The Longest Cycle Problem on Solid Grid Graphs

A Thesis
Presented to
The Division of Mathematical and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Maxim Kopylov

September 2025

Approved for the Division
(Computer Science)

James D. Fix

Acknowledgements

I would like to thank the College and the CS department for feeding my passion, and teaching me the skills that make any problem feel solvable.
Special thanks goes to my advisor, who gave me much needed direction, and helped me form the occasional coherent idea.

List of Abbreviations

Acronyms missing a letter refer to the original in some way; for example **HC** may be used to mean Hamiltonian Cycle, **LC** to mean Longest Cycle, etc.

TSP	Travelling Salesman Problem
HCP	Hamiltonian Cycle Problem
LCP	Longest Cycle Problem
GG	Grid Graph
SGG	Solid Grid Graph
LP	Linear Program
ILP	Integer Linear Program
MILP	Mixed Integer Linear Program
NP	Nondeterministic Polynomial time
P	Polynomial time

Table of Contents

List of Figures

Abstract

This thesis explores some computational problems in the context of grid graphs, namely the Hamiltonian cycle problem and travelling salesman problem. These are famous for being the canonical examples of NP-completeness, and any algorithm solving them efficiently would prove that P=NP. In a landmark paper, Umans and Lenhart devise an algorithm that shows that the Hamiltonian cycle problem on solid grid graphs is in P.

This original contribution of this thesis examines the related longest cycle problem on these same graphs, whose complexity is currently unresolved. We explore integer linear programming solutions to this problem, which are tested on a large number of randomly-generated solid grid graphs using CPLEX and Python. Our work supports a recent conjecture made about this problem, and poses a direction towards a polynomial time solution.

Chapter 1

Introduction

According to superstitition, a picture taken from a camera captures not only the visual likeness of a person, but also their soul. If one can exchange pleasantries through an exchange of images in the same way one would over a coffee or brunch, then an image holds something essential to both interactions — or so the thinking goes.

So does ours, when we think about grid graphs. Like either a digital picture or a person's soul, a grid graph can both represent a thing and contain some irreducible essence of it. More specifically, encoding other types of graphs as grid graphs preserves some complexity-theoretic properties, namely the difficulty of the travelling salesman (TSP) (and other related) problems for those types of graphs.

The topic of this thesis is the difficulty of such problems on a more restricted type of graph, the solid grid graph. For general graphs, there is no known polynomial-time algorithm for the Hamiltonian cycle problem (HCP). However, there is a polynomial-time algorithm for the Hamiltonian cycle problem on solid grid graphs (SGG-HCP). Since HCP and TSP are in the same difficulty class for general graphs, we've some optimism that they are also in the same difficulty class for solid grid graphs, and that the travelling salesman problem on solid grid graphs (SGG-TSP) is also solvable in polynomial time. Towards the end of this thesis, we provide describe an integer linear programming solution to the longest cycle problem on solid grid graphs (SGG-LCP), which is an important subproblem to TSP-SGG.

In Chapter I (this chapter), we give a broad overview of the topics discussed in this thesis.

In Chapter II, we precisely define the terms used throughout the rest of this thesis.

In Chapter III, we go over the establishing literature around the difficulty of the problems on solid grid graphs mentioned above.

In Chapter IV, we develop a theoretical framework for proving some statements about SGG-LCP.

In Chapter V, we experimentally verify the ideas presented in Chapter IV by coding them into a linear program and running them on a large set of examples. We conclude with an opinion on the state of the travelling salesman problem in solid grid graphs, and pose a direction for future research.

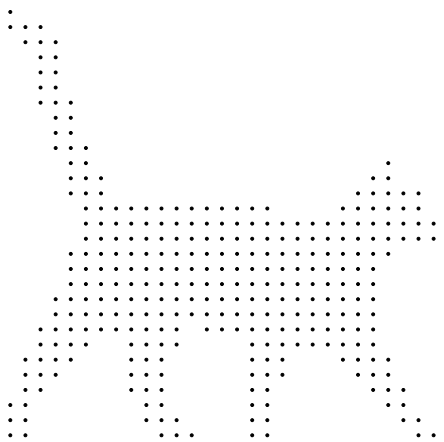


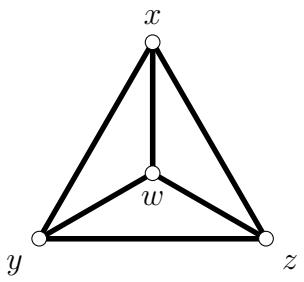
Figure 1.1: A cat in grid graph form.

Chapter 2

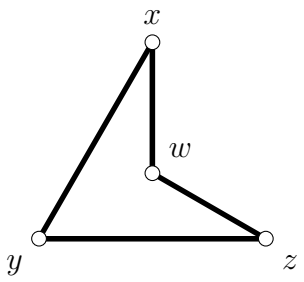
Preliminaries

2.1 Graphs

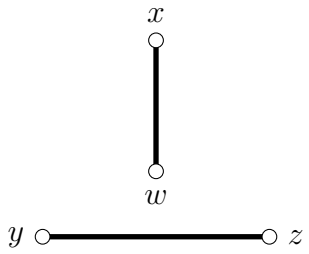
Graphs are a way of representing some relationships between things. They are depicted as networks of *nodes* or *vertices*, which represent objects, connected by *edges*, which represent the relationships among the objects. Graph theory is a rich and well-studied field of mathematics, which we introduce very briefly in this section.



(a) A graph with 6 edges



(b) A graph with 4 edges



(c) A graph with 2 edges

Figure 2.1

A **graph** G is a set of **nodes** or **vertices** $V(G)$, together with a set of **edges** or pairs of nodes $E(G)$. We consider two types of graphs: undirected graphs, and

directed graphs. In **undirected graphs**, the edge (a, b) is the same as the edge (b, a); for **directed graphs**, the edge (a, b) is different from (b, a). Figure 2.1 gives some examples of undirected graphs, while figure 2.2 gives an example of a directed graph.

A **subgraph** $F \subset G$ is a graph F where $V(F) \subset V(G)$ and $E(F) \subset E(G)$; figures 2.1b and 2.1c are subgraphs of 2.1a.

In this thesis, we will mostly consider undirected graphs. In such graphs the **neighbor** of a node u is a vertex v such that (u, v) is an edge in the graph G. An edge e is **incident** to a node u if it contains u. The **degree** of a node is the number of edges incident to it, or the number of neighbors it has. In figure 2.1a, each node has a degree of three.

We call a sequence of neighboring vertices $x_0, x_1, x_2 \dots x_\ell$ a **walk** of length ℓ, where each **step** (x_i, x_{i+1}) of a walk is a directed edge of the graph. A walk without repeated steps is called a **trail**, and a walk without repeated vertices is called **simple** or a **path**. The vertex sequence a, c, b, d, f, d, e in figure 2.2 is a trail, while the sequence a, c, b, d, e is a path.

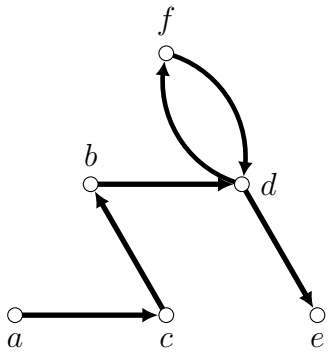


Figure 2.2: A trail on a graph with 6 vertices.

We can write that a path exists between x and y as $x \rightsquigarrow y$. We say graphs A and B are **connected** whenever, for each pair of vertices x and y, $x \rightsquigarrow y$. A maximally connected subgraph of a graph is called a **component**.

A **simple cycle** or **cycle** is a walk that ends where it starts and only repeats the starting vertex as the last vertex. A **Hamiltonian cycle** is a cycle that includes all the vertices of a graph. A **tour** is a trail that ends where it starts, and includes every vertex in the graph. If it has a minimum number of edges, we call it an optimal tour or a **travelling salesman tour**. Figures 2.3 and 2.4 illustrate the differences between a Hamiltonian cycle and a travelling salesman tour.

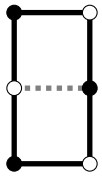


Figure 2.3: A Hamiltonian cycle

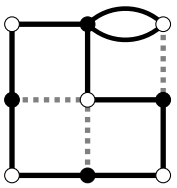


Figure 2.4: A travelling salesman tour

An **edge cover** is a set of edges for which every vertex is incident to at least one edge in that set. We say F is *covers* G or that F is *covering* if F is an edge cover and $V(F) = V(G)$. A k**-factor** is an edge cover where each vertex is incident with exactly k edges in that set. Each node in figures 2.1a, 2.1b, and 2.1c has degree three, degree two, and degree one; hence they are a 3-factor, a 2-factor, and a 1-factor respectively.

Although we are largely concerned with undirected graphs, paths and cycles are written as directed series of edges. We often use *doubly-directed* graphs to model undirected graphs, where both an edge and its twin are included in the graph.

If an edge $e = (a, b)$, then its reverse or **twin** edge is $\text{twin}(e) = (b, a)$. The **s-node** (for *starting*) of e is given by $s(e) = a$, and its **t-node** (for *terminating*) is given by $t(e) = b$.

It's often useful to assign a real-valued cost or weight to edges in graphs, especially for many of the graph optimization problems related to the ones we study here. The graphs we study here are unweigted.

A **grid graph** is an graph whose vertices lie on the integer plane, where only coordinates distance 1 apart from each other can be connected by an edge. A **face** of a grid graph is a tour whose interior contains no vertices of the grid graph. **Solid grid graphs** are grid graphs without any 'holes', or equivalently, where every bounded face has an area of 1. The **boundary** of a grid graph the tour that follows its outermost edges; we will sometimes refer to the area outside of the boundary as the *unbounded* face. Figure 2.5 shows a general grid graph, while figures 2.6, 2.3, and 2.4 show solid grid graphs.

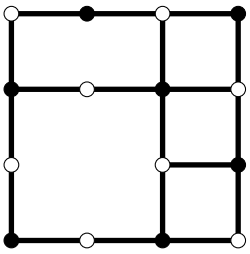


Figure 2.5: A grid graph that is not solid.

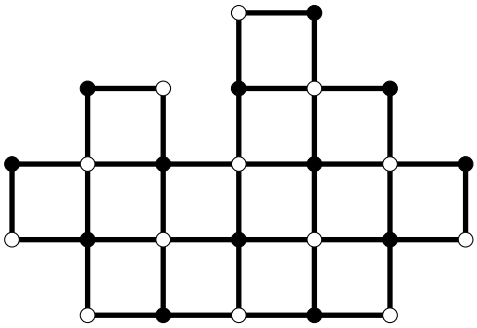


Figure 2.6: A solid grid graph.

Certain graphs are **bipartite**, meaning their vertices can be partitioned into *even* and *odd* sets, so that any neighbor of an even node is odd and any neighbor of an odd

node is even. We indicate the bipartite-ness of a graph with a checkerboard pattern that alternates between light and dark on its vertices.

2.2 Complexity Theory

The main subject of this thesis are two graph problems whose computational complexity have yet to be resolved for solid grid graphs, namely the travelling salesman problem (TSP) and the longest cycle problem (LCP). These are provably "probably difficult" for general graphs. For readers familiar with the theory, what we mean is that both problems are NP-complete for general graphs, but may not be for solid grid graphs. This means that it could be fruitful to seek a polynomial time algorithm for solving them on solid grid graphs.

We will talk more about these problems and their status in the next chapter. Here instead we give a brief overview of some of the terms we just used; for a more formal overview, please refer to chapter 34 of Cormen et al. (2009) and chapter 7 of Sipser (2006).

Computational complexity theory is the study of the hardness of certain computational problems, characterizing them by the amount of resources (usually running time) needed to solve them; problems that require more time are harder than problems that require less. In the most foundational blocks of this theory, a problem is formally defined as a set of instances along with a "yes" versus "no" decision that should be made for each instance. For example, in the Hamiltonian cycle problem (HCP) for undirected graphs the instances are undirected graphs and the decision should be whether or not a graph has a Hamiltonian cycle. An algorithm that solves HCP is one that given a description of a graph as an input outputs a 1 when there is a Hamiltonian cycle, and a 0 when there is not. In this theory, problems are categorized into complexity classes; one example is the time complexity class **P**. A problem is in P if it is solved by an algorithm that runs in time polynomial in the size of its inputs. We strongly suspect HCP is not in P, but this is currently an open question.

For many problems, including HCP, TSP, and LCP, a yes-no answer isn't all we want from an algorithm. For example, for HCP we'd like to know a Hamiltonian cycle for a graph when it has one. More formally, we'd say that such a cycle is a *witness* to a graph having the property of being Hamiltonian. Sparing the details, the class **NP** is the set of all decision problems with polynomial-sized witnesses that can be checked by an algorithm in polynomial time. What's being highlighted in this definition is that fact that, even though it's been difficult to find an efficient algorithm that decides whether a graph has a Hamiltonian cycle, it's easy to quickly check whether a given sequence of vertices forms the steps of a Hamiltonian cycle for a graph.

The famous **P$\stackrel{?}{=}$NP problem** is often dubbed one of the most important problems in contemporary computer science. It's easy to see that P is a subset of NP, but whether or not P is the same as NP is an open question, one that asks informally whether 'solving a problem is as easy as checking a solution to it'. In exploring the theory surrounding this problem, complexity researchers have developed the class of NP-complete problems, which are in a formal sense the hardest problems in NP with respect to P.

Again, sparing the details, this research programme looks at the relative hardness of decision problems. In doing so, we say that a problem A is at least as hard as the problem B when the instances of A can be translated in polynomial time to instances of B while preserving the decision about the instance. Such a translation is called a reduction of A to B. Compellingly, this would allow us to say that if B is in P, then A is also in P, for example. Any problem K in NP that's NP-complete, it turns out, is one where each problem in NP can be reduced to K via some polynomial time reduction.

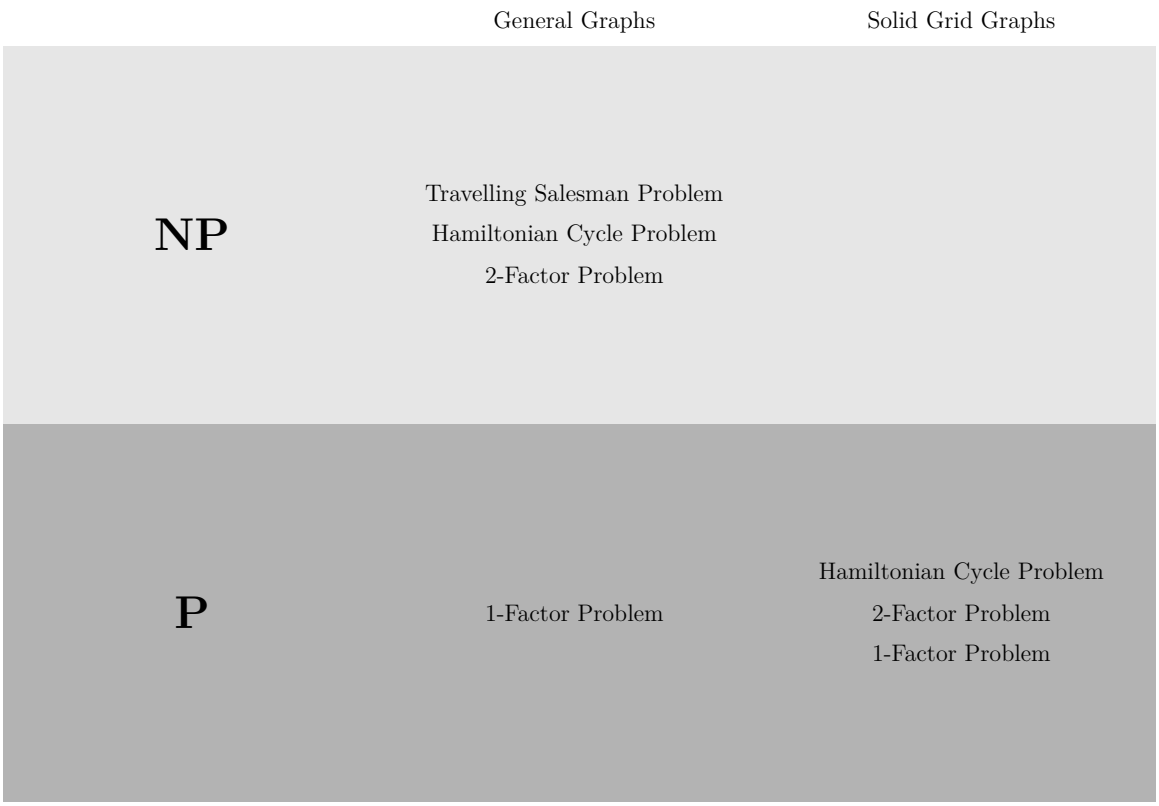


Figure 2.7: The complexity hierarchy of various problems related to the travelling salesman problem in solid grid graphs. All the problems in P are also in NP, but whether the converse is true is an open question.

2.3 Linear Programming

A good attempt to find an efficient solution to a problem should use any tool available to do so. One such historically fruitful and very general tool is linear algebra, as well as its close relative, geometry. **Linear programming** uses linear algebra to find solutions to optimization problems, most interestingly in combinatorial and not-especially-geometric contexts. This section gives a brief introduction to this topic, derived primarily from Grötschel et al. (1991).

We use the standard linear algebra notation, where c^T is the transpose of vector c, if x is a vector then $c^T x$ is the dot-product of c and x, and so forth. We might also use bold font to indicate matrices, and imply elementwise relations between scalars and matrices or vectors (e.g. '$0 \leq b$' means '0 is less than or equal to every element of b').

A **linear constraint** is a linear inequality or equality, e.g. one in the form $c_1 x_1 + c_2 x_2 \cdots c_i x_i \leq b$. Multiple linear constraints of the same type can be written concisely in matrix form; for example $\mathbf{A}x \leq b$. The ith row of $\mathbf{A}$ is written as $\mathbf{A}_i$, so its ith constraint is $\mathbf{A}_i x \leq b_i$. If we wanted to be more verbose, we could write the ith row and kth column of $\mathbf{A}$ as $\mathbf{A}_{i,k}$, and the linear constraint as $\sum_k \mathbf{A}_{i,k} \cdot x_k \leq b_i$.

A set of linear constraints in the form $\mathbf{A}x \leq b$ describe a **polyhedron** called the *feasible region* in the n-space of possible solutions $x \in \mathbb{R}^n$. Each constraint row describes a half space $\mathbb{R}^n$, and the solutions x where these m constraints are met all sit within the intersection of these half spaces. Each face on this polyhedron corresponds to some row i constraint $\mathbf{A}_i x \leq b$.

LP algorithms find the points on polyhedra that maximize a linear **objective function** (in our case, these **optima** are all vertices of the polyhedron). We write these kinds of problems as **linear programs** (LPs), which specify the objective function to be optimized, the polyhedron, and any other constraints. A linear program can be written in the following way:

$$\max c^T x$$
$$\mathbf{A}x \leq b$$

that is, it maximizes the objective $c^T x$ subject to the linear constraints $\mathbf{A}x \leq b$.

Linear programs can be solved in polynomial time. However, when dealing with unweighted graphs and other discrete structures, we are interested in **integer linear programs** (ILPs), or linear programs that have integer-valued solutions. Many NP-complete problems can be reduced to an ILP, so there is no guarantee that an ILP has a polynomial runtime (unless P=NP).

When looking specifically for binary-valued solutions, we often encounter matrices that are *totally unimodular* (TU), where every 2×2 submatrix (i.e. a 2×2 matrix

produced from any reordering and deletion of rows and columns) has determinant -1, 0, or 1. These ILPs can be solved in polynomial time.

However, this condition is often too strong, since polytime ILPs that aren't TU do exist. One way around this through an **LP relaxation**, which relaxes variables in an ILP to be non-integer. Relaxed ILPs are called **mixed-integer linear programs** (MILPs). Some MILPs can be shown to be **totally primal integral** (TPI), where optimal solutions are guaranteed to be integer-valued even if they are not constrained to be so.

For cases where there are no theoretical guarantees, MILP solvers will automatically use a variety of methods to solve a linear program as efficiently as possible in practice. We can treat them as black boxes, shedding light on them only as necessary.

2.3.1 LP example: maximum matching

A note about notation

In Grötschel et al. (1991), linear programs are written as $\max x \ \mathbf{A}x \leq b$, like we do above. Later, when formulating actual ILPs, they use indicator variables X that correspond to entries of x, i.e. each entry of x corresponding to an edge e is X_e.

While this distinction might not be really necessary, we follow it here because of two reasons: to distinguish binary values in an LP, and because our LPs will get notationally very messy and X_{v^+} is easier to read than x_{v^+}. So for all intents and purposes, a big X is the same as a little x in an LP, and more specifically $0 \leq X \leq 1$.

Moroever, we'll also use auxillary variables such as d_v or t_e, and we'll use big X variables to indicate the 'outputs that we care about' from an LP that are distinct from these auxillary variables.

The following example gives an ILP for finding a maximum matching of a graph G. A *maximum matching* is the largest subgraph of G that is also a 1-factor.

The solution vector x is indexed by each edge e in the graph, where $x_e = 1$ if the edge e is in the matching and 0 otherwise. We write these as indicator variables X_e.

$$X_e = \begin{cases} 1 & \text{if } e \text{ is in the matching} \\ 0 & \text{otherwise.} \end{cases}$$

Each node in the solution has a degree of at most 1, so for each node v we have the constraint

$$\sum_{e \,|\, v \in e} X_e \leq 1 \tag{2.1}$$

Our objective is to maximize the number of edges included in the solution. Then the ILP can be written like this

$$\begin{aligned} &\max 1^T x \\ &x \in \{0,1\} \\ \text{under } &(2.1) \\ \text{s.t. } &\mathbf{A}x \leq 1 \end{aligned}$$

where

$$\mathbf{A}_{v,e} = \begin{cases} 1 & \text{if } v \in e \\ 0 & \text{otherwise.} \end{cases}$$

Each constraint of the polyhedron has the form $\mathbf{A}_v x \leq 1$, where the columns of $\mathbf{A}_v$ correspond to the edges in G incident to v. Since all entries in $\mathbf{A}$ are 1 or 0, the matrix is totally unimodular. This means that finding integer solutions to this linear program is in P.

2.3.2 LP example: maximum 2-factor

The following example gives an ILP for finding a maximum 2-factor on an undirected graph G. A *maximum 2-factor* is the largest subgraph of G that is also a 2-factor. In our linear program, we model the undirected graph G with doubly-directed edges.

However, instead of a degree of 1, each node in the solution has a degree of at most 2. And even more specifically, the maximum number of 'incoming' edges and the maximum number of 'outgoing' edges is at most 1 (we call these the *edge-flow* constraints). We define

$$\text{in}(v) = \{e \,|\, t(e) = v\}$$

$$\text{out}(v) = \{e \,|\, s(e) = v\}$$

and so our constraints are that

$$\sum_{e \in \text{out}(v)} X_e \leq 1 \tag{2.2}$$

$$\sum_{e \in \text{in}(v)} X_e \leq 1 \tag{2.3}$$

To prevent satisfying these constraints with a doubled edge, we ensure that only one among an edge and its twin can be included in the solution.

$$X_e + X_{\text{twin}(e)} \leq 1 \tag{2.4}$$

(here we leave out the quantifier $\forall e \in E(G)$ since it can be inferred)

To make sure that a node has even degree, we can constrain its net flow to be 0, so that the number of incident incoming edges is equal to the number of incident outgoing edges.

$$\sum_{e \in \text{in}(v)} X_e = \sum_{e \in \text{out}(v)} X_e$$

$$\Longrightarrow$$

$$\sum_{e \in \text{in}(v)} X_e - \sum_{e \in \text{out}(v)} X_e = 0 \tag{2.5}$$

The constraints (2.2), (2.3), and (2.5) together ensure that a node has either degree 0 or 2.

As in the 1-factor ILP, our objective is to maximize the number of edges included in the solution. Then the ILP can be written as follows:

$$\begin{aligned} &\max 1^T x \\ \text{s.t. } &\mathbf{A}x \leq b \\ \text{under } &(2.2),\ (2.3),\ (2.4),\ (2.5) \\ \text{and } &x \text{ is binary.} \end{aligned}$$

Though our constraints aren't phrased directly with respect to the $\mathbf{A}x \leq b$ form, with some work the reader should be able to figure them out. Let $n = |V|$ and m be the number of undirected edges of G; what we have is a matrix $\mathbf{A}$ with $2m$ columns, n rows from constraint (2.2), another n rows from constraint (2.3), an additional set of m rows from constraint (2.4), and a set of n rows for the non-positive constraint (2.5). The vector b the $\mathbf{A}x \leq b$ form of our LP has an entry for each row with either a 0 or 1, depending on the constraint. In the next example we do away with worrying about naming the variables of x, and we let the reader work out the values of c, A, and b.

This linear program is not totally unimodular due to the non-positive constraint (2.5), and even more strongly, finding a 2-factor is generally NP-complete and so is solving this linear program as an ILP.

2.3.3 LP example: undirected cycle of length ℓ

In later sections, being able to find cycles with ℓ edges on undirected graphs will be very useful. Here we present an ILP that does this.

This ILP is inherits its constraints directly from the 2-factor LP, as a cycle is a 2-factor on the nodes it covers. However, the 2-factor ILP doesn't guarantee that the solution is only one component. To get this guarantee, we introduce the idea of a timestep into our linear program.

The general idea is to consider a cycle as a walk that takes a certain amount of time. If our clock stepping into a vertex reads k, our clock stepping out of a vertex should read $k + 1$. Moreover, at the start of our walk we've accumulated 1 unit of time, and at the end we should step back into where we started having accumulated ℓ units of time.

We'll label the time at an edge e as t_e, where t_e is 0 if e isn't part of the cycle, and however many steps away it is from the start of the cycle otherwise.

We should make sure that any vertex can be the start, because there might be some specific structure around a vertex that prevents a cycle including it from being the longest. We can do this by introducing another set of variables d_v, where d_v is 1 if v is distinguished as the start and 0 otherwise.

On nodes included in the cycle, the time at an edge going out is 1 added to the time at an edge coming in. However, if a node isn't included, the time coming in is the same as the time going out (which is 0). Implied here is another set of variables X_v, which indicates the inclusion of node v in the solution. Note that this is essentially for notational convenience, since $X_v = 1$ if $\sum_{\text{in}(v)} X_e = 1$ and 0 otherwise; i.e. $X_v = \sum_{\text{in}(v)} X_e$. Hence an initial constraint is:

$$\sum_{e \in \text{out}(v)} t_e = \sum_{e \in \text{in}(v)} t_e + X_v$$

When v is distinguished as being the starting node, the time at the edge going out is $1 - \ell$ added to the time at the edge coming in (since we begin at the start with 1 unit and end at the start with ℓ units). We can modify the previous constraint to include this case using d_v as a switch to subtract ℓ. Since d_v is 0 when v is not the starting vertex, this applies to every vertex v of the graph.

$$\sum_{e \in \text{out}(v)} t_e = \sum_{e \in \text{in}(v)} t_e + X_v - \ell \cdot d_v \tag{2.6}$$

To further restrict the possible time values on an edge, we constrain that an edge's time value is at least 1 if the edge is 1, and 0 if the edge is 0 or if its twin is 1.

$$t_e \geq X_e \tag{2.7}$$

$$t_e \leq \ell \cdot X_e \tag{2.8}$$

$$t_e \leq \ell \cdot (1 - X_{\text{twin}(e)}) \tag{2.9}$$

(as before, we leave out the quantifier $\forall e \in E(G)$ when it can be inferred)

The following constraints specify that the starting node is part of the cycle, and that there's only one starting node.

$$d_v \leq X_v \tag{2.10}$$

$$\text{(for each } v \text{ in } V(G))$$

$$\sum_{v \in V(G)} d_v = 1 \tag{2.11}$$

By a simple counting argument we can see that the time values of each edge on the graph should sum to $\frac{\ell(\ell+1)}{2}$. Since the solution is a cycle, the number of edges should equal the number of nodes covered, both of which are ℓ.

$$\sum t_e = \frac{\ell(\ell+1)}{2} \tag{2.12}$$

$$\sum X_v = \ell \tag{2.13}$$

$$\sum X_e = \ell \tag{2.14}$$

All of our variables should be integer valued, some of them being binary.

$$t_e \in \mathbb{Z}_+ \tag{2.15}$$

$$d_v \in \{0, 1\} \tag{2.16}$$

$$X_v \in \{0, 1\} \tag{2.17}$$

$$X_e \in \{0, 1\} \tag{2.18}$$

Any x satisfying these constraints will work, so the linear program is:

$$\begin{gathered} \text{find } x \\ \text{s.t. } \mathbf{A}x \leq \mathbf{b} \\ \text{under (2.2) - (2.18)} \end{gathered} \tag{2.19}$$

Needless to say, this integer linear program is not totally unimodular. Even checking its feasibility reduces to the general Hamiltonian cycle problem (by setting $\ell = |V|$), which is in NP-hard. It can be nevertheless be used as part of an efficient program with additional constraints, which we will do in later chapters.

Chapter 3

Previous Work

In this chapter we address the status of various grid graph problems, and give a rough technical overview of the methods used to get here. The original contribution of this thesis, covered in later chapters, attempts to develop polytime solutions for the longest cycle problem and the related travelling salesman problem on solid grid graphs.

Hamiltonian cycle problem on grid graphs (GG-HCP).
Decide whether a grid graph has a Hamiltonian cycle. This problem is **NP-complete**.

Hamiltonian cycle problem on solid grid graphs (SGG-HCP).
Decide whether a solid grid graph has a Hamiltonian cycle. This problem is in **P**.

longest cycle problem on solid grid graphs (SGG-LCP).
Decide whether the number of edges in a longest cycle of a solid grid graph is greater than k. This complexity of this problem is currently **open**.

Travelling salesman problem on solid grid graphs or *The travelling salesman tour problem on solid grid graphs (SGG-TSP).*
Decide whether the number of edges in a travelling salesman tour of a solid grid graph is less than k. The complexity of this problem is currently **open**.

3.1 Hamiltonian cycle problem

3.1.1 Papadimitriou et. al (1982)

In "Hamilton paths in grid graphs", Itai et al. (1982) prove the Hamiltonian cycle problem to be NP-complete in grid graphs. They do this by reduction, showing that cubic (i.e. with nodes having degree three) bipartite graphs can be embedded in grid graphs so that a Hamiltonian path in one implies a Hamiltonian path in the other. Because the problem is known to be NP-complete for the former, this shows that it's NP-complete for the latter as well.

The reduction has two stages: embedding a bipartite cubic graph into a thin rectangular grid graph, then scaling the result in a way that allows for Hamiltonian cycles.

The first stage preserves the structure of an embedded bipartite cubic graph G in an embedding G' (i.e. if x and y are neighbors in G, then there is a path $x \rightsquigarrow y$ in G' that is vertex-disjoint with G, apart from x and y). Fig. 3.1 illustrates this step, where the cubic graph in 3.1a is transformed into the embedding shown in 3.1b.

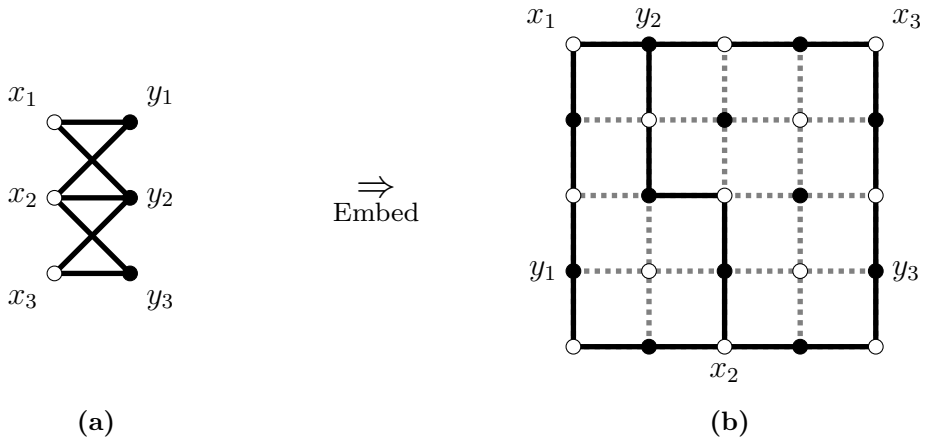


Figure 3.1: An initial 'thin' embedding of a small bipartite cubic graph. Subsequent transformations essentially scale these up, and further preserve the structure of the original graph.

The next stage scales the result of the previous so individual vertices become *clusters* of nine grid-graph vertices, and edges are reconstituted as 2-thick rectangles or *tentacles*. A small example of the final result is given in fig. 3.2.

The 2-thickness of a tentacle allows for the option of a returning path along an edge not permissible in a thin embedding; this allows for adding paths that cover the extra nodes added to a scaled up Hamiltonian path in G. The rest of the paper proves that with minor (i.e. polynomial-time computable) considerations, clusters and tentacles can be arranged so that a Hamiltonian path exists in the final grid graph if and only if one exists in the original cubic bipartite graph.

The important lesson for us here lies in the broad implications of a successful reduction. If we think of a reduction as a transformation between media, like color

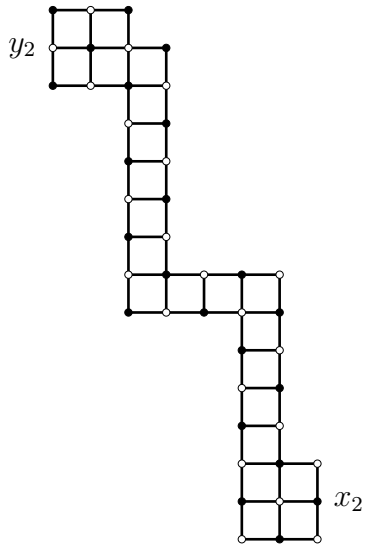


Figure 3.2: One possible embedding of the subgraph $x_2 \rightsquigarrow y_2$ in figure 3.1

to grayscale imagery, it being successful means answering a question about a thing in one medium is made no easier by converting it to another. An essential aspect of a thing is captured in either representation; in this case it's whatever is preserved in the 'structure-preserving' transformations we described above.

Somehow we should think that it's not possible to transform any graph into a solid grid graph while preserving its structure. As a medium of expressing things, solid grid graphs are limited in the sorts of structures they can express. This might lead us to believe that they are fundamentally much less complicated than other types of graphs, and that other NP-complete problems might be easier for them than other types of graphs.

3.1.2 Umans and Lenhart (1997)

The Hamiltonian cycle problem is decidable in polynomial time for solid grid graphs. The proof is given in the extended abstract "Hamiltonian Cycles in Solid Grid Graphs" by Umans and Lenhart, where they devise a scheme that iteratively improves a 2-factor until it becomes Hamiltonian or until it can't be improved.

We earlier defined a 2-factor as a covering set of edges where each node has a degree of 2. Here, however, we think of a 2-factor as a collection of blobbish things (cycles) that want to be merged.

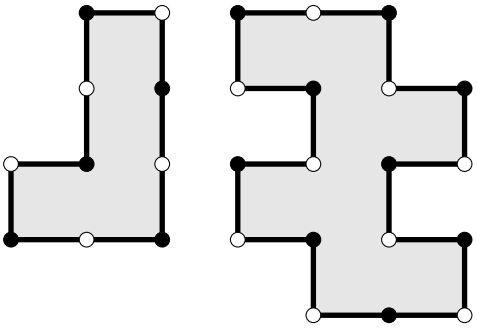


Figure 3.3: A 2-factor with each cycle's interior shaded.

Umans and Lenhart's construction finds spots between cycles suitable for merging called **type III cells** (*cell* here means the same thing as *face*). Essentially, by flipping the edges included in a type 3 cell, one can create a bridge between two blobs and turn them into one.

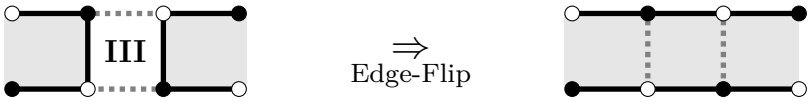


Figure 3.4

Thinking about blobs is useful here because of how edge-flips affect the *interiority* and *exteriority* of a cell. Considering the cells to the left and right of fig. 3.4, we find that before the flip, we have an in-out-in pattern, and after the flip, we have an in-in-in pattern.

Finding a Hamiltonian cycle here essentially boils down to repeatedly finding and flipping type III cells. This is achieved XOR'ing the edges on the perimiters of *alternating strips*, which are so named because their perimeters alternate between being in a 2-factor and not being in a 2-factor. Flipping edges in this way can decrease the number of components in the 2-factor.

(a) 1-cell and general odd alternating strips **(b)** 2-cell and general even alternating strips

Figure 3.5

More specifically, flipping an odd alternating strip (fig. 3.5a) decreases the number of components in a 2-factor. Type III cells are odd alternating strips of length 1, and

even alternating strips can be flipped as part of an *alternating strip sequence* to get an odd alternating strip — if the graph is Hamiltonian. Umans and Lenhart prove that if this can't be done, the graph is not Hamiltonian, and give a method for finding alternating strip sequences in polynomial time.

It seems surprising that we can glean facts about an entire graph just by looking at the places where it follows a certain pattern which is a priori fixed across all possible variations of that graph. And though it might seem like this says more about how the pattern exploits a graph being basically a bunch of boxes, we need to use a fundamentally different approach to get the same information for general grid graphs, even when they are also basically a bunch of boxes. So the local 'shape' of these graphs isn't the only reason for this, and it's not obvious how restricting them to be solid makes such a difference. We leave this as an interesting philosophical puzzle.

3.2 Travelling salesman problem

3.2.1 Fekete et. al (2017)

In "On the Travelling Salesman Problem in Solid Grid Graphs", Fekete et. al tackle the more general travelling salesman problem on solid grid graphs. In so doing, they essentially refute the idea that a 2-factor can be used to find an optimal travelling salesman tour in any straightforward way. Instead, they show that SGG-TSP can be reduced to SGG-LCP when a solid grid graph has a 2-factor, and give good reasons for the general case as well.

In their concluding paragraph, the authors state that "... it may be useful to know whether there always exists a longest cycle such that all vertices that are not part of this cycle lie on the boundary of the graph. If true, it may be possible to solve the question of how many vertices of the boundary must be deleted, such that the remaining graph is Hamiltonian". One can see that in this case, the longest cycle is the Hamiltonian cycle of the largest solid grid subgraph that has one, since removing boundary nodes preserves solidity. In the next chapter, we give an argument for why this is usually the case and when it might not be.

Chapter 4

Dual Graphs

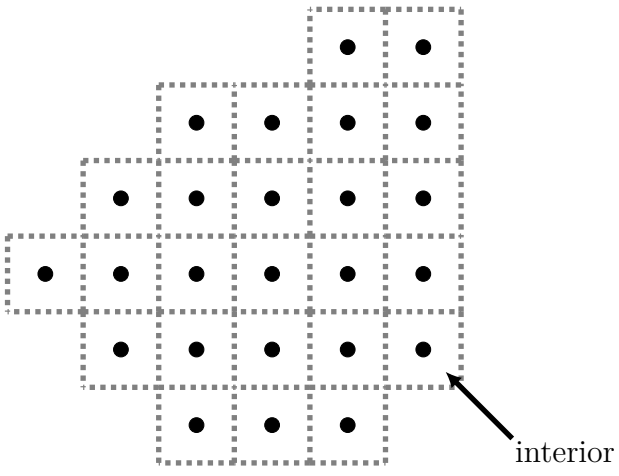
In the following sections we develop the claim that apart from easily detectable edge cases, the longest cycle of a solid grid graph covers a Hamiltonian solid grid graph.

Up until this point, we've avoided any mention of 'thin' solid grid graphs, i.e. ones where an edge bisects an exterior face. As it turns out (and Fekete et. al make this explicit), we need only consider 'bulky' solid grid graphs without such an edge for the purposes of solving SGG-LCP; hence we only consider bulky solid grid graphs in this and any following section.

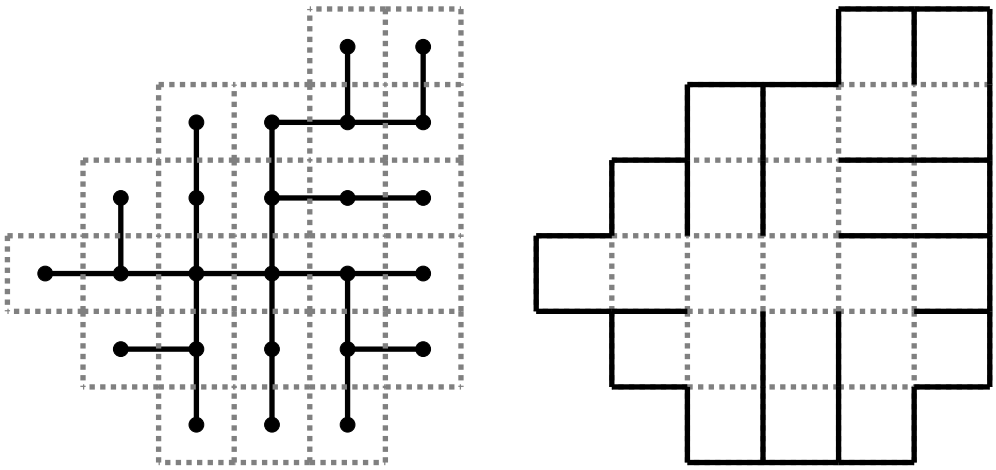
4.1 Characterizing simple cycles with respect to the dual

We define the **dual** of a planar graph to be a graph where vertices correspond to faces of the original **primal** graph. Written as G^+, the *interior* of a solid grid graph dual assigns a vertex to each bounded 1×1 square, while everything else is called the **exterior of the dual**.

There is a 1-1 correspondence between a solid grid graph and its dual. Under our definition, any edge missing on a face of a graph is crossed by an edge in the dual and vice-versa.



Since the dual is also a graph, so we can also think about trees, cycles, and other useful structures on it. The **boundary** of any of these is the edges of the faces in the primal graph corresponding to nodes in the dual, minus the edges crossed by an edge in the dual graph structure. More compellingly, if $G^\dagger$ is the dual of G, then G is the boundary of $G^\dagger$. (Sometimes we will refer to the boundary of a graph, which is the boundary of the interior of its dual).



(a) A dual tree. **(b)** The boundary of the dual tree.

Figure 4.1: Note that each edge in (a) corresponds to a missing edge in (b).

The dual of a simple cycle in a solid grid graph has a specific structure. For starters, dual nodes are edge-connected if and only if they are adjacent to each other. In addition, the dual of a simple cycle is connected, solid, and its diagonally-neighboring nodes share at least one non-diagonal neighbor. We can exploit this structure to find a longest cycle.

The next sections go more in depth on each of these **simple cycle dual properties**. In order to do so, we take the following statements as intuitively true:

I. Adjacent faces of a simple cycle share a missing edge. Moreover, if two faces of a simple cycle share a missing edge, they are adjacent.

II. Every face of a simple cycle is adjacent to another face of the simple cycle.

4.1.1 Adjacency and connectedness

One can through experiment experience the fact that a simple cycle has a dual that is connected, and that nodes in its dual are connected if and only if they are adjacent to each other. The proof of this is as follows.

Thm III: Nodes in the dual of a simple cycle are edge-connected if and only if they are adjacent.

Pf. Since dual nodes correspond to faces, *I* implies that nodes adjacent to each other in the dual correspond to faces that share a missing edge. Recalling our definition of the dual, removing an edge in the primal adds an edge to the dual, where the

added edge crosses the removed edge. So if two dual nodes correspond to adjacent faces of a simple cycle, they are connected by the dual edge crossing the missing edge shared between the two faces. Hence adjacent nodes of a simple cycle dual are connected. Going the other way, connected nodes in the dual of a simple cycle correspond to faces that share a missing edge, which must be adjacent.

Thm IV: The dual of a single component simple cycle also has a single component.

Pf. From *II* and *III*, a path between faces of a simple cycle implies a path between the dual nodes corresponding to those faces. Thus if the faces of a simple cycle belong to a single component, so do the nodes in its dual.

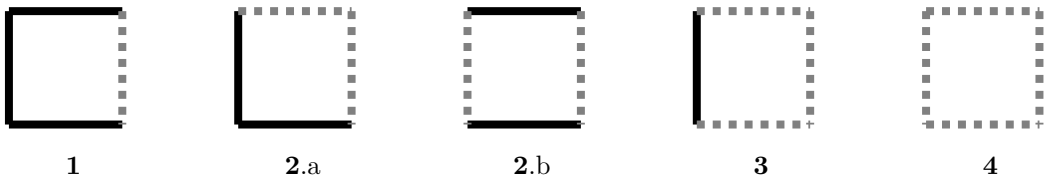
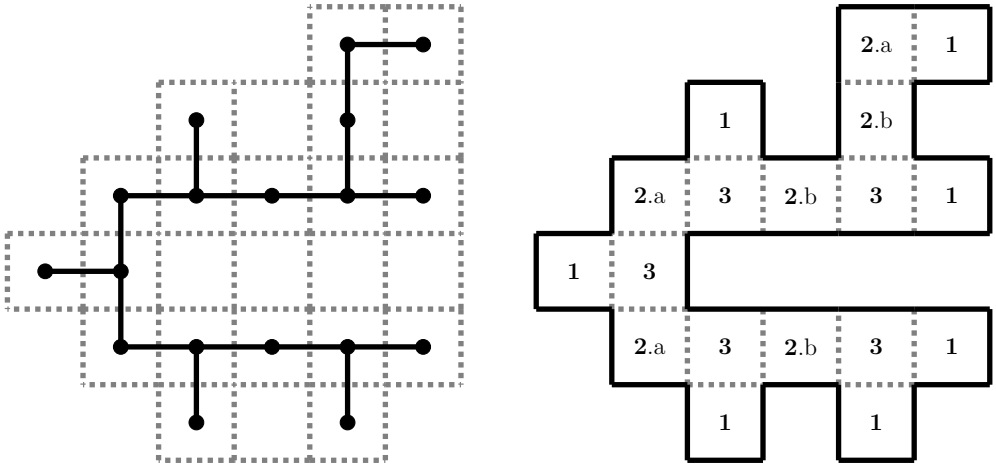


Figure 4.2: The five types of faces that can constitute a simple cycle longer than four edges, unique up to rotation. Each is labelled by the number of missing edges it shares with neighboring faces, all of which are on the interior of the cycle.



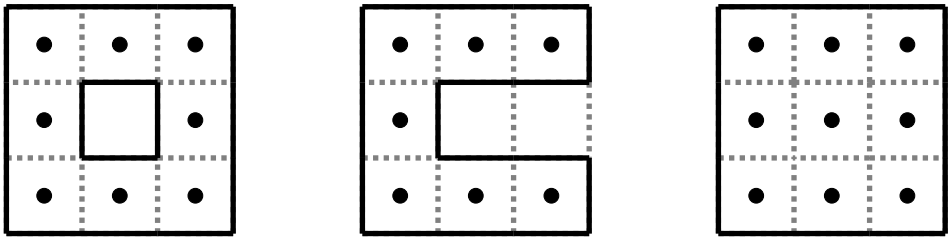
(a) The dual graph or 'footprint' of a simple cycle.

(b) The boundary of (a) with each face labelled.

Figure 4.3: Each dual edge in (a) corresponds to a pair of neighboring faces in (b) that share a missing primal edge.

4.1.2 Solidness

The dual of a simple cycle is also a solid grid graph, in the sense that it can't have any holes. Holes in the dual of a graph produce boundaries with an outer and inner component, which can be combined or deleted by removing the dual cycle or filling it in.

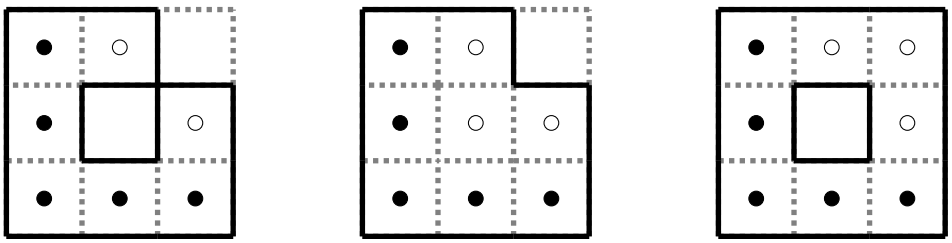


(a) **(b)** **(c)**

Figure 4.4: A dual graph that has a hole results a boundary with two components (a). Breaking the cycle makes it have one component (b), while filling it removes the inner component (c).

4.1.3 Diagonalness

Dual nodes diagonal to each other without shared neighbors cause nodes in the primal graph to have degree four, and so can't occur as part of any simple cycle. Hence, the dual of a simple cycle requires that any pair of diagonal nodes share at least one neighbor.



(a) **(b)** **(c)**

Figure 4.5: The unfilled diagonal nodes force a primal node to have degree 4 if they do not share a neighbor (a). Their shared neighbors are also unfilled in (b) and (c).

4.2 2 × 2 squares

Looking locally at the duals of longest simple cycles, we see that each 2 × 2 **square** centered on a primal node has a limited number of configurations with respect to the dual. Most notably, it seems unlikely that a 2 × 2 square would contain more than 3 dual nodes. This is no coincidence; a node centering a **4-full square** has degree 0, and this should be rare in a longest cycle. In fact, when the graph is Hamiltonian, there can't be any 4-full squares at all.

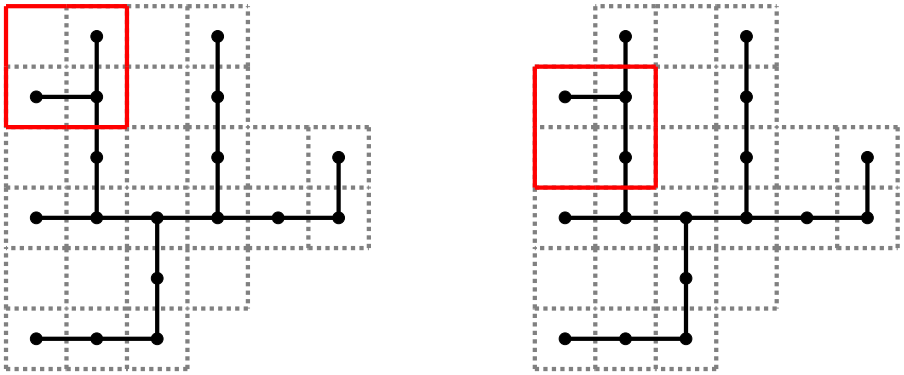


Figure 4.6: Distinct overlapping 2 × 2 squares, outlined in red, covering different parts of the same longest cycle.

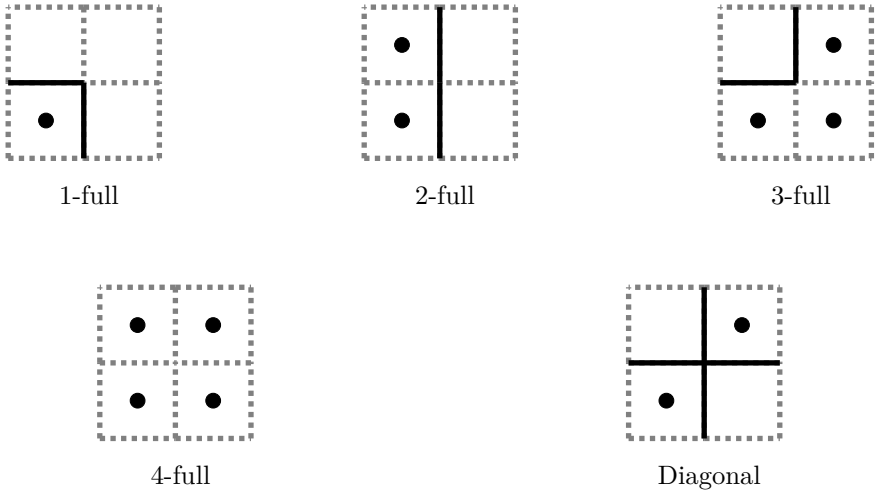


Figure 4.7: 1-full, 2-full and 3-full configurations allow the center node to have a degree of 2, and their occurrence should be maximized in a longest cycle. 4-full configurations should be rare, since their center node must have degree 0. Diagonal configurations should always be excluded.

We can extend this argument even further. When a solid grid graph is non-Hamiltonian, one of its longest cycles must have a 4-full square in one of two cases. In the first case, one of corners of the 4-full square can be 'tucked in', resulting in a

longest cycle of equal or greater length. In the second case, this can't be done without shortening the cycle, and so a longest cycle is forced to include the 4-full square. We argue that the latter case is somewhat artificial and quite rare. Hence, even when a solid grid graph is non-Hamiltonian, its longest cycle very likely covers a Hamiltonian solid grid subgraph.

This sheds some light on the question posed in section 3.2.1. Since removing non-boundary nodes from a solid grid graph can never make Hamiltonian solid grid graph, the above argument supports the claim that finding a longest cycle allows one to find the boundary nodes required to be removed to result in a Hamiltonian solid grid graph.

Because they allow us to build dual graphs whose boundaries cover Hamiltonian solid grid graphs, we call the **simple cycle configurations** depicted by the first three squares shown in figure 4.7 *Hamiltonian simple cycle configurations*.

4.2.1 Forced 4-full squares in a longest cycle

Most 4-full squares can have a corner tucked or otherwise removed to either increase the length of a boundary or keep it the same.

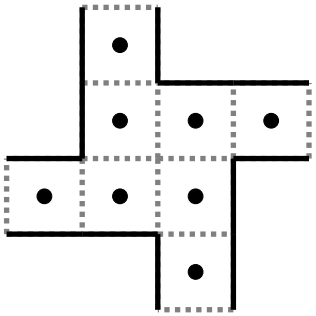
(a) **(b)**

Figure 4.8: A dual graph before and after tucking its corner.

(a) **(b)**

Figure 4.9: An example where removing a corner lengthens a boundary.

However, there is one case where neither of these can be done, because tucking a corner guarantees the resulting boundary to be shorter.



Since such cases guarantee any resulting boundary to be shorter, and because some combination of the rules depicted in figures 4.9a and 4.9 could be applied if otherwise, these cases must exist on the boundary of the initial graph prior to any attempt at constructing a longest cycle. The most straightforward way to detect them is to tuck each corner of each square of the initial dual graph and see if any resulting boundary is shorter, which can be done in polynomial time. We do this as part of our experimental section, ensuring that the random solid grid graphs we generate don't have any of these cases.

4.3 The size of a Hamiltonian cycle with respect to its dual

If n is the number of edges of a Hamiltonian cycle and k is the number of nodes in its dual, then $n = 2k + 2$. Moreover, if a dual is cyclic, $n < 2k + 2$. The first equality is true of some non-Hamiltonian solid grid subgraphs with n deficits from cylic components and surpluses from others, but satisfy the equation as a whole. Nevertheless, this is an incredibly useful fact as we will see in the next chapter.

Chapter 5

An ILP for the Longest Cycle on Solid Grid Graphs

In previous chapters, we've discussed the defining features of solid grid graphs, and some results about them. In this chapter, we'll use that information to make an integer linear program that finds a cycle of length ℓ, which can be used in an algorithm to solve SGG-LCP. We relate this algorithm to the question posed in 3.2.1, and provide evidence that supports the claim that the longest cycle of a solid grid graph covers all but a minimum number of boundary nodes (except in forced 4-full square circumstances).

To do this, we build an oracle that provides a cycle of length ℓ, or signals that it can't otherwise. Initially setting $\ell = |V|$, we decrease ℓ after each step until a cycle of length ℓ is given. Since no cycle with length greater than ℓ exists, this is the longest cycle.

We will use two oracles: the baseline one, and the modified one. Our baseline oracle is the general longest cycle ILP given in 2.19, which doesn't exploit any solid grid graph structure, but allows us to verify that we have the correct ℓ. Our modified oracle works similarly to the baseline, but heavilty exploits the structure of solid grid graphs.

Moreover, the modified oracle relaxes all of the variables used in the baseline, leaving only a small set of variables indicating the presence of a dual node as discrete. This is completely sufficient to get integer-valued solutions to all the other variables, even if they are continuous. We'll explain why this is in the coming sections.

5.1 Compass functions

To build our ILP, we should exploit the structure of solid grid graphs. To do this, we need a way to orient the positions and directions of primal and dual edges relative to each other.

We can call the *diagonal* nodes around a *center* node u by the letters a, b, c, d whenever they are arranged like figure 5.1. More precisely, we can say $a(u) = u + (-0.5, -0.5)$, $b(u) = u + (0.5, -0.5)$, and so on. When centered on a dual node, edges

on these nodes form the primal edges of a face.

The nodes lying in each cardinal direction of u are pictured as v_1, v_2, v_3 and v_4, and we call them the *neighbors* $N(u)$ of u.

Crucially, a pair of neighbors form a directed edge, which cross either a dual or a primal edge. The *primal edge function* gives the primal edge crossed by a dual edge, and symmetrically for the *dual edge function*. In figure 5.1 $\text{primal}(u^+, v_1^+) = (c, d)$, while $\text{dual}(b, a) = (v_4^+, u^+)$.

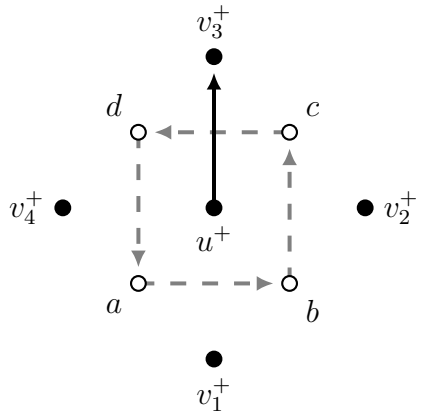


Figure 5.1: The directed edges of a face spin counterclockwise around its dual node.

This construction is useful because it can relate dual nodes to primal edges, and relate groups of dual nodes to each other. We can now easily talk about Hamiltonian 2×2 squares in a precise way:

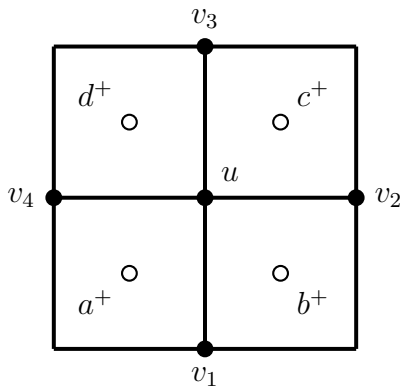


Figure 5.2: A sort of solid grid graph compass

For example, Hamiltonian 2×2 having no 4-full squares means at most 3 of the nodes a, b, c, d can be present in any one 2×2 square.

5.2 Constraints

5.2.1 Mazewalking

The tale of the computer scientist stuck in a maze is a well-known acclaimed story, and few haven't heard of the dazzling chemistry between characters such as 'her right hand' and 'the wall', who supported our protagonist as they went forward and traced the outline of the maze to find the exit. We unfortunately don't have the time to give a more detailed overview of this classic work of literature, but will nevertheless cite its essential teachings in our discussion.

More precisely, one can trace simple cycle going counterclockwise by holding their right hand to the boundary and walking forward. Because they will eventually return to their starting point, this means an interior point of a counterclockwise cycle sees the boundary as moving counterclockwise around it, and by symmetry, an exterior point sees the boundary as moving clockwise.

For solid grid graphs, a directed edge is counterclockwise on exactly one face, so we can immediately deduce the presence of an interior face given a directed edge in any step of a longest cycle algorithm. Hence, we say such a directed edge is an *oriented edge* and that it *orients a face*.

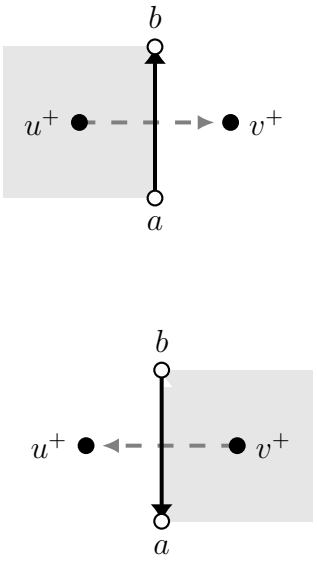


Figure 5.3

In figure 5.3, we say that the edge (a, b) orients u^+ as an interior face and that (b, a) orients v^+ as an interior face. These already produce a very useful family of *mazewalking constraints*, the first of which is that a counterclockwise edge e implies the interior face u^+ (eq. 5.1). In the same vein, a clockwise edge q implies the face u^-. Here we use the indicator notation as before, where $X_{u^+} = 1$ if u is an interior face of the cycle and $X_e = 1$ if e is an edge on the cycle.

$$X_e \leq X_{u^+} \tag{5.1}$$

(for any counterclockwise edge e on u)

$$X_q \leq X_{u^-} \tag{5.2}$$

(for any clockwise edge q on u)

Using superscript $+$ and $-$ is just notational convenience, since they mean mutually exclusive things, i.e. $X_{u^-} = 1 - X_{u^+}$. For practical performance reasons, it might be useful to add as many explicit constraints as possible derived from this equality, but to save space we won't include all of them here.

$$X_q \leq 1 - X_{u^+}$$

5.2.2 Rules of the dual

In previous chapters, we laid out the rules of the dual graph and how they relate to the primal graph. One of these rules is that neighboring dual nodes on the interior of a longest cycle can't have a primal edge between them. On the other hand, then, an exterior dual node and interior dual node have to have a primal edge between them. Because each edge is counterclockwise only to one face, we know which way such an edge would go (or not go) in each of these situations. If, as in figure 5.3, (a, b) was more likely to be present than (b, a), then u^+ would be more likely to be present than v^+.

$$X_{u^+} - X_{v^+} = X_{(a,b)} - X_{(b,a)}$$

In general, the above encodes for the relationships between dual node neighbors and the primal edge shared between them.

$$X_{u^+} - X_{v^+} = X_{\text{primal}(u^+,v^+)} - X_{\text{primal}(v^+,u^+)} \tag{5.3}$$

We've talked about how primal edges imply interior faces and vice versa. However, there is an important edge case where no relationship necessarily exists, namely when an interior face has 4 neighboring interior faces and 0 primal edges. Since a face with 4 interior neighbors is always interior, we need only the following constraint to account for this case.

$$\sum_{v^+ \in N(u^+)} X_{v^+} \leq X_{u^+} + 3 \tag{5.4}$$

5.2.3 2×2 squares

We can now precisely and succinctly convey the 2×2 square constraints as a set of linear constraints, and in fact have already done so while introducing the compass functions. We only need take the diagonals of each primal node to cover the interior of a graph's dual, so adding these constraints centered on each primal node u as in figure 5.1 is sufficient to ensure each 2×2 square of a graph is constrained. Not all nodes in a 2×2 square need to be in the interior of the dual (the first 2×2 square in 4.6 is an example), but at least one should be.

$$X_{a^+} + X_{b^+} + X_{c^+} + X_{d^+} \leq 3 \tag{5.5}$$

$$(X_{a^+} + X_{c^+}) - (X_{b^+} + X_{d^+}) \leq 1 \tag{5.6}$$

$$(X_{b^+} + X_{d^+}) - (X_{a^+} + X_{c^+}) \leq 1 \tag{5.7}$$

Equation 5.5 ensures the square includes less than 4 faces, while 5.6 and 5.7 together ensure that if two faces are diagonal to each other, exactly one of their shared neighbors is also present.

5.2.4 Connectivity

We've given constraints that can give cycles on solid grid graphs. However, we haven't yet in any part of this thesis given a constraint that ensures the solidness property of section 4.1. This can be done by ensuring that the dual solved for gives a single component cycle, which is the same general issue addressed in the linear programming preliminary 2.19.

Fortunately, our approach here will be even simpler than the one in the example, since we can be more picky about which node we choose to be the start of the cycle (assuming our earlier analysis of 2×2 squares is correct). Any node not on the boundary is part of the longest cycle, so we can choose any of these to be the start, saving us the overhead of another set of discrete variables.

Hence, our time constraints are similar to 2.19. In the starting node's case (which we add as an initialization step), the time of the outgoing edge is 1 and the time of the incoming edge is ℓ.

$$X_v = 1 \tag{5.8}$$

$$\sum_{e \in \text{out}(v)} t_e = 1 \tag{5.9}$$

$$\sum_{e \in \text{in}(v)} t_e = \ell \tag{5.10}$$

(for a single distinguished v not on the boundary)

Otherwise, the time of an outgoing edge is 1 added to the time of the incoming edge.

For each vertex v that is not distinguished we require that

$$\sum_{\text{out}(v)} t_e = \sum_{\text{in}(v)} t_e + X_v \tag{5.11}$$

5.2.5 Hamiltonian solid grid graphs

Since we aim to show that a longest cycle covers a Hamiltonian solid grid graph, we add the constraint that every node covered by an interior face of a longest cycle is included in the solution. This prevents the 4-full square case, where a primal node lies on an interior face but is not included.

$$\begin{aligned} X_{u^+} &\leq X_a \\ X_{u^+} &\leq X_b \\ X_{u^+} &\leq X_c \\ X_{u^+} &\leq X_d \end{aligned} \tag{5.12}$$

5.2.6 Global constraints

One of the main supporting factors in the claim that a MILP relaxation of this program correctly solves SGG-LCP is a set of constraints that require all non-integral variables to sum to an integer. Our general setup gives us a very good opportunity to exploit this, namely by knowing the number of edges and nodes to be covered as ℓ.

Constraint 5.13 applies the equation described in section 4.3. Constraint 5.14 reflects our hypothesis that only boundary nodes are excluded by a longest cycle. Letting $I = \{\text{dual nodes on the interior of } G^+\}$ and $B = \{\text{primal nodes on the boundary of } G\}$,

$$2 \sum_{v^+ \in I} X_{v^+} = \ell - 2 \tag{5.13}$$

$$|B| - \sum_{v \in B} X_v = |V| - \ell \tag{5.14}$$

5.3 ILP

We now have enough information to formulate an ILP for the longest cycle problem that exploits the unique structure of solid grid graphs. As with the (2.19) linear programming preliminary, we need only find satisfying integer solutions, and don't need to optimize under an objective function. Unlike the preliminary, all variables except the ones corresponding to dual nodes are continuous.

$$\begin{aligned} &\text{find } x \\ \text{s.t. } &\mathbf{A}x \leq \mathbf{b} \\ \text{under } &\text{(2.2) - (2.3.3) except (2.6)} \\ &\text{(5.1) - (5.14)} \\ \\ &t_e \in \mathbb{R}_{\geq 0} \\ &X_e \in \mathbb{R}_{[0,1]} \\ &X_{v^+} \in \{0, 1\} \end{aligned} \tag{5.15}$$

We only need the dual node indicators X_{v^+} to be integer because of constraint 5.3, which forces the difference between each primal edge and its twin $X_e - X_q$ to have values -1, 0, or 1, and constraint 2.4, which prevents both edges from being equal to 1. The integrality of the solutions to t_e follows from the integrality of the primal edges. Hence, the integrality of the dual node indicators is sufficient for the integrality of the entire solution, and LCP-SGG is essentially the problem of finding integer values of X_{v^+} satisfying this LP.

5.4 Experiments

5.4.1 Psuedocode - testing 5.15

Data: $C \leftarrow$ an MILP; $G \leftarrow$ a solid grid graph
Result: the length of the longest cycle of G
begin
  $\ell \leftarrow |V(G)|$;
  while C *has no integer solutions for a cycle of length* ℓ *on* G **do**
    $\ell \leftarrow \ell - 1$
  end
  output ℓ.
end

Running this algorithm for $C \leftarrow$ (5.15) and $C \leftarrow$ (2.19) on a large number of examples (≥ 1000) results in the same answer for every G. This gives very strong empirical evidence towards our hypothesis that the longest cycle of a solid grid graph without forced 4-full squares covers all but a set of boundary nodes. Some of these examples are displayed on the next pages.

CPLEX for Python was used to implement the ILPs described in this thesis, and a custom planar graph visualization library was written to render examples directly into TikZ. The code used for this can be found in a .zip file as part of the digital version of this document.

5.5 Future work - towards polynomiality via LP

In practice, relaxing the integrality of X_{v^+} and solving the resulting LP often gives solutions where some variables are integer, even if by coincidence. When an objective function is used (e.g. maximizing the number of interior dual nodes of odd parity), there can even be the sense that the solutions are within rounding error, where an interior node X_{u^+} has a value of $\frac{3}{4}$ and its neighbor X_{u^+} has a value of $\frac{1}{4}$, and sort of giving a higher confidence that X_{u^+} is a dual node on the interior of the cycle and X_{v^+} isn't. In these cases, a polytime algorithm could be to round each of these values to 1 or 0 as needed.

However, such rounding techniques could also be applied more generally to the (2.19) ILP, without needing the solid grid graph structure. One direction for future research could be to explore rounding techniques (or other iterative methods) specifically for solid grid graphs. Intutitively it seems likely that SGG-LCP is in P, but proving this to be true could be part of the research direction described above.

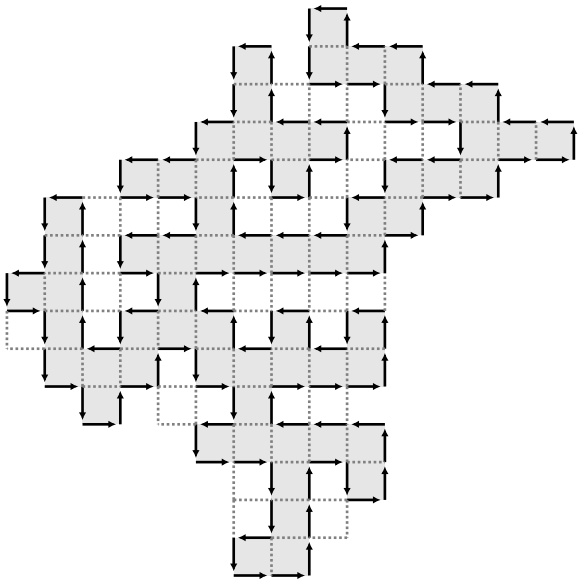


Figure 5.4: SGG-LC no. 100

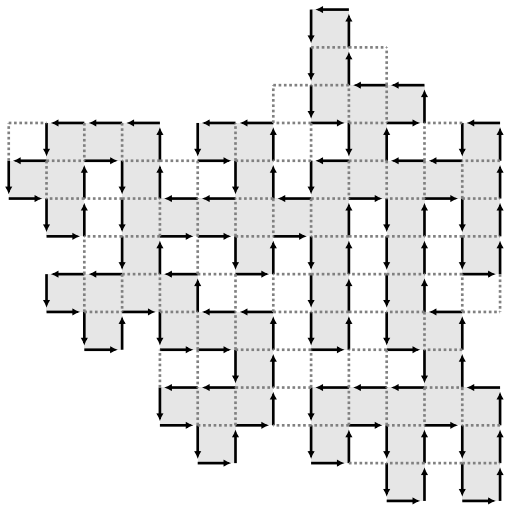


Figure 5.5: SGG-LC no. 101

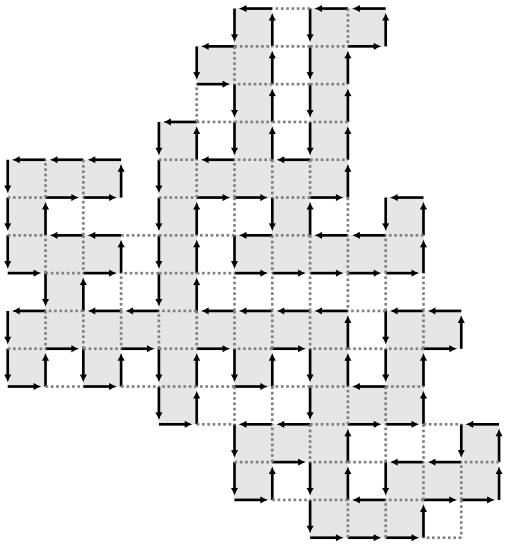


Figure 5.6: SGG-LC no. 102

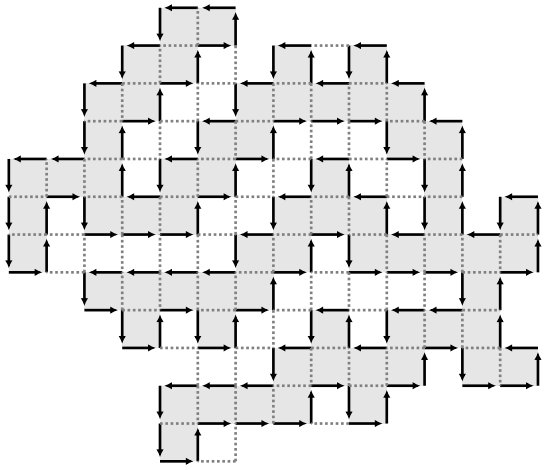


Figure 5.7: SGG-LC no. 103

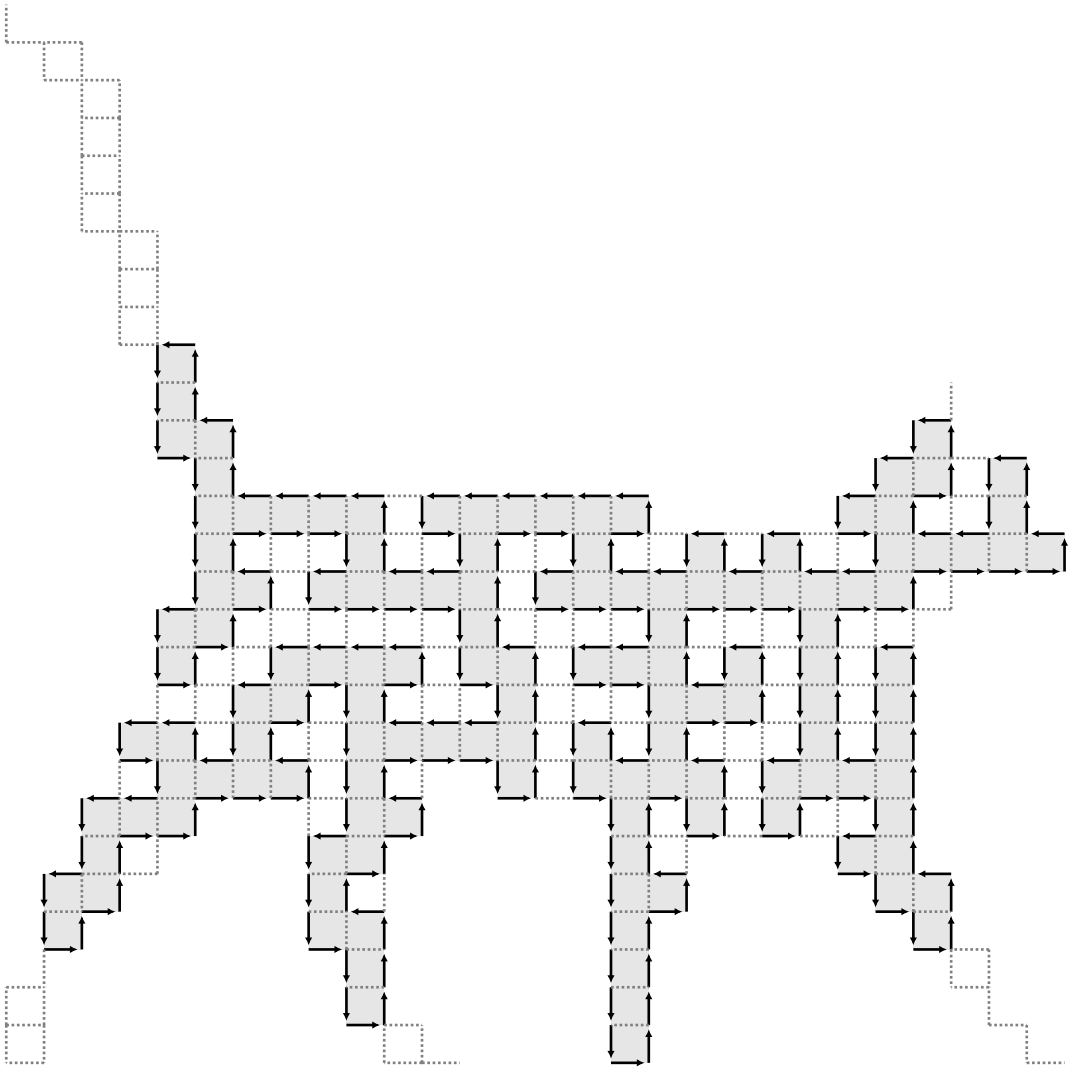


Figure 5.8: The longest cycle on a cat in solid grid graph form.

References

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction To Algorithms*. The MIT Press, 2009 third ed.

Fekete, Sándor P., Rieck, Christian, and Scheffer, Christian. "On the Travelling Salesman Problem in Solid Grid Graphs." 2017.

Grötschel, Martin, Lovász, László, and Schrijver, Alexander. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, 1991 second ed.

Itai, Alon, Papadimitriou, Christos H., and Szwarcfiter, Jayme Luiz. "Hamilton Paths in Grid Graphs." *SIAM Journal on Computing* 11.4 (1982): 676–686.

Sipser, Michael. *Introduction to the Theory of Computation*. CENGAGE Learning, 2006 second ed.

Umans, C. and Lenhart, W. "Hamiltonian cycles in solid grid graphs." *Proceedings 38th Annual Symposium on Foundations of Computer Science*. 1997 496–505.