the mean runtime of the dynamic program and the heuristic with DFS order. The scale is logarithmic, as at linear scale, the heuristic runtime simply appears flat at the bottom. Marks are colored and shaped by algorithm and instance class.

Spine Length	Yes-Instances	No-Instances	Yes (DFS)	Yes (BFS)
2	141	101	126 (89%)	93 (66%)
3	1107	757	1023 (92%)	778 (70%)
4	9343	6297	8092 (87%)	6077 (65%)
5	80952	54336	65635 (81%)	48652 (60%)
6	698352	468667	529979 (76%)	387235 (55%)
7	6041183	4053036	4285811 (71%)	3090527 (51%)

Table 6.3: Listing of the number of lobsters evaluated in the experiment, summed by spine length. The columns “Yes (DFS)” and “Yes (BFS)” show the absolute and relative number of lobsters which we know to be yes-instances (verified by the dynamic program) recognized as such by the heuristic using the indicated vertex order.

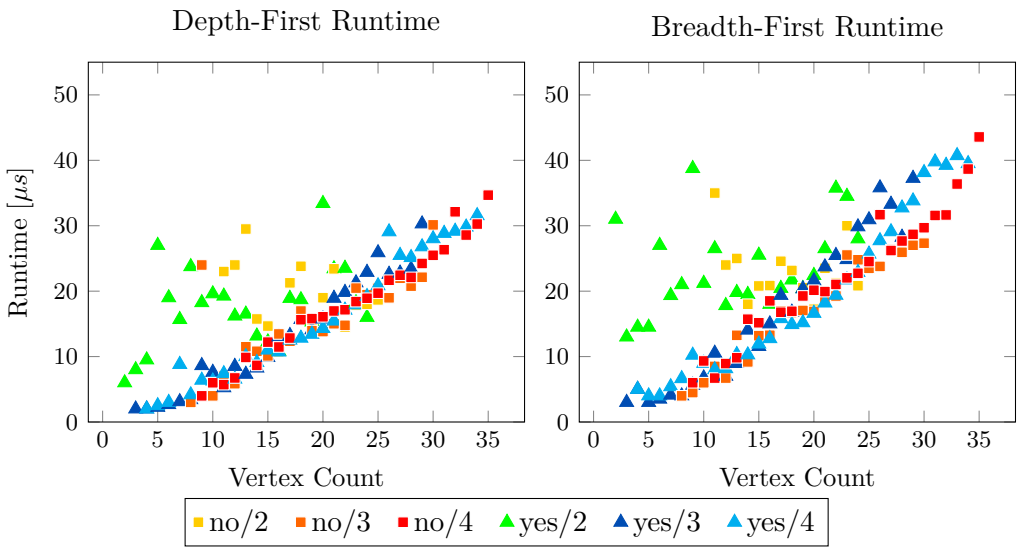

Figure 6.5: The runtime of the heuristic algorithm as in Figure 6.3, sampled on a different system for comparison.

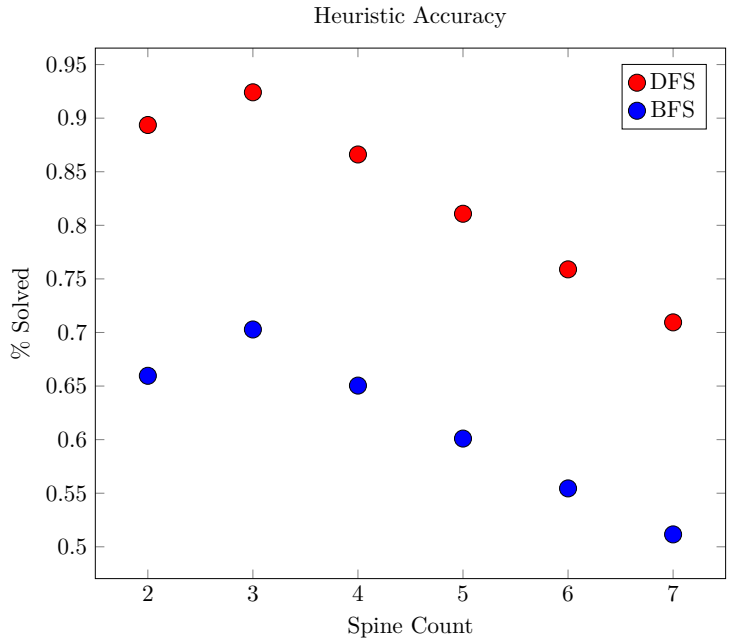

Figure 6.6: The circles indicate the percentage of yes-instances correctly recognized by the two variations of the heuristic algorithm.

CHAPTER 7

Conclusion

In this thesis, we reviewed the unit disk contact graph recognition problem and its complexity on different graph classes. We rigorously defined two main approaches to decide the tri-grid x-monotone weak UDC recognition problem on lobster graphs: the dynamic program as theorized by Bhore et al. [Bho+21], and a fast and less accurate heuristic approach as an original contribution.

The treatise on the dynamic program includes discussion on several strategies which improve performance. The heuristic is presented as two variants, which we compare in empirical evaluation. This evaluation shows that the depth-first variant is superior to the breadth-first variant.

We present `udcrgen`, a program that offers a software implementation of several different algorithms for UDC graph recognition: the recognition algorithm for caterpillars described by Klemz et al. [KNP15], the dynamic program to recognize UDC lobsters as described in Chapter 4 and based on a proof by Bhore et al. [Bho+21] and the original heuristic to recognize UDC lobsters as described in Chapter 5. `udcrgen` supports a "benchmark mode" to exhaustively explore the space of small lobster instances with these algorithms and various input and output formats.

Our data from running the benchmark implementation demonstrates the real-world runtime that we can expect. It shows that the heuristic may yield a triple-digit speedup with accuracy declining from 90% with increasing instance size in the answers.

7.1 Open Questions

Our results are specific to x-monotone representations on the triangular grid. Based on our lack of counter-examples, it might be reasonable to assume that these specific lobsters are the same that admit a weak UDC reprensention in general. Still, this assumption is so far unproven.

Imagine a counter-example which is not x-monotone. There is no intuitive way to define a UDC lobster in a way that will force the spine to "bend back on itself". The specimen would have to force this kind of constellation by tricky specification of branches and leaves, but will soon step on its own toes for lack of space around the spine. The embedded disks on both ends may only get in the way of each other.

Imagine another counter-example which is not restricted to the triangular grid. As discussed in Chapter 2.3, whether every lobster which admits a weak UDC also admits a weak UDC on the triangular grid is an open question. Considering the small subtree depth of the lobster, the question is whether the difference between the tri-grid and some tighter packing can be relevant or even ruin the linear-time result.

The x-monotonicity concept can be described as a relaxation of an even tighter restriction, the *straight spine*, in which no bends in the spine are allowed. Clearly, UDC lobsters do not all have a straight spine—a well-placed 4-leafed branch enforces a bend on a lobster that still admits an embedding perfectly fine. However, this leads us to consider whether x-monotonicity might be too relaxed. Perhaps it is sufficient to restrict UDC lobsters' spines to just two of the six cardinal directions of the triangular grid. We offer this conjecture as the final contribution of this thesis.

Conjecture 1 (60°-Monotone UDC Recognition for Lobsters)**.** *Any lobster G which admits a weak UDC representation also admits a weak UDC representation on the triangular grid wherein, for any two consecutive spine vertices v and u, the angle between $\overrightarrow{vu}$ and the x-axis is either* $0°$ *or* $60°$*.*

We hope to have hereby contributed a small step towards further research on the much more significant open question:

> What is a property $p(G)$, of any graph G, that holds if, and only if, the UDC problem for G can be decided in linear time?

List of Figures

List of Tables

List of Algorithms

Bibliography

[Ala+13] Md. Jawaherul Alam, Therese Biedl, Stefan Felsner, Michael Kaufmann, Stephen G. Kobourov, and Torsten Ueckerdt. „Computing Cartograms with Optimal Complexity“. In: *Discrete & Computational Geometry* 50.3 (2013), pp. 784–810. DOI: 10.1007/s00454-013-9521-1.

[Bho+21] Sujoy Bhore, Maarten Löffler, Soeren Nickel, and Martin Nöllenburg. „Unit Disk Representations of Embedded Trees, Outerplanar and Multi-legged Graphs“. In: *Graph Drawing and Network Visualization*. Vol. 12868. Springer International Publishing, 2021, pp. 304–317. DOI: 10.1007/978-3-030-92931-2_22.

[BK98] Heinz Breu and David G. Kirkpatrick. „Unit disk graph recognition is NP-hard“. In: *Computational Geometry* 9.1-2 (1998), pp. 3–24. DOI: 10.1016/S0925-7721(97)00014-X.

[Bow+15] Clinton Bowen, Stephane Durocher, Maarten Löffler, Anika Rounds, André Schulz, and Csaba D. Tóth. „Realization of Simply Connected Polygonal Linkages and Recognition of Unit Disk Contact Trees“. In: *Graph Drawing and Network Visualization*. Vol. 9411. Springer International Publishing, 2015, pp. 447–459. DOI: 10.1007/978-3-319-27261-0_37.

[CCN19] Man-Kwun Chiu, Jonas Cleve, and Martin Nöllenburg. „Recognizing embedded caterpillars with weak unit disk contact representations is NP-hard“. In: *Proceedings of the 35th European Workshop on Computational Geometry*. 2019. URL: http://www.eurocg2019.uu.nl/papers/47.pdf.

[Che] Otfried Cheong. *The Ipe extensible drawing editor*. URL: https://ipe.otfried.org/ (visited on 12/31/2022).

[Cle20] Jonas Cleve. „Weak Unit Disk Contact Representations for Graphs without Embedding“. In: *CoRR* (2020). arXiv: 2010.01886.

[Goo] Google. *Google Testing and Mocking Framework*. URL: https://github.com/google/googletest (visited on 10/09/2022).

[Hal80] William K. Hale. „Frequency assignment: Theory and applications“. In: *Proceedings of the IEEE* 68.12 (1980), pp. 1497–1514. DOI: 10.1109/PROC.1980.11899.

[IC] Kitware Inc. and Contributors. *CMake Build System*. URL: https://cmake.org/ (visited on 10/09/2022).

[KK85] Krzysztof Kozminski and Edwin Kinnen. „Rectangular duals of planar graphs“. In: *Networks* 15.2 (1985), pp. 145–157. DOI: 10.1002/net.3230150202.

[KNP15] Boris Klemz, Martin Nöllenburg, and Roman Prutkin. „Recognizing Weighted Disk Contact Graphs“. In: *Graph Drawing and Network Visualization*. Springer International Publishing, 2015, pp. 433–446. DOI: 10.1007/978-3-319-27261-0_36.

[KNP22] Boris Klemz, Martin Nöllenburg, and Roman Prutkin. „Recognizing Weighted and Seeded Disk Graphs“. In: *Journal of Computational Geometry* 13.1 (Sept. 2022), pp. 327–376. DOI: 10.20382/JOCG.V13I1A13.

[Koe36] Paul Koebe. „Kontaktprobleme der konformen Abbildung“. In: *Ber. Sächs. Akad. Wiss. Leipzig, Math.-Phys. Klasse* 88 (1936), pp. 141–164.

[MM13] Colin McDiarmid and Tobias Müller. „Integer realizations of disk and segment graphs“. In: *Journal of Combinatorial Theory*. B 103.1 (2013), pp. 114–143. DOI: 10.1016/j.jctb.2012.09.004.