University of Alberta

Finding Hamiltonian Cycles: Algorithms, Graphs and Performance

by

Basil Vandegriend

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

Department of Computing Science

Edmonton, Alberta
Spring 1998

Finding Hamiltonian Cycles: Algorithms, Graphs and Performance

Basil Vandegriend

Abstract

In this thesis we examine various aspects of the Hamiltonian cycle problem. Our central thesis is to study the interaction between the algorithm, the graph and the observed performance. Our survey of Hamiltonian cycle algorithms summarizes and classifies the different techniques and heuristics used in the literature. We discuss various design issues concerning these different algorithms and introduce some new algorithmic techniques. Our investigation of hard graphs for the Hamiltonian cycle problem obtains new results on random graph classes which show them to be easy for a backtrack Hamiltonian cycle algorithm. We introduce and obtain proofs for a generalization of the knight's tour problem which we show produces hard graphs for Hamiltonian cycle algorithms. Throughout the thesis, we provide theoretical and experimental evidence identifying and characterizing the interaction between algorithm, graph and performance.

Contents

List of Figures

List of Tables

List of Symbols

$G = (V, E)$, $|V| = n$, $|E| = m$: Specifies a graph G with vertex set V and edge set E, with size n and m respectively.

$V(G)$: The set of vertices of graph G.

$E(G)$: The set of edges of graph G.

$\delta(G)$: The minimum degree of the vertices of graph G.

$\overline{d}(G)$: The mean degree of the vertices of graph G.

$c(G)$: The number of components of graph G.

$d(x)$: The degree of vertex x.

$d^+(x)$: The outbound degree of vertex x in a directed graph.

$d^-(x)$: The inbound degree of vertex x in a directed graph.

$d_Q(x)$: The degree of vertex x with respect to subgraph Q (i.e. the number of edges incident on x and another vertex in Q).

$N(x)$, $N(X)$: The set of neighbours of a single vertex x or a set of vertices X. A vertex a is a neighbour of vertex b if $(b, a) \in E$. $N(X)$ is defined to be open: it does not include vertices in X.

$N^+(x)$, $N^-(x)$: The set of outbound and inbound neighbours, respectively, of a vertex x in a directed graph. $N^+(x) = \{y|(x, y) \in E\}$ and $N^-(x) = \{z|(z, x) \in E\}$.

$s(P), e(P)$: The starting and ending vertices of a path P.

$P^-(x)$, $P^+(x)$: The vertex before and after x in the path P, respectively. Note that before and after are with respect to the start and end vertices in the path. $P = \{s(P), \ldots, P^-(x), x, P^+(x), \ldots, e(P)\}$

$\oplus$: The exclusive-or operation on sets. $C = A \oplus B$ means that an element is in C only if it is A or B but not in both A and B.

$G_{n,p}$, $G_{n,m}$, $M_{n,m}$, $R_{n,d}$: Various random graph models. See Sections 2.2.1 and 2.2.2.

forced edge: An edge incident on a degree 2 vertex in a graph. Any Hamiltonian cycle must contain all forced edges (see Theorem 1).

forced path: A path made of forced edges. Formally a path $P = \{p_1, p_2, \ldots, p_k\}$ with (p_i, p_{i+1}) a forced edge for $1 \leq i < k$.

non-Hamiltonian edge: An edge which cannot be in any possible Hamiltonian cycle.

bipartite graph: A graph in which the vertices can be partitioned into two sets, with no edges existing between vertices in the same set.

whp: The term *whp* (with high probability) means that a graph G has property Q with probability $p = 1 - o(1)$ as $n \to \infty$.

$O(f(n))$: Big-Oh notation: "$O(f)$ is the set of functions $g : \mathcal{N} \to \mathcal{R}^*$ such that for some $c \in \mathcal{R}^+$, $n_0 \in \mathcal{N}$, $g(n) \leq cf(n)$ for all $n \geq n_0$." [2, page 29].

Chapter 1

Introduction

The Hamiltonian cycle problem is a NP-C problem which involves finding for a graph a cycle that contains all the vertices of the graph. In everyday terms, imagine a salesperson[1] who must visit every city within her sales area. She only wants to visit each city once, and wants to finish back at the city she started at because that is where she lives. Unfortunately for her, she must fly between the different cities (to save time), and not all the cities have flights connecting each other. Thus, she must devise a route that visits all the cities once and returns to her home city using the flights that are available. This is the Hamiltonian cycle problem. Formally for a graph $G = (V, E), |V| = n$, the Hamiltonian cycle problem is to find a cycle $C = (v_1, v_2, \ldots, v_n)$ such that $v_i \neq v_j$ for $i \neq j$, $(v_i, v_{i+1}) \in E$ and $(v_n, v_1) \in E$.

The Hamiltonicity of a graph (whether or not it has a Hamiltonian cycle) is an important property often used by graph theorists. Much work has been done on the Hamiltonicity of random graphs and specific graph classes. The knight's tour problem, a subset of the Hamiltonian cycle problem, has received considerable attention from mathematicians as well. However, in general it seems that researchers in Computing Science have neglected the development of practical, experimentally verified algorithms for the Hamiltonian cycle problem and focused instead on other NP-C problems such as SAT, graph coloring and the travelling salesmen problem (which the Hamiltonian cycle problem is a subset of). We are not denying that many good reasons exist for studying these other problems. However, we believe that the Hamiltonian cycle problem is complex and hard (it is NP-C) and worthy of study by itself.

Therefore in this thesis we examine the Hamiltonian cycle problem. There are two main components to any such examination: the algorithms used to solve the problem and the characteristics of the different problem instances. We will examine a variety of Hamiltonian

[1]Without loss of generality, we assume the salesperson is female.

cycle algorithms and will also investigate the existence of hard graphs for which it is difficult to determine Hamiltonicity. In our research, our primary thesis is that for the Hamiltonian cycle problem there exists a complex interaction between the algorithms being used, the properties of the graphs being solved, and the performance of those algorithms. Throughout this thesis we investigate and characterize this interaction. The major ramification of our thesis is that the two topics of algorithms and hard graphs cannot be effectively studied in isolation. Our exploration of design issues for Hamiltonian cycle algorithms will need to consider the nature of the graphs the algorithms will be used on. Our investigation of graphs that are difficult to solve will need to account for the algorithms used to solve them.

We begin in Chapter 2 with an in-depth survey of the previous work done on the Hamiltonian cycle problem. In particular, we focus on the different Hamiltonian cycle algorithms that have been devised by different researchers and distill out the different techniques and heuristics that have been employed.

In Chapter 3 we continue our investigation of Hamiltonian cycle algorithms by exploring the design issues involved. In particular, we examine and critique the various techniques and heuristics surveyed in Chapter 2, and devise a few new techniques and heuristics. We show how many design decisions are contingent upon the type of graph (or properties of the graph) that the algorithm will be run on.

Chapter 4 is a bit of a detour as we explore generalizations to the knight's tour problem. Our goal is to devise graphs which will be difficult for our Hamiltonian cycle algorithms to solve. We first review the literature addressing this problem. We then devise a new generalization, the generalized knight's circuit problem, and present theoretical and experimental results concerning the Hamiltonicity of instances of this problem.

Chapter 5 is the capstone of the thesis. In this chapter we search for hard Hamiltonian cycle graphs. We examine different graph models and sets including the generalized knight's circuit graphs devised in the previous chapter. Phase transitions for the Hamiltonian cycle problem can help identify hard instances. We perform a short review of some of the previous work in this area. We obtain new results concerning phase transitions and a particular random graph model $G_{n,m}$ that refine the findings of previous researchers. Throughout the chapter, we explore how the determination of which graphs are hard is effected by the algorithm being used. We present experimental evidence that clearly identifies graphs that are hard for one algorithm and easy for another.

We present our conclusions and a discussion of future work in Chapter 6.

Chapter 2

Survey of Work on Hamiltonian Cycles

2.1 Introduction

The literature contains many references that refer to Hamiltonian cycles. The purpose of this chapter is to give an organized overview of this work and then review in more detail the work most relevant to this thesis.

Since the Hamiltonicity of a graph is an important property utilized often by graph theorists, much research has been done on the Hamiltonicity of various restricted graph classes, and algorithms for finding Hamiltonian cycles for these graphs. Random graphs in particular have received much attention from researchers. Section 2.2 describes the work done on determining the Hamiltonicity of random graphs. Also of interest to us from graph theory are the theorems which describe necessary conditions for the existence (or non-existence) of Hamiltonian cycles. These theorems are presented in Section 2.3.

The Hamiltonian cycle algorithms developed for arbitrary graphs are of particular interest to us because they tend not to make any assumptions about the structure of the graph and thus are applicable to any graph, unlike most of the algorithms developed for particular graph classes. An overview of these Hamiltonian cycle algorithms is given in Section 2.4, and specific algorithms are presented in Section 2.5.

2.2 Hamiltonian Cycles and Random Graphs

In this section we examine the work of researchers concerning the Hamiltonicity of random graphs. In Section 2.2.1 we refer to two general, random, undirected graph models. In Section 2.2.2 we examine other random graph models, including sparse (low edge density) random graphs and regular random graphs.

2.2.1 General Random Graph Models

In this section we examine previous work on two random graph models, $G_{n,p}$ and $G_{n,m}$. Basically, these models define a probability distribution over all possible graphs. The $G_{n,p}$ model defines a distribution for a sample space of graphs with n vertices. Each of the $\binom{n}{2}$ possible edges has an independent probability p of existing. In other words, the probability of a particular graph G with n vertices and q edges $= p^q(1-p)^{\binom{n}{2}-q}$. [23, pp. 6–7] The $G_{n,m}$ model defines a distribution for a sample space consisting of graphs with n vertices and m edges. The probability of a particular graph $G = \binom{\binom{n}{2}}{m}^{-1}$. [23, pp. 8–9]

Pósa was one of the first to consider the Hamiltonicity of random graphs. In [24] he proved that a random graph $G \in G_{n,m}$ with $m = cn \log n$ *whp* [1] contains a Hamiltonian circuit (if c is sufficiently large). Due to the foundational nature of Pósa's work, we examine his proof in detail in Section 2.2.3.

Komlós and Szemerédi in [21] extended this result as follows. For $m = \frac{n}{2}(\log n + \log\log n + c_n)$, they proved that

$$\begin{aligned}\lim_{n\to\infty} P(G_{n,m} \text{ has a Hamiltonian cycle}) &= \begin{cases} e^{-e^{-c}} & \text{if } c_n \to c \\ 1 & \text{if } c_n \to \infty \end{cases} \\ &= \lim_{n\to\infty} P(\delta(G_{n,m}\ ) \geq 2)\end{aligned}$$

Bollobás in [5] uses a random graph process in which a graph of n vertices (that starts with no edges) has edges between random vertices added one at a time. He proves that *whp* the first edge added which increases the minimum degree of the graph to 2 also makes the graph Hamiltonian.

Later work by other researchers focused more on proving the performance of algorithms for finding Hamiltonian cycles on graphs with a sufficiently large edge density. In this section we will only summarize the various results; Section 2.5 will describe the algorithms in detail.

Shamir [28] proved that for $G \in G_{n,p}$, $p = \frac{1}{n}(\log n + c \log\log n)$, $c > 3$ his algorithm *whp* finds a Hamiltonian path in $O(n^2)$ time.

Angluin and Valiant [1] provide algorithms UHC and DHC, which find Hamiltonian cycles in undirected and directed graphs respectively. These algorithms find a solution *whp* for $G_{n,m}$, $m \geq cn \log n$. A transformation between the $G_{n,m}$ and $G_{n,p}$ graph models is also given to show that the results also apply to the $G_{n,p}$ model. Both the UHC and DHC algorithms have run times of $O(n(\log n)^2)$.

[1] We define the term *whp* (with high probability) to mean that a graph G has property Q with probability $p = 1 - o(1)$ as $n \to \infty$.

Bollobás, Fenner and Frieze in [4] present a polynomial algorithm called HAM. On graphs in $G_{n,m}$, $m = \frac{n}{2}(\log n + \log\log n + c)$, the algorithm *whp* will find a Hamiltonian cycle if one exists. HAM runs in $O(n^4)$ time. As a corollary, they show that the probability of failure of this algorithm is so small that if such instances are solved by a standard backtrack algorithm then the Hamiltonian cycle problem for this class of graphs can be solved in polynomial expected running time. In [15], Frieze presents his DHAM algorithm, which is an extension of the results of [4] to directed graphs. DHAM has a running time of $O(n^{1.5})$.

Thomason [30] presents a linear time algorithm for finding a Hamiltonian cycle of a graph in $G_{n,p}$, $p \geq n^{-\frac{1}{3}}$. The expected running time is $O(n/p)$, and the storage required is $O(n)$. The computational model used assumes the existence of an oracle which can determine whether any two specified vertices are adjacent in $O(1)$. An extension to this algorithm for directed graphs is also given which requires $O(n^{2.5}p^{-1.5})$ running time.

2.2.2 Sparse and Regular Random Graph Models

In this section we examine the work done on more specialized random graph models. The $M_{n,m}$ graph class (referred to as the m-out model) establishes a distribution over undirected multigraphs of n vertices. Basically, for each of the n vertices, m edges are added to randomly selected vertices. More formally, for each $v \in V_n = \{1, 2, \ldots, n\}$ make m random choices $c(v, i) \in V_n, i = 1, 2, \ldots, m$. Then $M_{n,m} = (V_n, E(n, m))$, where $E(n, m) = \{(v, c(v, i)) : v \in V_n, 1 \geq i \geq m, v \neq c(v, i)\}$ [16]. The $R_{n,d}$ graph class specifies a distribution over graphs of n vertices for which each regular graph of degree d has an equal probability of being selected.

Frieze in [16] presents Hamiltonian cycle algorithms for these constant average degree graph models. He produces an $O(n^3 \log n)$ time algorithm which *whp* finds a Hamiltonian cycle in $M_{n,5}$ and another $O(n^3 \log n)$ time algorithm which *whp* finds a Hamiltonian cycle in $R_{n,85}$. Robinson and Wormald improve this second result in [26] to show that *whp* d-regular graphs have Hamiltonian cycles for any fixed $d \geq 3$. In [17] this non-constructive proof is extended to an algorithmic version, in which Frieze, et al. show that for fixed $d \geq 3$ and $G \in R_{n,d}$, there exists a polynomial time algorithm which constructs a Hamiltonian cycle *whp*.

Cooper and Frieze [11] prove that the $M_{n,4}$ graph model is Hamiltonian *whp*. They also examine a $D_{k\text{-in},l\text{-out}}$ graph model for digraphs, similar to the $M_{n,m}$ model. They prove that $D_{2\text{-in},2\text{-out}}$ is Hamiltonian *whp*.

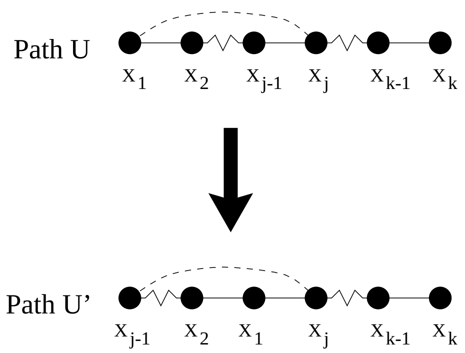

Figure 2.1: The rotational transformation.

2.2.3 Pósa's Proof

In [24] Pósa proves that a random graph $G \in G_{n,m}$ with $m = cn \log n$ *whp* contains a Hamiltonian cycle. In this section we examine his proof in detail.

A key concept introduced by Pósa is the rotational transformation, which is defined as follows. Let G be an arbitrary graph, and $U(x_1, x_2, \ldots, x_k)$ a path in G. If there is an edge $(x_1, x_j), 1 < j < k$ then the path U can be transformed to $U'(x_{j-1}, x_{j-2}, \ldots, x_1, x_j, x_{j+1}, \ldots, x_k)$ (see Figure 2.1) where the order of the first $j-1$ vertices is reversed.

In Lemma 1 of his proof, Pósa defines the path U to be of maximum length in G. By applying the rotational transformation to U multiple times, we obtain transformed paths U', U'', etc., for which x_k always remains an endpoint. Pósa defines the set of vertices H to contain all the other endpoints of these paths. So x_1 is in H, while x_k is not. Pósa defines another set X which consists of those vertices which are:

- not x_k
- not a member of H
- on the path U and not adjacent to a vertex in H

Note that all vertices of G not in U are in X.

Using these definitions of the sets H and X, Pósa proves that a vertex of H and a vertex of X cannot be joined by an edge. In Lemma 2 of his proof, Pósa considers a random graph $G \in G_{n,p}$, $p = c \log n / n$ and two disjoint subsets of particular (bounded) size. He proves that the probability of the existence of an edge joining the two subsets approaches 0 as $n \to \infty$ (for $c \geq 30$).

Theorem 1 states that for a random graph G as defined above, G *whp* contains a Hamiltonian path. The proof of this theorem is as follows.

Consider the graph $G(x)$, which denotes the graph with $n-1$ vertices obtained from G by erasing x. We want to calculate the estimation of the probability of $L(x)$, a path of maximum length in G which passes through x.

Choosing a particular, arbitrary path U of maximum length in $G(x)$, we define the subsets H and X (from Lemma 1) for this graph. There are two possible cases. In the first case, H is small enough that Lemma 2 can be used. The second case involves larger sizes of H. If x is adjacent to any element of H then the path U can be transformed to a path U' (using the rotational transformation) with an endpoint adjacent to x. This would allow the path U to be extended to include x. However, the set H is large enough that the probability of a vertex x existing not adjacent to H approaches 0.

Thus, there are two possible events that prohibit the formation of a Hamiltonian path. The probability of the first event is calculated in Lemma 2. The second event's probability is calculated in the proof for Theorem 1, and is also shown to approach zero as n increases to infinity.

Thus from [24], "with probability tending to 1, every path of maximum length in G passes through all points of G".

Theorem 2 states that for a random graph $G \in G_{n,p}$, $p = c_1 \log n/n$, c_1 sufficiently large, G has *whp* a Hamiltonian cycle. The proof is as follows.

Let us construct two random graphs G_1 and G_2 on the same set of n vertices, with edge probability $(c \log n)/n$ and $(\log n)/n$ respectively. Let G be the union of the two sets of edges. Thus G is also a random graph, with edge probability $((\log n)/n)(c+1-(c \log n)/n)$.

By Theorem 1, G_1 contains a Hamiltonian path U with probability approaching 1. We define the sets H and X as per Lemma 1. If G contains no Hamiltonian cycle, then one of the following three events of small probability must occur.

- There is no Hamiltonian path in G_1. (Small probability by Theorem 1.)
- H is small enough that Lemma 2 holds, and the probability of this event tends to 0.
- H is too large for Lemma 2. In this case x_n (the endpoint of the Hamiltonian path on G_1) cannot be joined by edges in G_2 to elements of H (since any edge from x_n to an $h \in H$ would allow the construction of a Hamiltonian cycle, by transforming the path U to a path U' with endpoints x_n and h, which along with the edge (x_n, h) makes a cycle). The probability of the event that there is no edge in G_2 between x_n and the elements of H tends to 0.

Thus the overall probability of no Hamiltonian cycle existing tends to 0.

Theorem 3 states the final proof: a random graph $G \in G_{n,m}$ with $m = cn \log n$ *whp* contains a Hamiltonian cycle. This theorem is proven by constructing a graph using an edge probability of $(c_1 \log n)/n$ (so that Theorem 2 holds). The graph is modified to get the correct number of vertices as stated in Theorem 3, and it is shown that the results of Theorem 2 (probability of a Hamiltonian cycle existing approaches one) still holds for those modifications.

2.3 Theorems Concerning Hamiltonian Cycles

This section presents various theorems from graph theory which describe necessary conditions for the existence (or non-existence) of Hamiltonian cycles. All of the theorems are accompanied with either proofs, or references to proofs. Unfortunately, due to the obvious nature of some of the theorems, the originator of each theorem is not always known, and most likely is not unique. For additional information on this subject, see various graph theory texts such as [6] or [31].

Theorem 1 *In a graph with a Hamiltonian cycle the degree of each vertex must be* ≥ 2. *If a vertex has exactly degree 2, then both the edges incident to that vertex must be in any cycle.*

Proof. The statements in the theorem follow directly from the definition of a Hamiltonian cycle. □

Theorem 2 *If a vertex has 3 neighbours of degree 2, that graph cannot contain a Hamiltonian cycle.*

Proof. From Theorem 1, the two edges incident on a degree 2 vertex must both be in any cycle. Assume a cycle exists. Then for the vertex with 3 degree 2 neighbours, each of its edges is incident on a degree 2 vertex, so each of these edges must be in the cycle. With 3 edges in the cycle incident to this vertex, it must occur more than once in the cycle. This is a contradiction, so no Hamiltonian cycle is possible. □

Theorem 3 *If vertex v has 2 neighbours a and b which are both of degree 2, then all edges $(v, x), x \neq \{a, b\}$, are not in any possible Hamiltonian cycle.*

Proof. Edges (v, a) and (v, b) must be in any Hamiltonian cycle by Theorem 1. Since any vertex only appears once in a Hamiltonian cycle, no other edge in the cycle can have v as an endpoint. Thus other edges from v cannot be part of any cycle. See [19]. □

Theorem 4 *If a path $U = u_1, u_2, \ldots, u_k$ exists with $k < n$ and $d(u_2), \ldots, d(u_{k-1}) = 2$ then the edge (u_1, u_k) cannot be in any Hamiltonian cycle.*

Proof. We call path U a forced path, since every Hamiltonian cycle must follow the path from u_1 to u_k since each intermediate vertex is of degree 2 (see Theorem 1). Since adding edge (u_1, u_k) makes a cycle shorter than a Hamiltonian cycle ($k < n$), such an edge cannot be part of any Hamiltonian cycle. $\square$

Theorem 5 *If S is a cutset of graph $G = (V, E)$, $S \subseteq V$ then no Hamiltonian cycle can exist when $c(G \setminus S) > |S|$ and no Hamiltonian path can exist when $c(G \setminus S) > |S| + 1$.*

Proof. Let C be a Hamiltonian cycle in G. For any cutset S, $C \setminus S$ must have at most $|S|$ components. Thus, $G \setminus S$ must have at most $|S|$ components when a Hamiltonian cycle exists. The proof is similar for Hamiltonian paths. See [6]. $\square$

Note that this theorem is very similar to the property of toughness in graph theory. A graph is t-tough if $|S| \geq tc(G \setminus S)$, for every cutset $S \subseteq V$. The toughness of G is the maximum t such that G is t-tough [31, page 220]. Theorem 5 proves that a toughness of at least 1 is necessary for a graph to be Hamiltonian.

Corollary 5.1 *If the graph G is bipartite with bipartition (X, Y) and $|X| \neq |Y|$ then no Hamiltonian cycle exists.*

Proof. Assume $|X| < |Y|$ without loss of generality. Then X is a cutset. The removal of X leaves $|Y|$ components. Since $|Y| > |X|$ no Hamiltonian cycle exists by Theorem 5. $\square$

Corollary 5.2 *For any independent set S of graph $G = (V, E)$ with neighbourhood $N(S)$, $V \setminus S \cup N(S) \neq \emptyset$, no Hamiltonian cycle can exist when $|N(S)| \leq |S|$.*

Proof. The neighbourhood $N(S)$ is a cutset. The removal of the cutset will produce at least $|S| + 1$ components (the members of the independent set, plus the remainder of the graph, which is specified to be non-empty). From Theorem 5, no Hamiltonian cycle can exist if $|S| + 1 > |N(S)|$. $\square$

Theorem 6 *If $G = (V, E)$ is simple and $|V| \geq 3$ and $\delta(G) \geq |V|/2$ then G is Hamiltonian.*

Proof. See [6, pp. 54–55]. $\square$

Theorem 7 *If $G = (V, E)$ is simple and $a, b \in V$ and $(a, b) \notin E$ and $d(a) + d(b) \geq n$ then G is Hamiltonian iff $G' = (V, E')$ is Hamiltonian, where $E' = E \cup \{(a, b)\}$.*

Proof. See [31, page 222]. □

Theorem 8 *A simple graph is Hamiltonian iff its closure* [2] *is Hamiltonian.*

Proof. See [6, pp. 55–57]. □

Corollary 8.1 *A simple graph G with $n \geq 3$ is Hamiltonian if the closure of G is complete.*

Proof. See [6, page 57] □

Theorem 9 *A simple graph G has a non-decreasing degree sequence $\{d_1, d_2, \ldots, d_n\}$ with $n \geq 3$. If there is no $m < n/2$ for which $d_m \leq m$ and $d_{n-m} < n - m$ then G is Hamiltonian.*

Proof. See [6, pp. 57–58]. □

2.4 An Overview of Hamiltonian Cycle Algorithms

In this section we present an overview of the work done on Hamiltonian cycle algorithms. There are two major classes of algorithms: polynomial time heuristic algorithms and backtrack algorithms. Section 2.4.1 and Section 2.4.2 respectively present an overview of these classes of algorithms. Section 2.4.3 presents a summary of the different algorithm techniques that have been utilized in the literature. Note that the actual algorithms are discussed in Section 2.5.

2.4.1 Heuristic Hamiltonian Cycle Algorithms

One common approach to tackling NP-C problems is to use some kind of heuristic algorithm. These algorithms are fast, running in linear or low-order polynomial time. They work by using heuristics – general rules of thumb – to guide their search for a solution. These heuristics may be simplifying assumptions about the problem, or may be restrictions on how to search (i.e. perform only local search). Randomization of the algorithm is another commonly-employed heuristic. While these heuristics permit the algorithm to execute quickly, the algorithm (usually) loses the ability to guarantee to find a solution, or determine if a solution exists.

Heuristic Hamiltonian cycle algorithms tend to have a similar structure. Figure 2.2 is an overview of the most common components of any heuristic Hamiltonian cycle algorithm. The various heuristic algorithms in the literature are differentiated by how they accomplish the tasks described in each of these components.

[2] The closure of G "is the graph obtained from G by recursively joining pairs of nonadjacent vertices whose degree sum is at least n until no such pair remains. " [6, page 56]

```
InitGraph: Perform initial analysis and pruning of the graph.
InitPath: Select an initial vertex to start the path representing a
    potential solution.
Do {
    GetNewNode: Find a new vertex that can be added to the path.
    If such a vertex is found {
        ExpandPath: Add the chosen vertex to the path and perform
            analysis and pruning on the graph. If this choice does not
            permit a Hamiltonian cycle to form, then remove the vertex
            from the path.
    }
    else {
        TransformPath: Transform the path so that a different node
            becomes the endpoint. If the current path cannot be
            successfully transformed, then return failure. If the path
            has been transformed too many times without being
            expanded, then return failure.
    }
    FormCycle: If the path being formed contains all the vertices in the
        graph then try to form a cycle. If a Hamiltonian cycle is formed,
        then return success. If no Hamiltonian cycle is formed, then
        optionally return failure.
} (end of do loop)
```

Figure 2.2: Outline of a heuristic Hamiltonian cycle algorithm.

2.4.2 Backtrack Hamiltonian Cycle Algorithms

One standard approach to solving NP-C problems such as Hamiltonian cycle is to use a backtrack algorithm, which evaluates (searches) all the potential solutions of the problem, looking for a valid solution. These algorithms typically employ pruning of some kind to restrict the amount of searching they need to do. However, in the worst case exponential work must be done. While these algorithms will always find a solution (if one exists), or will determine that no solution is possible, they usually take a long (sometimes intractable) amount of time to execute.

Backtrack Hamiltonian cycle algorithms share a similar recursive structure. Figure 2.3 is an overview of the essential components of any Hamiltonian cycle backtrack algorithm. The various backtrack algorithms we survey are differentiated by how they accomplish the tasks described in each of these components.

2.4.3 Summary of Algorithm Techniques

In this section we summarize the techniques used by the various Hamiltonian cycle algorithms described in Section 2.5. Table 2.1 lists each of the techniques along with the algorithm name and section in this chapter where we introduce and describe the technique.

The following list briefly describes each algorithm technique.

rotational transformation: This technique is used by heuristic algorithms to modify a path (same vertex set, different edge set) to change one of the endpoints.

directed rotational transformation: This technique is used by heuristic algorithms on directed graphs, and consists of two different transformations. The first transforms a path into a path plus a separate cycle, and the second transforms a path and separate cycle into a single path. For both transformations, the endpoint of the path changes.

cycle extension: This technique is used by heuristic algorithms to extend the length of a path. First, the path must also be a cycle (endpoints connected by an edge). Then, in a connected graph, one of the vertices v of this cycle must be joined to a vertex x outside the cycle. This vertex v becomes one of the endpoints of the path (by deleting an edge), and the path can then be extended to vertex x. This technique works on both directed and undirected graphs.

backtrack rotational transformation: This technique is used by heuristic algorithms when trying to modify a path's endpoints to obtain a vertex that is adjacent to an unvisited

```
BacktrackAlg() {
    InitGraph: Perform initial analysis and pruning of the graph.
    InitPath: Select an initial vertex v to start the path P
        representing a potential solution.
    Recurse(P, v)
    If Recurse returns success, then return P else return failure.
}

Recurse(Path P, endpoint vertex e) {
    While e has unvisited neighbours do {
        GetNewNode: Choose the next vertex x which is an unvisited
            neighbour of e. Add x to P.
        PruneGraph: Perform pruning on G. If the resulting graph does
            not permit a Hamiltonian cycle to form, then remove x
            from P and continue (at the start of the while statement).
        FormCycle: If P contains all the vertices in the graph then {
            Try to form a cycle.
            If a Hamiltonian cycle is formed then return success
            else remove x from P and continue (at the start of
            the while statement).
        }
        Backtrack: Recurse(P, x)
            If Recurse returns success, then return success
            else remove x from P.
    } (end of while)
    Return failure.
} (end of Recurse)
```

Figure 2.3: Outline of a backtrack Hamiltonian cycle algorithm.

vertex. The backtrack rotational transformation involves searching the possible paths in some systematic manner. The technique is essentially backtracking through the possible sets of rotational transformation that can be applied to the initial partial path. A dead end in one particular path no longer means the search will stop; instead the search will begin again from a different path. This technique is used until the current path is extended, or until some search termination criteria is reached.

crossover extension: This heuristic algorithm technique can be considered either an extension of the backtrack rotational transformation technique, or a formalized version of it. This technique tries to obtain a new ordering of the vertices in a path P that is also a cycle (endpoints connected by an edge), so that the cycle extension technique can be applied. The new ordering is obtained using a crossover. For a path P with endpoints u, v a crossover Q is a uv trail with $V(Q) \subseteq V(P)$ that produces a cycle C with $E(C) = E(P) \oplus E(Q)$ [3].

bypass extension: This heuristic algorithm technique is used when a path P cannot be extended by the previous techniques. It consists of finding a path using vertices not in P which can replace a section of P which is shorter than it, thus making P longer. To be more specific, the technique involves searching the subgraph $G - V(P)$ for a path R with endpoints a, b that are adjacent to vertices in P $(x \in N(a), y \in N(b))$. Let S be that portion of P between x and y. If $|R| > |S|$ then R can replace S in P and extend P.

chain extension: This heuristic algorithm technique is a combination of the bypass extension and crossover extension techniques. It consists of finding a path using vertices not in P and merging it with P to make P longer. First we search for a path R with endpoints a, b in the subgraph $G - V(P)$ and for which $\exists x, y \in P, x \in N(a), y \in N(b)$. Then we search for an xy trail Q which allows us to form a new path $P = P \oplus Q \oplus R$ (the merger of paths P and R).

low degree paths: This heuristic algorithm technique involves forming a set X of low degree vertices, and then embedding these vertices in paths, with endpoints that are not low degree. The idea is that the resulting graph contains mainly high degree vertices, making it much more likely to find a Hamiltonian cycle.

[3] The $\oplus$ symbol represents the exclusive-or operation on sets. $C = A \oplus B$ means that an element is in C only if it is A or B but not in both A and B.

Table 2.1: Summary of techniques for Hamiltonian cycle algorithms.

Technique Name	Algorithm	Section Described In
rotational transformation	Pósa's [24]	2.2.3
directed rotational transformation	DHC [1]	2.5.3
cycle extension	HAM [4]	2.5.4
backtrack rotational transformation	HAM [4]	2.5.4
crossover extension	LongPath [20]	2.5.9
bypass extension	LongPath [20]	2.5.9
chain extension	LongPath [20]	2.5.9
low degree paths	HideHam [7]	2.5.6
low degree first	DB2 [8]	2.5.7
multipath search	MultiPath [19]	2.5.11
pruning	MultiPath [19]	2.5.11

low degree first: This heuristic technique involves simply selecting the lower degree vertices first when forming a cycle.

multipath search: This technique for backtrack algorithms involves maintaining a list of segments (paths) which are initially formed from forced edges. At each point in the search a random endpoint from the segments is selected for expanding the paths. Segments are merged together when they share a common endpoint.

pruning: This general technique involves reducing the size of the search space, given a current graph and a partial solution. Many different types of pruning can be employed. One pruning method is to delete unneeded edges. Another type of pruning is to find forced edges and include them in the solution. A third form of pruning is to detect when no Hamiltonian cycle is possible for the current graph.

2.5 A Survey of Hamiltonian Cycle Algorithms

This section is a review of Hamiltonian cycle algorithms that have been developed in the literature for arbitrary graphs. The focus of this section is to examine the algorithms to see what techniques and heuristics have been considered and utilized. Since we are interested in algorithms that can be applied to arbitrary graphs, we tend not to discuss techniques and algorithms that are either applicable only to a particular graph model or that are extremely complicated to implement.

Table 2.2 lists the algorithms we examine. For each algorithm we list the name, the authors, the literature reference, the type of algorithm (heuristic or backtrack) and the

Table 2.2: Arbitrary graph Hamiltonian cycle algorithms.

Algorithm	Authors	Type	Graph Type
Pósa	Pósa [24]	heuristic	undirected
UHC	Angluin and Valiant [1]	heuristic	undirected
DHC	Angluin and Valiant [1]	heuristic	directed
HAM	Bollobás, Fenner and Frieze [4]	heuristic	undirected
SparseHam	Frieze [16]	heuristic	undirected
HideHam	Broder, Frieze and Shamir [7]	heuristic	undirected
DB2	Brunacci [8]	heuristic	undirected
DB2A	Brunacci [8]	heuristic	directed
LongPath	Kocay and Li [20]	heuristic	undirected
LinearHam	Thomason [30]	heuristic	undirected
MultiPath	Kocay [19]	backtrack	undirected
595Ham	Martello [22]	backtrack	directed
KTC	Shufelt and Berliner [29]	backtrack	undirected

type of graph it operates on (undirected or directed). Most of the names for the algorithms come from the papers where they are introduced. Where names were not given, or were not unique, we created names related to the techniques used by each particular algorithm. The algorithms are listed in the table in the same order as they are presented in the remainder of this section.

2.5.1 Pósa's Algorithm

Pósa's algorithm is a heuristic algorithm based on the work done by Pósa in [24] (see Section 2.2.3). Pósa's work has served as the basis for much of the later Hamiltonian cycle algorithm development, and thus is presented as an algorithm here. Since Pósa does not provide an explicit algorithm in his paper, some portions of the algorithm are open to various implementations. For the purposes of this section, one particular set of design choices is presented. Note that others before Pósa had similar ideas concerning finding Hamiltonian cycles. In particular, Euler in 1759 [13] came up with a method of constructing knight's tours of chessboards which is very similar to a randomized heuristic algorithm employing techniques such as the rotational transformation. (See [3, pp. 177–179].)

The idea behind Pósa's algorithm is to attempt to extend the length of a partial path until a Hamiltonian cycle is formed. No backtracking is ever done: the length of the partial path is non-decreasing. If the current path endpoint has no non-path neighbours then the rotational transformation is used to obtain a different endpoint, which allows additional expansion of the path. See Section 2.2.3 and Figure 2.1 for a description of the rotational transformation. The algorithm quits when it has tried all the neighbours of the current

```
Select an initial vertex at random and make it the start of a partial path P.
Do {
    Select a random neighbour vertex x of the current endpoint e that has
        not been processed yet. If no such vertex is found, then return failure.
    If x is not in P, then add x to P
    else {
        Apply the rotational transformation to P using the edge to x from e
            to obtain a new path P' and a new endpoint u. If u has not
            been an endpoint for this path length before, then set P to P'.
    }
    If P contains all the vertices in the graph, then check if the endpoints
        of P are connected by an edge. If so, a Hamiltonian cycle exists
        so return success.
}
```

Figure 2.4: Pósa's algorithm.

endpoint as endpoints. Pósa's algorithm is specified in Figure 2.4.

2.5.2 UHC Algorithm

The UHC algorithm of Angluin and Valiant [1] is similar to Pósa's algorithm: the main difference is that as edges are selected to be traversed, they are removed from the graph. Note that there is no explicit termination mechanism in the algorithm to prevent it from endlessly repeating. Instead, by deleting each edge from the graph as it is traversed, the algorithm is prevented from choosing that edge again in the future. Clearly the number of selections the algorithm can make is limited to the number of edges that exist in the graph. While no experimental results are presented in the paper, the authors state in the conclusion that "some preliminary experiments suggest that it might be better not to delete edges as they are explored." [1].

2.5.3 DHC Algorithm

In [1] Angluin and Valiant present another heuristic algorithm, DHC, for solving the directed-graph Hamiltonian cycle problem. The DHC algorithm is similar to the UHC undirected graph algorithm presented in the same paper (see Section 2.5.2). The major difference is in how the path is transformed when it can not be extended from the current endpoint. The rotational transformation cannot be used on directed graphs (since a directed path cannot be traversed in the reverse direction). Instead, the DHC algorithm uses

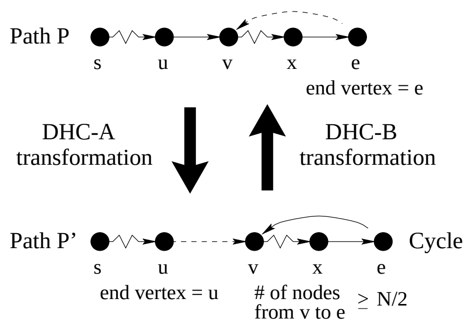

Figure 2.5: The DHC-A and DHC-B transformations.

a more complicated transformation, the directed rotational transformation, which performs the same function as the rotational transformation: if the next vertex to be visited is in the path, then change the path to get a new endpoint. The basic heuristic underlying this idea is to try to maximize the amount of exploration of the search space, rather than to try to exhaustively try every possibility.

The directed rotational transformation used by the DHC algorithm actually consists of two different transformations we will refer to as DHC-A and DHC-B (see Figure 2.5). The DHC-A transformation requires that the following preconditions are satisfied. Starting with a partial path P, the new vertex v to be visited (a neighbour of the current endpoint e) must be in P, and there must be at least $n/2$ nodes in P between v and e (inclusive). If u is the predecessor of v in P, then edge (u, v) is deleted from the path and edge (e, v) is added. The endpoint is changed to u. This produces a partial path (from the starting vertex s to the endpoint u), and a cycle (of at least $n/2$ vertices, with v being one of them).

While the rationale for requiring at least $n/2$ nodes was never made clear in the paper, having this condition prevents the algorithm from forming multiple cycles. A maximum of one cycle can exist along with a partial path.

The second DHC-B transformation is applied when the algorithm has a partial path and a cycle. The new vertex v to be visited (a neighbour of the endpoint u) is a member of the cycle, and we will designate the predecessor of v in the cycle as e. The edge (e, v) is deleted, the edge (u, v) is added and the endpoint is changed to e. This transformation converts the path and cycle to a single path.

Since each transformation transforms the current partial solution into the format required for the other transformation, it is obvious that the algorithm must alternately apply the two transformations. The DHC algorithm avoids alternating between two partial solutions by deleting each edge from the graph as it is traversed.

```
Select an initial vertex s at random and make it the start of a partial
    path P. Set the algorithm mode to A.
Do {
    Select a random neighbour vertex v of the current endpoint e. If no
        such vertex is found, then return failure.
    Delete edge (e, v) from the graph.
    If v is not in P, then add v to P
    else {
        If the algorithm mode = A and v ≠ s and there are at least n/2
        vertices between v and e inclusive along P, then {
            Apply the DHC-A transformation. Set the algorithm mode to B.
                (The partial solution now consists of a partial path P and a
                cycle C with at least n/2 vertices in it.)
        }
        else if the algorithm mode = B and v ∈ C then {
            Apply the DHC-B transformation. Set the algorithm mode to A.
                (The partial solution now consists solely of a partial path P.)
        }
    }
    If P contains all the vertices in the graph, then check if the original
        graph has an edge from the current endpoint of P to s. If so, a
        Hamiltonian cycle exists so return success.
} (end of do loop)
```

Figure 2.6: The DHC algorithm.

The DHC algorithm is specified below in Figure 2.6.

2.5.4 HAM Algorithm

In [4] Bollobás, Fenner and Frieze present an heuristic algorithm HAM. The HAM algorithm uses the rotational transformation just like the previously discussed algorithms, but it also contains some new techniques. The most important of these is cycle extension. As discussed in Section 2.5.1, the goal of all of these heuristic algorithms is to extend a partial path, until a Hamiltonian path is obtained. If there is no edge from the current endpoint to a neighbour not in the path, then previous algorithms use the rotational transformation to help avoid hitting a dead-end, and be forced to quit. The HAM algorithm instead first tries to apply the cycle extension technique, by checking if the partial path forms a cycle (is there an edge between the two endpoints?).

If the cycle exists, then the cycle extension technique can be applied. The basic idea

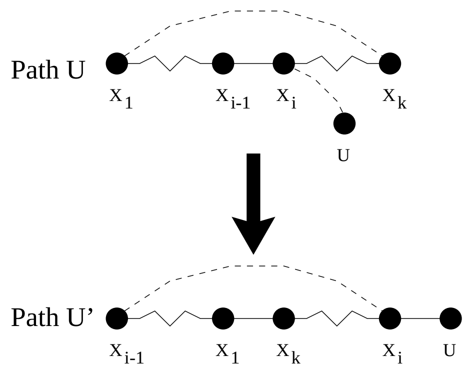

Figure 2.7: The cycle extension technique.

is that any vertex in the cycle can be made an endpoint of a path by deleting the edge between that vertex and one of its neighbours. So the algorithm scans through the vertices of the cycle, looking for a vertex with a neighbour not in the cycle. This vertex is made the endpoint of a new path, and one of its neighbours is made the other endpoint. Assuming the graph is connected, such a vertex is guaranteed to be found. (If the graph is not connected, no Hamiltonian cycle exists.) And with such a vertex as the endpoint, the algorithm can add at least one new vertex to the path. So the conversion of the partial path into a cycle guarantees that the path can be extended. See Figure 2.7.

The other new technique employed by the HAM algorithm is to emphasize extending the path over algorithm performance. The previous algorithms check only one neighbour of the current endpoint: if it is not a new vertex (not on the current path), then the rotational transformation is applied immediately. While this is a quick and easily implemented approach, it ignores the possibility that examination of a different neighbour would allow the path to be extended.

The HAM algorithm instead uses a form of backtrack search to search for an extension to the current path. Basically, the HAM algorithm tries each neighbour of both endpoints of the current path. If any neighbour is an unvisited vertex, or is the other endpoint (implying that the partial path forms a cycle), then the path can be extended, and the algorithm stops the search. If a neighbour vertex does not permit this, then a rotational transformation is applied and the new path is added to a list of paths to check. Once all the neighbours of the initial partial path are checked (with no path extension possible) then the algorithm starts checking the other paths (one path at a time). Note that the search is done breadth-first, rather than depth first. The algorithm is essentially backtracking through the possible sets of rotational transformations that can be applied to the initial partial path. Thus we refer to this technique as backtrack rotational transformation.

This backtrack path extension search is terminated (in failure) by setting an upper limit to the search depth. Note that the search depth is equal to the number of rotational transformations that were applied to the original partial path. However, the algorithm does not restrict instances of partial paths from repeating in the path array.

The HAM algorithm is specified in Figure 2.8. Note that the algorithm is deterministic and does not employ any randomized decision-making as do the algorithms previously discussed. However, a stochastic version could be easily implemented.

2.5.5 SparseHam Algorithm

In [16] Frieze presents SparseHam, a variation of the HAM algorithm given in [4] (see Section 2.5.4). Frieze proves that this algorithm can find *whp* a Hamiltonian cycle in two classes of random graphs with constant average degree. While SparseHam employs the same techniques – cycle extension and backtrack path extension search – as the HAM algorithm, there are three differences in implementation. First, SparseHam employs depth-first search for the backtracking path extension, rather than breadth-first search.

The second difference is in the criteria used to limit which paths are searched. The HAM algorithm had no such criteria, allowing paths to be repeatedly tried. The SparseHam algorithm instead stores the edge from the current endpoint to the selected vertex (the rotation edge) for each rotational transformation. While the path has not been extended (in length), future rotational transformations cannot use that edge (as a rotation edge). This prevents partial paths from being repeated while backtracking.

The third difference between the two algorithms is that SparseHam only considers extending the path from one of its endpoints. (HAM considers both endpoints). If SparseHam cannot find a path extension, it will then search back through its list of paths (obtained during the backtrack search), looking for an extension from the other endpoint.

The SparseHam algorithm is specified in two parts, the main function (Figure 2.9) and the TryExtend() function (Figure 2.10) which does the actual work of extending a path.

2.5.6 HideHam Algorithm

The HideHam algorithm by Broder, Frieze and Shamir [7] was designed to find *whp* Hamiltonian cycles in graphs where a Hamiltonian cycle is hidden within a random graph. More formally, the graph $G = (V, E)$ has an edgeset which is the union of a random graph $G_{n,p}$ with $p = d/n$ and a Hamiltonian cycle.

Start with the first vertex in the graph, and select the first
    neighbouring vertex. Set the current partial path P_c to
    contain these two vertices.
Do {
    Initialize the partial path array, and set the first element
        of this array equal to the current partial path P_c.
    Set OldPathLength $= |P_c|$.
    While $|P_c|$ =OldPathLength {
        $P_c = p_1, \ldots, p_k$. Create a list L of each neighbour of
            vertex p_1 and p_k. Do not include p_2 as a neighbour
            of p_1, nor p_{k-1} as a neighbour of p_k.
        For each $v \in L$ do {
            If $v \notin P_c$ then add v to P_c and break (out of the for loop).
            else if v is the other endpoint of P_c then {
                Form a cycle using P_c and the edge between P_1 and P_k.
                If the cycle is Hamiltonian, then terminate with success.
                else perform cycle extension (as described above) to
                    add a vertex to P_c and break (out of the for loop).
            }
            else apply the rotational transformation to P_c using v to
                get a new path P'. Add P' to the partial path array.
                (P_c is not changed.)
        } (end of for loop)
        If $|P_c|$ =OldPathLength then {
            If the number of partial paths examined is equal to the
                maximum search depth, then return failure.
            else set P_c to point to the next member of the partial
                path array. ($|P_c|$ is still the same as OldPathLength.)
        }
    } (end of while loop)
} (end of do loop)

Figure 2.8: The HAM algorithm.

```
SparseHamAlg() {
    Set the partial path P to contain the first vertex in the graph.
    Do {
        Do {
            Set OldPathLength = |P|.
            Clear path array.
            TryExtend(P)
            If TryExtend() returns success, then return P.
        } while |P| ≠OldPathLength
        For each P in the path array do {
            Reverse path P (to use the other endpoint).
            TryExtend(P)
            If TryExtend() returns success, then return P.
            If |P| ≠OldPathLength then break (out of for loop).
        } (end of for loop)
    } while |P| ≠OldPathLength
    Return failure.
} (end of SparseHamAlg() )
```

Figure 2.9: The main function of the SparseHam algorithm.

The HideHam algorithm consists of three major phases[4] which in general do the following. First, the algorithm builds a set X of the low degree vertices. Then HideHam builds a set of paths $P = \{P_1, \ldots, P_k\}$ which each contain at least one vertex $x \in X$ (where x is not an endpoint of P_i). In the final phase, the algorithm uses previously discussed techniques (rotational transformation, cycle extension) (with some minor modifications) to join the paths together into a single cycle.

The HideHam algorithm contains some artificial elements which were added for the proofs to work, and also seems to be tightly coupled to the specified graph model. Therefore we will only discuss the first two phases in limited detail.

The basic idea underlying the first two phases is that graphs with large minimum degree have Hamiltonian cycles which are easy to find. It is the presence of low degree vertices which makes it harder for cycles to exist and for cycles to be found. Thus, by embedding the low degree vertices into disjoint paths, the graph is (conceptually, at least) reduced to one with large degree vertices only. Each path is formed as follows: X is the set of low degree vertices. The endpoints of each path are not in X, and for any two consecutive vertices on

[4] The algorithm presented in [7] actually has 4 phases, but we omit phase 3 since it exists only because it is needed for the proofs developed in the paper.

```
TryExtend(P) {
    Let active endpoint of P be e and let the other (fixed) endpoint be s.
    If e has a neighbour x not in P then add x to P.
    else if e has an edge to s then {
        If P contains all the vertices of the graph then return success.
        else perform cycle extension to add a vertex to P.
    }
    else {
        Let X = the set of neighbours of e.
        For each x ∈ X do {
            If edge (e, x) has not been used in the current execution
                of TryExtend() then {
                Set OldPath = P.
                Modify P using the rotational transformation and vertex
                    x to get a new endpoint y.
                If y is not an endpoint for any path in the path array then
                    add P to the path array.
                TryExtend(P)
                If TryExtend() returns success, then return success.
                If |P| ≠OldPathLength then set P = OldPath.
            }
        } (end of for loop)
    }
} (end of TryExtend() )
```

Figure 2.10: The TryExtend() function of the SparseHam algorithm.

a path, at least one is in X. Each vertex of X is contained in one of the paths. We refer to this process as the low degree paths technique, which has some limited similarities to the multipath search technique of Kocay's MultiPath algorithm [19] (see Section 2.5.11).

Note that in forming the paths, for each vertex v in a path and not an endpoint, the edges to neighbours of v not on the path are deleted from the graph. This reduces the degree of other vertices, which may not be in X, and thus might produce additional low degree vertices. The HideHam algorithm uses an impractical modification to constructing X before making the paths to avoid this problem. Another alternative which seems more reasonable from an algorithmic standpoint is to add vertices which become low degree to the set X, as paths are created and edges removed.

2.5.7 DB2 Algorithm

The DB2 algorithm described by Brunacci in [8] is presented quite differently from the other algorithms in this survey. The DB2 algorithm considers the Hamiltonian cycle problem as a version of the travelling salesmen problem (TSP). In the TSP, graph edges are given weights, and the goal is to find a Hamiltonian cycle whose summation of edge weights is a minimum. The DB2 algorithm therefore establishes three categories of edges (with increasing weights): forced edges (an incident vertex is of degree two), normal edges, and non-edges (vertex pairs not in the graph). Edge selection is prioritized by the type of edge. So initially when forming a cycle, the DB2 algorithm may use non-edge vertex pairs (with large weights). The goal for DB2 is to find a Hamiltonian cycle that does not use any non-edges.

Another innovation used by Brunacci is to use vertex degree to prioritize vertex and edge selections. We call this the low degree first technique because low degree vertices are selected first by the algorithm. Edges are sorted according to the degree of the incident vertices. An edge (x, y) is specified by $d(x) \leq d(y)$ and is sorted first by $d(x)$, and then by $d(y)$.

The rationale behind this technique is simple. By first using the lower degree vertices, this leaves the higher degree vertices for later, when there are fewer choices, and more likelihood of dead-ends in the search. But since these latter vertices have higher degrees, there are more available options for searching, thus improving the chances of the search succeeding. This is the same reasoning as that used by HideHam in its low degree paths technique.

We now describe the DB2 algorithm. The first step is to sort the vertices by degree,

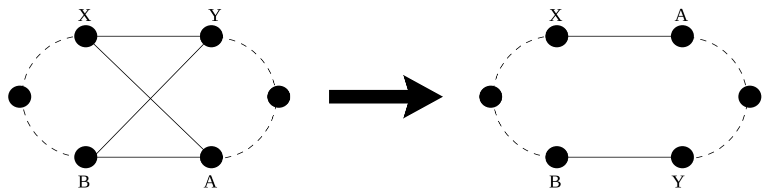

Figure 2.11: The transformation used by the DB2 algorithm.

and then form a priority list of the vertex pairs (edges), organized by category and by vertex degree within each category, as specified above. Next, the algorithm starts to form a cycle C by adding edges one by one starting from the top of the priority list. No vertex degree is allowed to exceed 2; edges which would break this limit are not added. If such a conflicting edge is a forced edge, then no Hamiltonian cycle is possible and the algorithm quits. If after adding all the forced edges a cycle exists, no Hamiltonian cycle is possible either (see Theorem 4) so the algorithm quits. When $n-1$ edges have been added, this stage is finished. The graph C is not yet a cycle, but in general will consist of zero or more cycles and a path P. The next step is to convert C into a cycle by deleting an edge in each cycle, and merging the new path to P, and then joining the endpoints of P when P becomes Hamiltonian. The paper [8] gives no details of this step. The final stage is to eliminate all non-edges in C using the following transformation. For a non-edge (x, y), consider all edges (a, b) where these vertices are in the following order within the cycle: $\ldots, x, y, \ldots, a, b, \ldots$. Sum the weights of the edges (a, x) and (b, y). Select the (a, b) edge which gives the lowest sum, and replace edges (x, y) and (a, b) with (a, x) and (b, y). See Figure 2.11. Note that the transformation is essentially the same as performing a rotational transformation on a path without edge (x, y) to obtain a cycle. Each non-edge is only replaced once. If that edge later must be reintroduced into the cycle (to eliminate another non-edge) then the algorithm returns failure.

2.5.8 DB2A Algorithm

The DB2A algorithm by Brunacci [8] to find Hamiltonian cycles on directed graphs is extremely simple: it simply converts a directed graph to an undirected graph, executes an algorithm (such as DB2) to find an undirected Hamiltonian cycle and then converts the solution back to the directed graph. The conversion from a directed to an undirected graph is a standard technique used, for example, to reduce the directed Hamiltonian cycle problem to the undirected Hamiltonian cycle problem to prove NP-completeness. A directed graph D is converted into an undirected graph G as follows: Each vertex $v \in V(D)$ is mapped

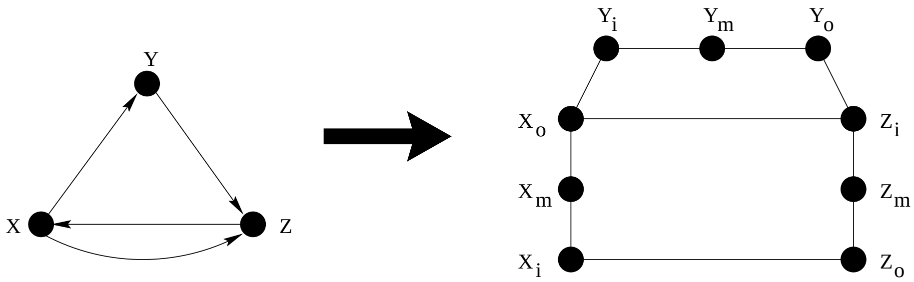

Figure 2.12: Conversion of a directed graph to an undirected graph.

into three vertices v_i, v_m, v_o of $V(G)$ and two undirected edges (v_i, v_m) and (v_m, v_o). Each directed edge (x, y) is translated into an undirected edge (x_o, y_i). See Figure 2.12. It is simple to prove that D has a directed Hamiltonian cycle iff G has a undirected Hamiltonian cycle [2, pp. 331-332].

Note that if $|V(D)| = n, |E(D)| = m$ then $V(G) = 3n$ and $E(G) = 2n + m$. The undirected graph is much larger than the directed graph, and thus this technique seems to be much less efficient than using a directed Hamiltonian cycle algorithm. However, the $2n$ new edges in G are all forced edges and therefore are quickly traversed by Hamiltonian cycle algorithms, as are n of the new vertices. In addition, this conversion might be useful for graphs that are mostly undirected, but have a few directed edges.

2.5.9 LongPath Algorithm

The LongPath algorithm by Kocay and Li [20] was designed to produce long paths rather than specifically Hamiltonian cycles. While the authors make virtually no references to the Hamiltonian cycle problem, this algorithm represents an interesting alternate approach. Virtually all of the Hamiltonian cycle algorithms reviewed in this section have a binary solution set: either they find a Hamiltonian cycle or they report failure. However, for heuristic algorithms, an acceptable approximate solution for certain applications may involve a less than maximum length cycle or path. As well, reporting the maximum path (or cycle) length found provides an additional metric for evaluating an algorithm's performance. The LongPath algorithm is also of interest because of three new techniques that the authors introduce: crossover extension, chain extension and bypass extension. These techniques are discussed in detail below, followed by a description of the actual algorithm.

The crossover extension technique can be considered either an extension or a formalized version of the backtrack rotational transformation technique. The basic idea is to form a cycle so that the cycle extension technique can be used. Starting with a path P (with

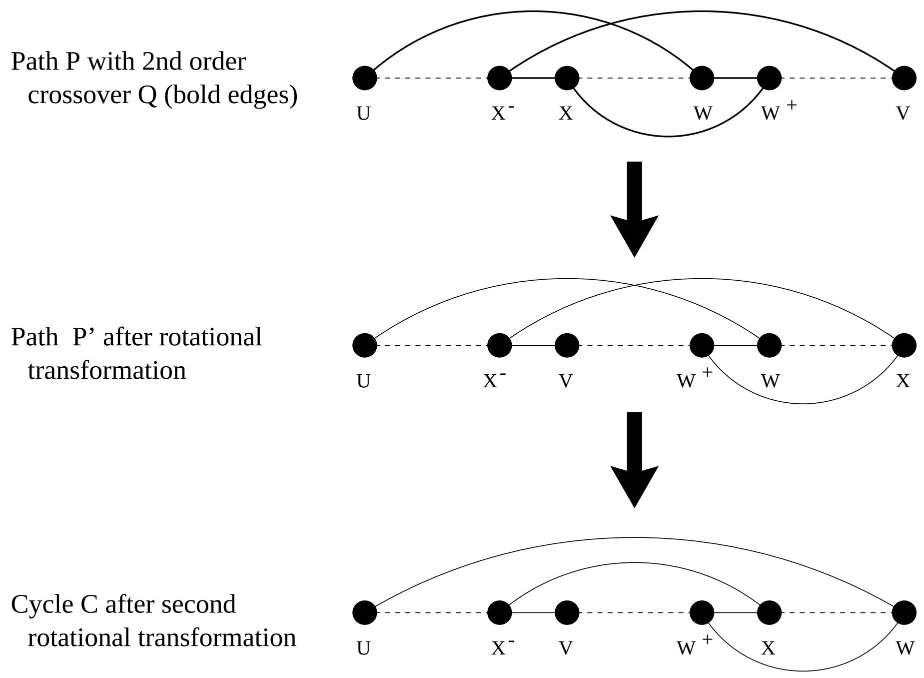

Figure 2.13: Path with order 2 crossover transformed into a cycle using two rotational transformations.

endpoints adjacent only to vertices of P) we want to find a new ordering of the vertices in P (using other edges) so that a cycle can be constructed, and the path then extended (assuming connectivity). The new ordering is obtained using a crossover.

We start with some definitions. A trail in a graph G is a path in which vertices may be repeated, but not edges. A crossover Q for a path $P = \{u, \ldots, v\}$ is a uv-trail (a trail from u to v) such that $V(Q) \subseteq V(P)$ and C is a cycle with $V(C) = V(P)$ and $E(C) = E(P) \oplus E(Q)$. The order of a crossover Q is equal to $|E(P) \cap E(Q)|$. For an arbitrary vertex x in a path P with endpoints u, v, we denote the vertex before x (towards u) by x^- and the vertex after x (towards v) by x^+. A cross-edge is any edge $(x, y) \in E(Q) - E(P)$.

So a crossover of order 0 is the edge (u, v) connecting the endpoints of the path. A crossover of order 1 is the same as a rotational transformation followed by a 0 order crossover. It appears that higher level crossovers can similarly be decomposed into a sequence of rotational transformations applied at one or both ends of the path. The exact sequence of rotational transformations needed is not always easy to determine for a particular crossover on a path. However, the cycle can easily be found by following the path while avoiding path edges that are also in the crossover. Figure 2.13 shows the transformation of a path with a second order crossover into a cycle, using two rotational transformations.

Once a crossover is found on a path, the sequence of rotational transformations corre-

```
FindCrossover(vertex w, integer k) {
    (w is the current endpoint of Q, k is the current search depth / order)
    Set CrossOver = False.
    If k > M then return.
    If v ∈ N(w) then {
        If Q plus edge (w, v) is a crossover then Set CrossOver = True and return.
    }
    For each x ∈ N(w) such that x ∈ P − Q, x ≠ w± and d_Q(x) ≤ 2 { do
        Add edge (w, x) to Q.
        if x ≠ u and (x, x⁻) ∉ E(Q) then {
            Add edge (x, x⁻) to Q.
            FindCrossover(x⁻, k+1).
            If CrossOver = True then return.
            Remove edge (x, x⁻) from Q.
        }
        if x ≠ v and (x, x⁺) ∉ E(Q) then {
            Add edge (x, x⁺) to Q.
            FindCrossover(x⁺, k+1).
            If CrossOver = True then return.
            Remove edge (x, x⁺) from Q.
        }
        Remove edge (w, x) from Q.
    } (end of for loop)
} (end of FindCrossover() )
```

Figure 2.14: The FindCrossover() function of the LongPath algorithm.

sponding to the crossover can be applied, producing a cycle. Use of the cycle extension technique will then allow the path to be extended in length. So the remaining question is how to find crossovers. The basic idea is to have a recursive algorithm that constructs a trail from u to v, and then checks if the trail is a valid crossover. The order of the crossover corresponds to the depth of recursion. Since the number of trails is expected to be exponential in the length of the path, the search is limited to a maximum order M [5]. Kocay and Li prove that this algorithm will find a crossover iff a crossover of order $k \leq M$ exists. Figure 2.14 presents the FindCrossover() algorithm. Some global variables are used. P is a path with endpoints u, v. Q is a u, w trail with vertices in P. M is the maximum order allowed. CrossOver is a boolean indicating if a crossover was found.

[5]Eliminating this restriction could lead to a new type of backtrack algorithm for Hamiltonian cycles. It would be interesting to investigate such an algorithm to see if it is guaranteed to find a cycle, if one exists, on arbitrary graphs.

The most complicated part of the FindCrossover() function is the *if* statement which determines if Q plus edge (w, v) is a crossover. The most straight-forward way of determining this is to check if $C = P \oplus Q$ is a cycle. While the conditions imposed on Q ensure that $\forall x \in P, d_C(x) = 2$, C will usually consist of several cycles rather than just one. Checking if C is a cycle by traversing all the edges would take too long. Thus, Kocay and Li introduce the concept of a segment graph.

A segment of P is defined with respect to Q (a uw trail) as a connected component of $P - E(Q)$. The set of segments of P with respect to Q is $S(Q)$. The segment graph $SG(Q)$ has vertices corresponding to the segments $S(Q)$. Two vertices are joined by an edge if the corresponding segments are joined by a cross-edge connecting an endpoint of one of the segments to an endpoint of the other. Kocay and Li prove that the components of $SG(Q)$ are equivalent to the components of $P \oplus Q$. So by tracking the components of $SG(Q)$, one can easily determine if $P \oplus Q$ is a circuit by checking if there is only one component of $SG(Q)$. There are some extra coding details involved in tracking the number of components of $SG(Q)$ which we do not cover here.

The chain extension and bypass extension techniques come into use when one has constructed a path P which does not include all the vertices in the graph G, and cannot be extended using other techniques. Kocay and Li refer to the vertices of the subgraph $H = G - V(P)$ as inner vertices of G with respect to P. Subgraph H can be decomposed into a set of connected components $H_1, \ldots, H_k$, $k \geq 1$. The bypass extension technique involves finding a path R with endpoints a, b in some component of H. If $\exists x, y \in V(P)$ such that $x \in N(a)$ and $y \in N(b)$ and $|R| + 2 > |P(x, y)|$ then path P can be extended by replacing $P(x, y)$ with the path $\{x, a, \ldots, b, y\}$. (Note that $P(x, y)$ is the portion of the path P from vertex x to vertex y inclusive.) See [20] for implementation details.

The chain extension technique is a combination of bypass extension and crossover extension. As before, we have a path R with endpoints a, b in some component of H and vertices $x, y \in P$ with $x \in N(a)$ and $y \in N(b)$. Kocay and Li define a bypass crossover Q to be an xy trail with $V(Q) \subseteq V(P)$ and $P \oplus R \oplus Q$ is a uv path. The algorithm to find a bypass crossover is very similar to FindCrossover(), with the only important change being the number of components of the segment graph $SG(Q)$ (whose vertices are components of $P \oplus Q$). See [20] for the details. Using the bypass crossover, path R can then be merged with path P, thus extending P. See Figure 2.15 for an example of this.

Successful use of the bypass extension or chain extension techniques will change a portion of the path and thus will potentially introduce a crossover that can be used to further extend

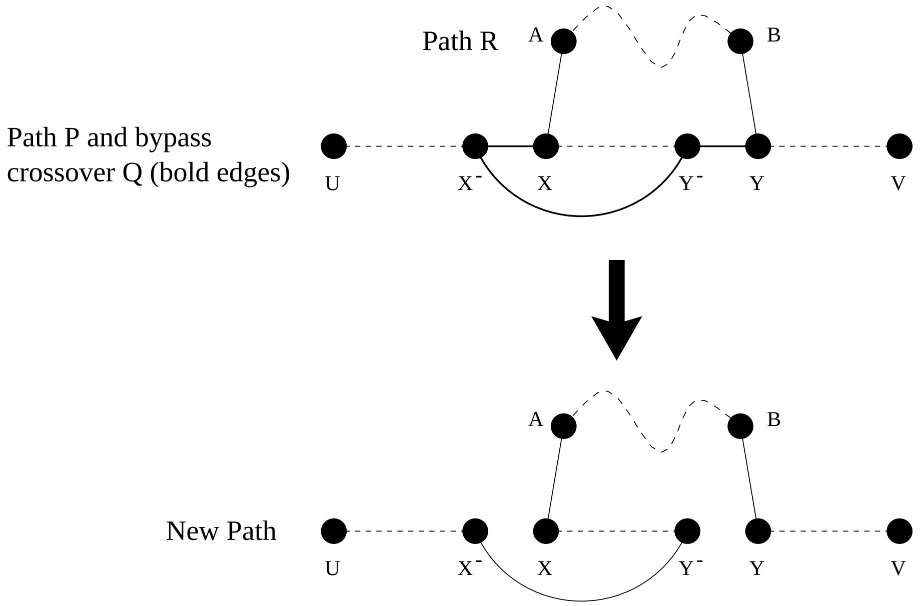

Figure 2.15: An example of the chain extension technique.

the path. Thus, after extending the path with either of these techniques, the crossover extension technique can be retried.

The complete LongPath algorithm is listed below in Figure 2.16. Kocay and Li tested their algorithm on a number of graphs, using crossovers of order ≤ 6 and bypass crossovers of order ≤ 2. They found that the use of the bypass and chain extension techniques made a measurable improvement in performance over just the crossover extension technique. Note that the LongPath algorithm only finds paths: to make it find Hamiltonian cycles, one only needs to perform an additional execution of the FindCrossover() function after a Hamiltonian path has been found.

2.5.10 LinearHam Algorithm

The LinearHam algorithm presented by Thomason in [30] is a linear expected time algorithm for finding a Hamiltonian path between two specified vertices. LinearHam is designed specifically for the random graph model $G_{n,p}$; it uses the edge probability p explicitly in the algorithm. Furthermore, LinearHam consists of three separate algorithms: A1, A2 and A3 that are applied sequentially with increasing time requirements but also increasing probabilities of success. The third algorithm (not specified in the paper since a standard backtrack algorithm could be used) requires exponential time but guarantees to find a Hamiltonian cycle or determine that one does not exist. Since the probability of the first two algorithms failing is exponentially low, the overall expected running time works out to

```
LongPathAlg() {
    Select an initial vertex $x$. Set $u = x$, $v = x$, $P = \{x\}$.
    While $P$ is not a Hamiltonian path do {
        While ($\exists x \in N(u)$ and $x \notin P$ or $\exists x \in N(v)$ and $x \notin P$) Add $x$ to $P$.
        Set OldPathLength = $|P|$. Set $Q = \{u\}$. FindCrossover($u, 0$).
        If a crossover $Q$ was found then {
            Convert $P$ into a cycle $C = P \oplus Q$. Extend $P$ using cycle extension.
            }
        else {
            Construct the components $H_1, \ldots, H_k$ of $H = G - V(P)$.
            For each $x \in P$ do {
                For each $a \in N(x)$, $a \notin P$ do {
                    Find the component $H_i$ such that $a \in H_i$.
                    Construct a linked list $A_i$ of all vertices $y \in P$ for which
                        $\exists b \in H_i$ such that $y \in N(b)$.
                    For each $y \in A_i$ do {
                        Find path $R(a, b)$.
                        If $|R(a, b) + 2 > |P(x, y)$ then reroute $P$ via $R(a, b)$.
                        else {
                            Set $Q = \{x\}$. FindBypassCrossover($x, 0$).
                            If a crossover $Q$ was found then Set $P = P \oplus Q \oplus R(a, b)$.
                        }
                        If $|P| >$ OldPathLength then break out of for loop.
                    } (end of for each $y$)
                    If $|P| >$ OldPathLength then break out of for loop.
                } (end of for each $a$)
                If $|P| >$ OldPathLength then break out of for loop.
            } (end of for each $x$)
        } (end of else)
        If OldPathLength = $|P|$ then $P$ cannot be further extended so return.
    } (end of while)
} (end of LongPathAlg() )
```

Figure 2.16: The LongPath algorithm.

be $O(cn/p)$.

Due to these factors, the LinearHam algorithm does not appear to be easily or usefully applied to arbitrary graphs. One of the techniques used by LinearHam, however, is a simplified version of the chain extension technique, which is utilized in both the A1 and A2 algorithms. In algorithm A1, after finding a path P, LinearHam searches for a vertex $w \in G - V(P)$ which is adjacent to two adjacent vertices x, y in P. (So $x \in N(y)$, $w \in N(x)$ and $w \in N(y)$.) If such a vertex w is found, then P can be extended by inserting w between x and y. In algorithm A2, LinearHam uses a slightly different version of chain extension, in which paths from a vertex $w \in G - V(P)$ to vertices of P are constructed. If the appropriate edge exists between the predecessors of two of these vertices, then the path can be extended to include w and the two paths leading to it. Figure 2.15 illustrates the configuration that LinearHam is looking for, assuming that vertex w lies within path R. LinearHam is only searching for one particular crossover and thus ignores the other possibilities. Thus, while LinearHam's versions of the chain extension technique would be easier to implement than the full version, one would expect LinearHam's versions to be less successful at extending paths, and thus less successful at finding Hamiltonian paths (or Hamiltonian cycles).

2.5.11 MultiPath Algorithm

Kocay's MultiPath algorithm [19] is an extended version of Christofides algorithm [10, Chapter 10]. Both are backtracking algorithms, using exhaustive search to find a Hamiltonian cycle. A key feature of these algorithms is the pruning techniques used to reduce the size of the search space. The MultiPath algorithm also uses an unique technique (which we call the multipath search technique) for performing the search which differentiates it from most of the heuristic algorithms that we have surveyed. We will examine both of these techniques.

The partial solution for most Hamiltonian cycle algorithms is a path that the algorithm tries to make longer. In the MultiPath algorithm, the multipath search technique is used as a different method of organizing the search. Instead of only one path, this technique involves maintaining a set of paths (called segments by Kocay). The first segment is a randomly selected edge. Whenever the algorithm detects a forced edge (an edge that must be in any Hamiltonian cycle) then this edge becomes a new segment. At each stage of the search, the algorithm selects a random endpoint of a random segment from which to extend the partial solution. As segments are added or extended, the algorithm checks whether segments can be merged together. If there are two segments with endpoints (a, b) and (b, x) then the two are merged into one segment (a, x).

There are several different ways in which pruning can be employed in a backtrack algorithm. One can detect edges of the graph which cannot be part of any Hamiltonian cycle and delete them. One can detect edges which must be in any Hamiltonian cycle (forced edges) and include them in the partial solution. One can check if the current graph (with various edges deleted) permits a Hamiltonian cycle to exist. These pruning techniques are all performed relative to the current partial solution. An edge may be unusable and deleted in one part of the search, and be forced in another.

The MultiPath algorithm uses all of these different methods of pruning. The first pruning operation the algorithm performs is based upon Theorem 3. If a vertex x is incident on two forced edges, then all other edges incident on x are deleted. The second pruning operation occurs after an edge is deleted. The algorithm checks if either of the endpoint vertices have a new degree of 2. If this is true, then there is a new forced edge to be added to the set of segments. When merging the segments, if the algorithm detects a forced cycle of edges (a segment with the endpoints the same, after merging) then no Hamiltonian cycle can exist so the algorithm must backtrack (see Theorem 4). Kocay also mentions another possible pruning operation: checking if the graph is connected.

Note that these operations can interact: one deleted edge can create a new forced edge, which can cause other edges to be deleted, and so on. The pruning is done repeatedly until no further changes take place, or until a vertex degree is reduced to 1 (no Hamiltonian cycle possible), or until n forced edges are found (implying that a Hamiltonian cycle exists).

Kocay's extension of Christofides' MultiPath algorithm is based upon the introduction of two additional pruning operations. The first involves looking for small cutsets which when removed produce more components then the size of the cutset (see Theorem 5). If such a cutset is found, then no Hamiltonian cycle exists. A cutset of size one corresponds to an articulation point (or cutpoint), which can be searched for by using a standard depth-first search algorithm. The second operation involves detecting a bipartition (X, Y) where $|X| \neq |Y|$. If such a bipartition is found, then no Hamiltonian cycle exists (see Corollary 5.1). Kocay shows how to detect this condition as part of the depth-first search for a cutpoint.

Figure 2.17 shows the MultiPath algorithm without the cutpoint or bipartition pruning (see [19] for the details of implementing these techniques). The ComputeSegments() function, which handles the merging of segments and the basic pruning, is presented in Figure 2.18.

```
MultiPathAlg() {
    For each vertex x of degree 2, with (a,b) ∈ N(x) do {
        Let P = (a,b) be a segment. Add P to the set of segments S
    }
    Select a random endpoint u of one of the paths P in S.
    MainSearch(G, S, P, u).
    If MainSearch() finds a Hamiltonian cycle then return success.
    else return failure.

} (end of MultiPathAlg() )

MainSearch(graph G, segment set S, segment P, vertex u) {
    For each w ∈ N(u) do {
        Add w to P.
        ComputeSegments(G, S, u).
        If ComputeSegments() finds a Hamiltonian cycle then return.
        If ComputeSegments() detects that no Hamiltonian cycle is possible then {
            Continue (with the next neighbour of u).
        }
        (Cutpoint and bipartition pruning would occur here.)
        MainSearch(G, S, P, u).
        If MainSearch() finds a Hamiltonian cycle then return.
    } (end of for loop)
} (end of MainSearch() )
```

Figure 2.17: The MultiPath algorithm.

```
ComputeSegments(graph G, segment set S, vertex u) {
    Let ProcessQueue = {u} (vertices to process)
    While ProcessQueue is not empty do {
        Let x = next element of ProcessQueue.
        Obtain P for x ∈ segment P.
        If x is in more than one segment then {
            (x must be an endpoint of two segments (a, x) and (x, y).)
            Merge the two segments into segment P.
        }
        Obtain (a, b) as the endpoints of P.
        If a = b then no Hamiltonian cycle is possible. Return.
        If x ≠ {a, b} (not an endpoint) then {
            For w ∈ N(x), w ∉ P do {
                Delete edge (x, w).
                If d(w) = 2 with (j, k) ∈ N(w) then {
                    Add P' = (j, k) to S. Add w to ProcessQueue.
                    If j or k are in S \ P' then add j or k to ProcessQueue.
                }
                If d(w) = 1 then no Hamiltonian cycle is possible. Return.
            } (end of for do loop)
        } (end of if statement)
    } (end of while do loop)
} (end of ComputeSegments() )
```

Figure 2.18: The ComputeSegments function of the MultiPath algorithm.

2.5.12 595Ham Algorithm

The 595Ham Algorithm specified by Martello [22] is an efficient Fortran implementation designed for directed graphs. It is a backtrack algorithm, although the implementation is non-recursive. 595Ham utilizes various pruning operations, many of which are similar to those used by the MultiPath algorithm.

The first pruning operation we will discuss is based upon Theorem 3. For directed graphs, the equivalent theorem is presented below. 595Ham searches for all forced edges (due to indegree or outdegree of 1), and deletes extra edges according to this theorem.

Theorem 10 *In a directed graph D with edge (x, y), if $d^+(x) = 1$ or $d^-(y) = 1$ then all edges $(v, y), v \neq x$ and all edges $(x, z), z \neq y$ are not in any possible Hamiltonian cycle.*

Proof. The edge (x, y) must be in any Hamiltonian cycle since either the outdegree of x or the indegree of y is the minimum possible for a Hamiltonian cycle to exist. Thus, all edges leaving x or arriving at y (except for (x, y)) cannot be in the cycle. □

For each forced edge (x, y), 595Ham extends the edge into a path of forced edges $\{u, \ldots, x, y, \ldots, v\}$ and then deletes the edge (v, u) if it exists. Each forced edge is stored in a list, and is used by the algorithm when extending the path. For detecting if a Hamiltonian cycle is possible on the current graph, 595Ham checks only if any vertex degree (inbound or outbound) has been reduced to 0. For a partial solution $P = \{P_1, \ldots, P_k\}$ which is being extended to vertex x, 595Ham deletes all edges $(P_k, v), v \neq x$, $(v, x), v \neq P_k$ and edge (x, P_1) if $|P| < n - 1$.

The 595Ham algorithm also uses the low degree first heuristic to guide its selection of the next vertex to extend the path. The vertex x with $\min(d^+(x), d^-(x))$ a minimum is selected. Figure 2.19 shows the 595Ham algorithm.

2.5.13 KTC Algorithm

Shufelt and Berliner in [29] developed the KTC algorithm with the goal of developing a backtrack algorithm which never backtracks. They used the knight's tour problem (on various-sized rectangular chessboards) as a generating set of test graphs. Their approach was to devise as many pruning operations as possible, with the end goal to be so effective at pruning that the algorithm never needs to back up. While their approach seems questionable, their pruning operations are applicable to general graphs, at least in theory. In practise, it seems that the graph configurations necessary for the use of these pruning operations will rarely occur and therefore it seems unlikely that using all of these pruning

595HamAlg() {
  Select an initial vertex r with $d^-(r)$ a maximum. Set $P = \{r\}$.
  While $P = \{r, \ldots, e\}$ is not a Hamiltonian cycle do {
    For each vertex v in the graph with $d^+(v) = 1$ or $d^-(v) = 1$ do {
      (v is part of a forced edge we will denote as (x, y).)
      Set $F = \{x, y\}$. (F will be a path of forced edges with endpoints F_1, F_2.)
      While F_1 or F_2 are part of a forced edge not in F, extend F.
      If F is a Hamiltonian cycle then return.
      If F is a Hamiltonian path then return failure.
      Add edge (x, y) to the list of forced edges.
      Remove all edges $(z, y), z \neq x$ and $(x, z), z \neq y$ and edge (F_2, F_1).
      If a vertex indegree or outdegree becomes 0 then return failure.
    }
    While e is the start of a forced edge (e, x) do {
      If $x = r$ and $|P| < n$ then goto Backtrack.
      If $x = r$ and $|P| = n$ then have a Hamiltonian cycle so return.
      Add x to P. x becomes the new endpoint e.
    }
    BackStart: Select a vertex $v \in N^+(e)$ such that edge (e, v) is untried
      according to the low degree first heuristic.
    If no such vertex is found then goto Backtrack.
    Remove all edges $(e, x), x \neq v$ and $(x, v), x \neq e$ and edge (v, r).
    If $d^+(x) = 0$ or $d^-(x) = 0$ for any vertex incident on a removed edge then
      goto Backtrack.
    Continue (with main while statement).
    Backtrack: While previous vertex selection was due to a forced edge {
      Undo all changes of previous vertex selection.
    }
    Undo all changes of previous vertex selection.
    If try to undo selection of root vertex r then
      no Hamiltonian cycle is possible so return failure.
    else goto BackStart.
  } (end of 595HamAlg())

Figure 2.19: The 595Ham algorithm.

operations in a Hamiltonian cycle algorithm will ever be efficient time-wise. Shufelt and Berliner discovered that only the first five or so of their pruning operations resulted in a reduction in the time required on knight's tour graphs. The other pruning operations either increased or did not change the amount of time required on knight's tour graphs. Thus, the usefulness of the pruning operations for the most part seems dubious.

Shufelt and Berliner developed 26 rules for pruning the search. We will mention only the first four rules which were shown to reduce the time required to find Hamiltonian cycles. See [29] for details on the other rules. Note the similarities between these pruning operations and those used by the MultiPath algorithm and the 595Ham algorithm. The first rule prohibits a move if it leads to a degree 1 vertex. Rule two is based upon Theorem 3: a move to a new square v is prohibited if it would result in v having at least two neighbours of degree 2. The third rule is called backplanning. From the initial vertex starting the partial solution, the KTC algorithm searches for any forced edges and follows them, building what Shufelt and Berliner describe as an endpath. In essence, the algorithm expands only one end of the path, but adds forced edges to the other end as they appear. Rule three prohibits moves to vertices in the endpath. While a redundant rule, we mention it because the inclusion of backplanning in the algorithm seemed to improve its performance. Rule four is nearly identical to rule two. If a vertex has two degree 2 neighbours, then all the additional edges incident on this vertex can be deleted.

Chapter 3

The Design of Hamiltonian Cycle Algorithms

3.1 Introduction

Most of the work on Hamiltonian cycle algorithms (see Chapter 2) has focused on either using them for proofs, or for proving probabilistic bounds on their performance (given a certain random graph model). There are only a few references that focus on designing efficient Hamiltonian cycle algorithms. Kocay's MultiPath algorithm [19] and Kocay's and Li's LongPath algorithm [20] contain many innovative algorithmic techniques. However, there is limited theoretical and experimental validation that these techniques make their algorithms more efficient. The DB2 and DB2A algorithms of Brunacci [8] and the 595Ham algorithm of Martello [22] are presented as being efficient algorithms, but no evidence of this assertion is provided. Shufelt and Berliner in [29] did investigate the effectiveness of various pruning operations as part of their KTC algorithm. Unfortunately, they only use graphs based on instances of the rectangular knight's tour problem. And their goal - a backtracking algorithm which never backtracks - conflicts with our idea of an efficient algorithm, as we discuss below.

Clearly the literature on the Hamiltonian cycle problem has somewhat neglected the area of algorithm design. This may be due to a perception that the Hamiltonian cycle problem is easy compared to other $NP-$hard problems. In any case, we feel that a clear discussion and investigation of the design of Hamiltonian cycle algorithms is not only useful, but needed. For example, recent papers [9, 14] investigating phase transitions for the Hamiltonian cycle problem only reported results on graphs of up to 24 vertices, most likely because they used primitive backtrack algorithms. Our goal is that this chapter may serve as a reference for those who need good algorithms to find Hamiltonian cycles (or prove non-Hamiltonicity) in practice.

We start by discussing various general aspects of algorithm design in Section 3.2. We explain how we will judge the performance of algorithms. In Section 3.3 we investigate the design of a backtrack Hamiltonian cycle algorithm and in Section 3.4 we explore heuristic Hamiltonian cycle algorithms. We present our concluding remarks in Section 3.5.

3.2 Methodology for Algorithm Design

Before we can start designing a specific Hamiltonian cycle algorithm, we need to discuss the methodology we plan to use. Our goal is to obtain efficient algorithms. What do we mean by that? From an algorithmic complexity point of view, an algorithm's efficiency depends upon the amount of time (or storage) required for it to execute. However, we prefer more of an operational point of view. A more efficient algorithm is one that is more capable of solving the problem at hand. For the Hamiltonian cycle problem, this can mean several things. Let us consider algorithm A and an algorithm $A+$ which is always more efficient (this might not be realistic). For a single graph, $A+$ would find a Hamiltonian cycle in less time than A algorithm. For a set of graphs, $A+$ would find Hamiltonian cycles for a higher percentage of the graphs than the A algorithm. For a graph with no Hamiltonian cycle, the $A+$ algorithm might either report that no Hamiltonian cycle exists in less time than A (if both are backtrack algorithms), or might find a longer path than A in the same amount of time (if both are heuristic algorithms).

So our definition of an efficient Hamiltonian cycle algorithm is an algorithm which minimizes the time required to find a Hamiltonian cycle and which maximizes the percentage of Hamiltonian cycles found over arbitrary sets of graphs. For graphs with no Hamiltonian cycle an efficient backtrack algorithm minimizes the time spent proving no Hamiltonian cycle exists and an efficient heuristic algorithm maximizes the length of the longest path found (in a certain period of time). This definition is the basis for how we will compare the performance of different algorithms.

Our definition of efficiency ignores storage requirements for the simple reason that with modern technology storage is never a problem for the Hamiltonian cycle problem. Though the search space is exponential in the size of the problem, the storage space is only polynomial. In fact, the most space-inefficient data structures of the Hamiltonian cycle algorithms we examined do not require more than $O(n^2)$ space.

When defining efficiency, we only consider searching for a single Hamiltonian cycle on a particular graph. Some researchers have considered the problem of finding all possible

Hamiltonian cycles (usually in relation to the knight's tour problem; see [29] for example), but we do not consider this extension of the basic Hamiltonian cycle problem.

Our definition of efficiency uses the phrase "arbitrary sets of graphs". We mean two different things with this phrase. Firstly, we want our algorithms to work on all possible graphs. Secondly, we are acknowledging that testing algorithms with respect to a benchmark set of problems is frequently unproductive (see [18] for a discussion of this). The problem with benchmarks is that they are always open to the criticism of being unrepresentative. Even a general random class of graphs such as $G_{n,p}$, which is able to generate all possible graphs, is vulnerable to this accusation, since the probability distribution is such that Hamiltonian cycles are easily found in a majority of these graphs. And even if a benchmark of hard problems is available, the difficulty of a particular problem can vary according to the algorithm used, as we discuss in Chapter 5.

However, this does not mean that the efficiency of different techniques cannot be evaluated. In this chapter we present arguments that give reasons for the usefulness (or not) of a particular technique, or that show why one technique often will outperform another technique. We also consider how the efficiency of the various techniques is affected by the characteristics of the graphs being solved. For instance, if we were evaluating the addition of bipartite checking to a backtrack algorithm on graphs based on generalized knight's tours, we would consider the fact that all of these graphs are bipartite (with equally sized sets). We could hypothesize that bipartite checking would never prune the search, and thus increase the time required in all instances. Thus, we would conclude that bipartite checking would not increase efficiency on these types of graphs.

Note that in this chapter, we discuss design with respect to undirected graphs only.

3.3 Design of a Backtrack Hamiltonian Cycle Algorithm

In this section we discuss the design of backtrack Hamiltonian cycle algorithms. There are three major components of a backtrack algorithm that we will examine:

- Pruning: What kind of initial pruning (and graph analysis) should be done? What kind of pruning and analysis should be done during the search?

- Search Method: Should a multipath approach be used, or is a singlepath approach just as effective?

- Vertex Selection: How should vertex selection, both initially and during the search, be handled? Is the low degree first heuristic effective?

In the following sections we will examine these questions.

In Section 3.3.4 we present a new technique, graph collapsing, which tries to reduce the search space by more quickly proving that a graph is not Hamiltonian.

3.3.1 Pruning in Backtrack Algorithms

In this section we examine the issue of pruning in backtrack Hamiltonian cycle algorithms. We have already discussed pruning in our review of the MultiPath algorithm (see 2.5.11), but will repeat that information here.

We define pruning to be an operation which reduces the size of the search space. Note that we do not include heuristic techniques (such as vertex selection ordering) which can reduce the search space for some techniques and increase it for others. By search space, we mean the number of vertices (or edges) explored while looking for a Hamiltonian cycle.

Pruning can be done at two different stages in the algorithm. First, *initial pruning* is completed before the algorithm's search has started. Second, *search pruning* is performed at each stage of the search.

The basic design issue we consider here is for each pruning operation; does its inclusion in a backtrack Hamiltonian cycle algorithm improve its efficiency? In this context, according to our definitions in Section 3.2, an efficient pruning operation is one that reduces the time required to find a Hamiltonian cycle. The more time required to execute a pruning operation, the greater the accompanying reduction in the search space must be for the pruning operation to be worth doing. When we refer to a successful pruning operation we refer to one that obtains a reduction in the search space. Note that successful pruning is not necessarily efficient pruning. Initial pruning for this reason can be potentially more complex than search pruning, since initial pruning is only done once, and will tend to involve limited resources with respect to the rest of the search. This point has not received much attention in the backtrack algorithms we surveyed; they use the same pruning both initially and during the search. For this reason, we look at initial pruning separately from search pruning.

There are several different ways in which pruning can be employed in a backtrack Hamiltonian cycle algorithm. One can detect edges of the graph which cannot be part of any Hamiltonian cycle and delete them. One can detect edges which must be in any Hamiltonian cycle (forced edges) and include them in the partial solution. One can check if the graph permits a Hamiltonian cycle to exist. The pruning operations that we surveyed in Chapter 2 can be divided into two general categories. *Global checking* evaluates the entire graph to see if it possesses some property that makes the existence of a Hamiltonian cycle impossible. The global checking operations we have seen are:

- If the graph is not connected, no circuit is possible.
- If the graph has a cutpoint, no circuit is possible.
- If the graph is bipartite with unequally sized partition sets, no circuit is possible.

Graph reduction searches only part of the graph (usually only a few vertices or edges) to determine if an edge can be deleted or is forced. The list of graph reduction operations is as follows.

- Delete other edges of a vertex incident to two forced edges.
- Make an edge forced that is incident on a degree 2 vertex.
- Delete the edge connecting the endpoints of a forced path. (This includes the partial solution, if it is not Hamiltonian.)

There is one remaining pruning operation we have not listed above. If a degree 1 vertex exists in the graph, no circuit is possible. We assume that all Hamiltonian cycle algorithms make use of this, and thus do not include it in the list of pruning operations.

Note that since graph reduction can involve modifications to a graph (by deleting certain edges), global checking should not be performed until all the graph reduction operations are finished. We will examine the efficiency of these operations both during initial pruning and search pruning.

Additional graph reduction and global checking operations exist that we do not discuss here. Shufelt and Berliner [29] developed many graph reduction operations for their KTC algorithm, some of which were quite complex. Kocay [19] discussed searching for small cutsets which prove that no Hamiltonian cycle is possible (using Theorem 5). This is an extension of global checking using cutpoints. One possible (but untested) source of techniques for global checking is to use graph classes such as interval graphs [1]. If a graph class has a low-order polynomial detection algorithm and a low-order polynomial Hamiltonian cycle algorithm then we could try to detect if the graph is a member of that graph class, and if so find the Hamiltonian cycle. Alternatively, we could try to show that the graph is a member of a graph class that has no Hamiltonian cycle.

[1] We credit Joe Culberson for this idea.

Initial Pruning

As we mentioned above, initial pruning is only performed once, at the start of a search, and thus can be more complex while remaining as efficient as search pruning. There is another use for initial pruning. During the search, it may be convenient to the algorithm for the graph to posses certain properties such as biconnectivity. Verifying that the graph possesses such properties in the initial pruning may increase the overall efficiency of such an algorithm, without a measurable increase in efficiency from just the initial pruning.

The success of the various pruning operations depends upon the existence of graph structures that the operations are searching for. For example, most of the graph reduction operations require the existence of degree 2 vertices. All generalized knight's circuit graphs have such vertices (in the corners, and perhaps along the edges) and thus would be good candidates. However, graph reduction fails on graphs with a minimum degree of 3. Similarly, using global checking to find a cutpoint is only useful if some of the graphs are not biconnected. For reasonable parameter values, all generalized knight's circuit graphs are initially biconnected, and after graph reduction, which will occur mostly around the edges of the graph, we expect them to remain biconnected. So testing for biconnectivity would not seem useful on generalized knight's circuit graphs.

To contrast, in [14] Frank and Martel found that for small instances of $G_{n,m}$ of particular edge densities, there are very few biconnected graphs that are not Hamiltonian. Thus, testing for biconnectivity is a good way of eliminating most of the non-Hamiltonian graphs. We would thus expect biconnectivity pruning to be efficient in this case, since the time required to find a cutpoint should be much less than the time required to backtrack through the entire search tree and demonstrate that no Hamiltonian cycle exists.

From this discussion, we see that the use of the various pruning operations will depend upon our knowledge of the properties of the graph set we are using.

Search Pruning

At first glance, it appears that efficient search pruning must always be a subset of initial pruning. This is not necessarily true. For example, graph reduction requires degree 2 vertices, which may not exist in the initial graph. But as vertices are traversed during the search and extra edges are deleted, degree 2 vertices can appear, justifying the use of graph reduction.

The success and efficiency of initial pruning is mostly determined by the structure and properties of the original graph. Search pruning is different, however. Its success depends upon the structure and properties of the current graph being searched, and this graph is modified at each stage of the search. While the original structure of the graph still affects search pruning, it is not as important as it was for initial pruning. For example, it is easy to imagine a backtrack search of a generalized knight's circuit graph that starts at one end, goes halfway down the board, and then comes back, leaving a cutpoint to the other half of the board. In this case, biconnectivity checking would detect this situation and force the algorithm to backtrack, rather than evaluate every possible path in the first half of the board.

Despite this, we can still generate good hypotheses about the performance of different pruning operations on certain types of graphs. On graphs of mostly low degree vertices (degrees ≤ 4), we would expect graph reduction to work well because the deletion of edges quickly leads to the creation of new degree 2 vertices, which produces new forced edges and perhaps more edge deletions. In Section 5.7.1 we discuss a constructed graph which is designed so that the algorithm tends to follow a certain path, leaving behind an unvisited component. If component checking was performed during the search, the algorithm would not follow these 'false' paths and the graphs would become much easier to solve.

3.3.2 The Search Method

In this section we examine how the search should be performed. The typical approach is to maintain a single path as the partial solution, and extend the path from one endpoint. We call this the *singlepath* method. The *multipath* method used by Kocay [19] and Christofides [10] maintains a list of paths (including all forced edges), and expands a random endpoint of a random path at each stage of the search. We also propose a compromise between these two methods, the *doublepath* method, which uses a single path, but extends the path from either endpoint.

Let us assume that the backtrack operation does search pruning using graph reduction. In particular, let the algorithm delete any edge connecting the endpoints of a forced path. This will include the endpoints of the current partial solution (or solutions, for the multipath method). From an examination of its operation, we have no reason to expect a difference in performance between the singlepath and doublepath methods. Any path the algorithm forms must eventually be closed into a cycle, regardless of which side of the path was extended (or both). The only difference in performance of choosing one endpoint over the other (or extending both) is in the pruning that occurs as the paths are extended. Without knowledge of the entire graph structure, there seems to be no way of correctly deciding every time which to use.

An examination of the multipath method is even more difficult. There does not seem to be any clear benefit of the multipath method over the singlepath and doublepath methods and the multipath method requires a more complex implementation. We do foresee one possible (but small) benefit. In Section 5.7.1 we discuss a specially constructed hard graph which is designed so that the algorithm tends to follow a certain path, leaving behind an unvisited component. If there is a forced edge in this component, then the multipath algorithm would extend a path within this component, and might sooner reach a dead-end than before. However, the rate of expansion of any path is inversely proportional to the total number of paths. Thus, it is uncertain whether the multipath method would actually perform better in even this contrived case.

3.3.3 Vertex Selection

In this section we examine the process of vertex selection: how the backtrack algorithm selects the next vertex to visit. As we discuss below, the search method that is used (singlepath, doublepath or multipath) affects how we evaluate the vertex selection process.

For the doublepath and multipath search methods, one must select the vertex to use as an endpoint to extend the path from. In the previous section, we assumed this was done randomly. If we instead select the endpoint with the largest degree, then we will maximize the number of edges pruned. Alternatively, we can use the low degree first heuristic and select the endpoint with the neighbour of smallest degree (this would take more work to calculate for the multipath method). Note that these two approaches would not work overly well together: if we always extend the path to the smallest degree vertices we can find, then the endpoints will tend to be of small degree, and thus we will be wasting our time looking for a large degree vertex. For the remainder of this discussion, we assume that the endpoint to extend the path from has been selected.

Clearly, if the endpoint is incident on a forced edge, that edge must be followed. If there is no forced edge, then the standard approach is to randomly try the neighbours of the endpoint. The other approach is to use the low degree first heuristic, which we saw used by Brunacci [8] and Martello [22], and has also been called Warnsdorff's Rule with respect to solving knight's tours [3, pg 181]. The reasoning behind why this heuristic could improve an algorithm's performance is as follows. By first using the lower degree vertices, this leaves the higher degree vertices for later, when there are fewer choices, and more likelihood of the algorithm needing to backtrack. But since these latter vertices have larger degrees, there are more available options for forming a Hamiltonian cycle, thus improving the chances of the search succeeding. An alternate explanation comes from taking the constraint satisfaction viewpoint. The lower degree vertices are more highly constrained, and thus should be satisfied (fit into the solution) first [25, pg 91].

We could also consider a high degree first heuristic, which selects the highest degree vertex at each step of the search (this heuristic was used in [9]). The justification for this heuristic is that it maximizes the amount of pruning being done. First, as we leave each vertex (which will be the largest degree possible), we will delete all edges except the one we are following. Second, since we are using up the larger degree vertices first, the remaining vertices will tend to be of lower degree. The edges we delete will be incident on these lower degree vertices, which means further reductions in vertex degrees, possibly leading to the creation of forced edges and further pruning. This pruning means that the algorithm will sooner backtrack out of dead-ends then before. This would seem to improve the algorithm's efficiency on graphs with no Hamiltonian cycle. However, on graphs with a Hamiltonian cycle this may mean that the algorithm has a harder time actually constructing a circuit, because of the increased backtracking, even if the backtracks happen sooner.

From this informal analysis, we might expect the low degree first heuristic to outperform the high degree first heuristic on graphs with Hamiltonian cycles, and the reverse performance on graphs with no Hamiltonian cycle.

Certain backtrack algorithms must also select a vertex to start the search from. The choice of an initial vertex can have a major effect on the performance of a backtrack algorithm. One simple method is to randomly select an initial vertex. One heuristic is to select the largest degree vertex to start from, with the justification being that this provides the most edges to use to complete the cycle. While this appears to be a trivial issue, experiments we have performed (see Section 4.7) indicate that on certain graphs, choosing different starting vertices results in orders of magnitude difference in the time required to find a Hamiltonian cycle.

3.3.4 The Graph Collapse Technique

A major benefit to using a backtrack Hamiltonian cycle algorithm over a heuristic algorithm is that the backtrack algorithm is able to prove that a given graph has no Hamiltonian cycle. This may be important for certain graph sets, such as our generalized knight's circuit problem, in which we want to determine which instances have a circuit. Unfortunately, for a backtrack algorithm to prove that no Hamiltonian cycle exists, it must examine the entire search space. The pruning operations discussed above are able to reduce this search space, but the time required quickly becomes intractable as the graphs become larger.

We propose the graph collapse technique to help further reduce the search space for large, low degree graphs such as generalized knight's circuit graphs. This technique was used by Schwenk [27] to prove (on paper) that a chessboard of size 3×8 has no knight's circuit. The basic idea is to transform the graph G into a collapsed (smaller) graph G_c on which it should be easier to show that no cycle is possible.

We form the collapsed graph $G_c = (V_c, E_c)$ from graph $G = (V, E)$ as follows. Let G have a set of k distinct forced paths $P_f = \{P_1, \ldots, P_k\}$. We replace each forced path P_i in G with a vertex x_i in G_c. (Note that $|V_c| > 2$ must hold.) For each vertex y incident on one or both of the endpoints of P_i, the edge (x_i, y) is added to G_c. (This includes endpoints of other forced paths, so edges (x_i, x_j) could be added to G_c. Duplicate edges are avoided.) Formally, $V_c = V + \{x_1, \ldots, x_i\} \setminus V(P_f)$. $E_c = E + \{(x_i, y) \mid \exists z, z \in V_e(P_i), (y, z) \in E\} + \{(x_i, x_j) \mid \exists y \in V_e(P_i), z \in V_e(P_j), (y, z) \in E\} \setminus \{(i, j) \mid i \in V(P_f), (i, j) \in E\}$ where $V(P_f)$ is the set of vertices comprising the forced paths and $V_e(P_i)$ is the set of endpoint vertices of forced path P_i (so $|V_e(P_i)| = 2$).

We prove the following general theorem.

Theorem 11 *If G_c is a collapsed graph of G and G_c has no Hamiltonian cycle then G has no Hamiltonian cycle.*

Proof. We prove the contrapositive. Assume G has a Hamiltonian cycle. Any Hamiltonian cycle must include the forced paths P_f. Any other edges forming the cycle are in G_c, with the exception of edges that become duplicates in G_c, since all but one edge of a duplicate set is deleted from G_c. We prove that of any set of duplicate edges in G_c, no more than one of the corresponding edges in G can be in any possible Hamiltonian cycle. There are two kinds of duplicate edge sets. For an arbitrary forced path $P_i = \{a, \ldots, b\}$ and an arbitrary vertex z all in G, the edges (a, z) and (b, z) would form a duplicate set in G_c. Assume edges (a, z) and (b, z) are both in a Hamiltonian cycle. Forced path P_i plus these edges and vertex z forms a cycle. But since $|V_c| > 2$, $|P_i| < n - 1$ (since z and at least one other vertex or forced path must be in G). Thus the cycle is non-Hamiltonian, and we have a contradiction. The second kind of duplicate edge set occurs when we have two arbitrary forced paths $P_i = \{a, \ldots, b\}$ and $P_j = \{c, \ldots, d\}$ with edges (a, c) and (b, d) forming a duplicate set in G_c. Assume both these edges are in a Hamiltonian cycle. This means both forced paths form a cycle. But since $|V_c| > 2$, there must be an additional vertex or forced path not included in the cycle, so the cycle is non-Hamiltonian, and we have a contradiction. Thus, no more than one edge of a duplicate set is in any Hamiltonian cycle, and therefore all the (non-forced) edges of the Hamiltonian cycle in G are also in G_c. Thus for any Hamiltonian cycle in G an equivalent Hamiltonian cycle can be constructed in G_c. □

There are several general observations we can make about the graph collapse technique. First, the process of converting a graph to a collapsed graph seems identical to the process of converting a directed graph to an undirected graph by making all the edges non-directional. An informal proof of this is as follows. Using the conversion process described in Section 2.5.8, a directed graph D has each of its vertices replaced with three vertices comprising a forced path in the new graph G. If the new graph G is collapsed using a set of forced paths corresponding to the sets of three vertices used in the conversion process, then the graph D' will be obtained, identical in structure to graph D, but undirected.

A second observation involves the reasoning behind why the technique could improve a backtrack algorithm's efficiency. The idea is that the construction of a collapsed graph will offer the opportunity for additional pruning, which could in turn lead to the construction of a new collapsed graph. As the graph quickly gets reduced in size, we should quickly reach a point in which the Hamiltonicity of the graph can be determined. Note that to use the technique requires a graph with forced paths (degree 2 vertices). The technique would seem to work the best on graphs with forced paths scattered evenly throughout the graph.

If the original graph is Hamiltonian, then the algorithm will eventually find a Hamiltonian cycle on the collapsed graph (due to Theorem 11). This may or may not be useful to the algorithm. Non-Hamiltonian graphs with Hamiltonian collapsed graphs do exist. Figure 3.1 contains one such example. Note that edges (A_1, A_3) and (B_1, B_3) can be removed (they connect endpoints of a forced path), leading to vertices A_1 and B_3 being incident on 2 vertices of degree 2. The edges from both these vertices to vertices C_1 and C_3 can be deleted, which causes degree 1 vertices to be produced. Therefore graph G is non-Hamiltonian. The collapsed graph G_c is obtained by replacing each set of vertices X_1, X_2, X_3 with a corresponding vertex X, and is Hamiltonian.

In practise, it seems difficult to construct a non-Hamiltonian graph that has undergone graph reduction and for which global checking fails, which has a corresponding collapsed graph that is Hamiltonian. [2] Thus, the graph collapse technique could take the Hamiltonian

[2] We conjecture that for a graph G that has undergone graph reduction and for which global checking fails, with corresponding collapsed graph G_c, G is Hamiltonian iff G_c is Hamiltonian. A proof or counterexample of this conjecture would be useful. Note that in Figure 3.1, graph reduction can still be applied to graph

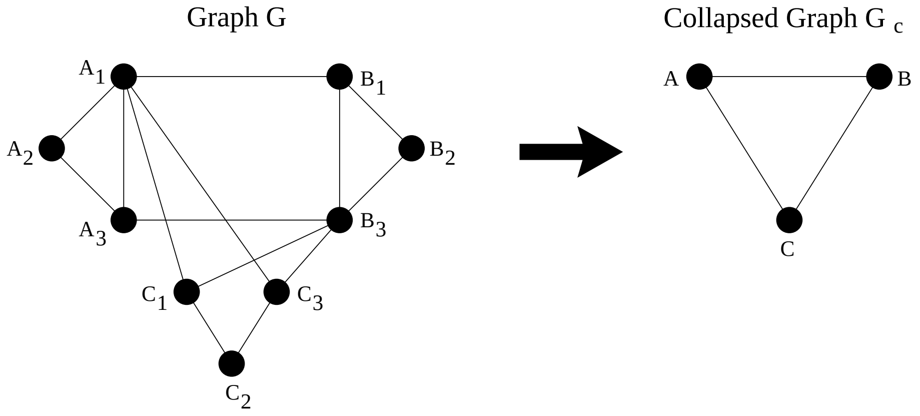

Figure 3.1: Example of a non-Hamiltonian graph with a corresponding Hamiltonian collapsed graph.

cycle obtained on the collapsed graph and try to form a Hamiltonian cycle on the original graph, with some probability of success. However, if the original graph is not found to be Hamiltonian, then the algorithm must abandon the collapsed graph (and the search performed on it) and search the original graph.

Note that the graph collapse technique is untested, and we suspect it will seldom (if ever) improve a backtrack algorithm's efficiency. We feel it is worthy of mention because of the additional insights it offers concerning how Hamiltonian graphs differ from non-Hamiltonian graphs.

The graph collapse technique can be implemented as follows. At each stage of the backtrack search, after pruning, check the number of forced paths. If a certain minimum number of forced paths exist, then construct the collapsed graph and perform pruning. This step can be repeated as many times as possible, until either no Hamiltonian cycle is found to exist, and the algorithm returns to the original graph and backtracks, or until a Hamiltonian cycle is found, in which case the algorithm must backtrack to the previous collapsed graph (or back to the original graph). The algorithm can then try to use the cycle in the collapsed graph to form a cycle in the current graph (collapsed or original). If a cycle cannot be formed, we must continue with the search using the current graph. If a collapsed graph is formed with few or no forced paths, then we continue with the backtrack search. Note that after a few iterations of the search, we may have enough forced edges to collapse the graph again.

G. After graph reduction, G is obviously non-Hamiltonian. If the graph collapse technique is applied, the resulting graph will be non-Hamiltonian as well. Thus graph G is not a counterexample.

3.4 Design of a Heuristic Hamiltonian Cycle Algorithm

In this section we discuss the design of heuristic Hamiltonian cycle algorithms. Most of the heuristic algorithms we surveyed in Chapter 2 followed the same basic concept. They start building a path, keep trying to extend it until they get a Hamiltonian path, and then form a Hamiltonian cycle. No backtracking is ever done: the length of the partial path is non-decreasing. The algorithm quits when it cannot find a way to extend the length of the path. Thus, the major limitation of these algorithms is the techniques they use in extending the path. Poor techniques lead to an inability to extend the path, which causes the algorithm to fail.

Therefore, when considering heuristic design issues, the main focus should be on whether techniques improve the ability of the algorithm in extending the path and finding cycles. While our end goal is still better efficiency, the structure of heuristic algorithms makes the time required by a particular technique less important than how successful it is. Note how this is different from the backtrack algorithm. The reason for this difference is due to the different structure of a heuristic Hamiltonian cycle algorithm. Heuristic techniques for extending the path are usually executed a constant number of times per vertex, and a maximum of n vertices are explored. Backtrack techniques may end up being performed an exponential number of times (due to backtracking).

Most of the search pruning performed by backtrack Hamiltonian cycle algorithms will not work with heuristic algorithms. The various heuristic algorithms avoid dead-ends in the search by modifying the existing path to obtain a new endpoint or a cycle. This requires that the non-path edges incident to vertices in the path are not deleted (as they are in backtrack algorithms). Thus, since edges are not being deleted during the search, there is no point to performing search pruning, since the graph is not changing during the search. Initial pruning, however, may still be of use.

We examine the following issues of heuristic Hamiltonian cycle algorithms in the following sections.

- Algorithm Techniques: Which heuristic algorithm techniques do we have to choose from? Which techniques must we choose between?

- Search Termination: How does the algorithm determine when to quit searching?

- Initial Pruning: Can the initial pruning done by backtrack algorithms be useful for heuristic algorithms?

- Search Method: Should the path be expanded from one end or both ends: a singlepath versus a doublepath approach?
- Vertex Selection: How should vertex selection, both initially and during the search, be handled? Is the low degree first heuristic effective? We introduce a new heuristic technique – the non-path neighbours technique – which provides us with additional options for ordering the search.

3.4.1 Heuristic Algorithm Techniques

In this section we review the various heuristic algorithm techniques that have been utilized in the literature. We discuss how the different techniques can be combined, or if they are competing techniques that we must choose between.

To aid our discussion of the various techniques, we divide the actions of a heuristic algorithm into three stages. At each stage, the algorithm has a path P which it is trying to extend into a Hamiltonian cycle.

1. The endpoints of the path are adjacent to vertices not in the path. The path can be extended immediately (possibly with the use of heuristic techniques to choose between different non-path vertices).
2. The endpoints of the path are only adjacent to other vertices in the path. The rotational transformation, cycle extension, backtrack rotational transformation and crossover extension techniques can all be used to extend the path.
3. The techniques of the previous stage have failed in their search (reached a dead-end). The bypass extension and chain extension techniques can be used to extend the path.

The second stage of search is where we have the most choices in terms of heuristic techniques. The two basic choices are either to perform one or more rotational transformations, or to try to form a cycle and use cycle extension. These options are not exclusive. For example, when we are considering which rotational transformation to apply, we may want to search for one that will lead to a cycle. Order zero and order one crossovers (as per the crossover extension technique) are easy to find in this case. If we can not construct such a cycle, then we could either use the backtrack rotational transformation technique or the full crossover extension technique. Note that these two techniques are not intended to be used together. A crossover is essentially a sequence of rotational transformations. So if we cannot form a cycle using the backtrack rotational transformation technique, this implies that we will most likely not find a crossover either. One advantage of the backtrack rotational transformation technique over the crossover extension technique is that it is constantly trying to modify the path's endpoints. New endpoints may allow for immediate extension of the path. The crossover extension technique however is only trying to form a cycle so that cycle extension can be performed. The backtrack rotational transformation technique is also perhaps easier to implement.

Another issue relevant to the second stage of the search and the techniques being used is when to terminate the search, or in other words, when to give up. This is easiest if the crossover extension technique is being used: a maximum crossover order is specified, and the search is terminated if no crossover with order less than or equal the maximum is found. Otherwise, the algorithm is performing a sequence of rotational transformations. There are several search termination methods that can be employed, which we discuss in Section 3.4.2.

In the third stage, the bypass and chain extension techniques are complimentary, and can both be utilized in a single algorithm. The basic idea of both these techniques is to extend the path by modifying an internal segment of the path. Note that successful extension of the path will allow us to return to stage two of the search (but not stage one, since the endpoints will not have changed). While these techniques can clearly be used to extend an existing path, their usefulness for a Hamiltonian cycle algorithm can be questioned. The problem is that these techniques are only executed when the other techniques are unable to extend the path, and thus are unable to form a cycle. Thus the bypass or chain extension techniques not only need to increase the path length, but also need to permit the other techniques to form a cycle. In theory, this is possible (especially if the crossover extension technique is used). However, in practise it is unclear if this extra work would result in corresponding increase in the success rate of the algorithm.

3.4.2 Search Termination

The issue of search termination is the question of how (or when) to stop the search. In this section we examine three different methods for terminating the search. These methods assume the algorithm is performing a sequence of rotational transformations. When evaluating these methods, we look at three different criteria. The first is how well does the method avoid repetition in the search. When using rotational transformations to modify the path, it is easy after a few transformations to obtain a previously-seen path. Since searching a path more than once will not produce any new results, we want to avoid repetition. The second criteria is how well does the algorithm avoid ignoring unexplored paths.

If the method is too strict in terminating the search (to avoid repetition), then unexplored paths will not be tried. Thus, we want an optimal balance between these two criteria. The third criteria is ease of implementation and efficiency of execution.

We now present and evaluate the three search termination methods.

vertexonce: Permit each vertex to occur as an endpoint of the path only once. If this leaves no choices for performing a rotational transformation then terminate the search. This method from Pósa's algorithm is easy to implement and helps avoid repeating paths we have considered before. However, occasionally repeating an endpoint via a different transformation may help us form a cycle.

rotationlimit: Limit the number of rotational transformations applied. If the backtrack rotational transformation technique is being used, then this is equivalent to limiting the search depth. This method from the HAM algorithm is easy to implement but allows for paths to be repeated in the search, and thus we do not prefer it.

edgeonce: Permit each rotation edge used in a rotational transformation to be only used once. (If the endpoint vertex is e and the neighbouring vertex in the path that the transformation uses is v then the rotation edge is (e, v).) This method from the SparseHam algorithm is less restrictive than the vertexonce method, but will still prevent paths from being repeated in the search. For this reason, we prefer this search termination method. However, the edgeonce method is less efficient (requiring more time and perhaps space) compared to the vertexonce method.

3.4.3 Initial Pruning

We discussed initial pruning in Section 3.3.1 with respect to backtrack algorithms. Much of that discussion also applies to initial pruning and heuristic algorithms. However, it is possible that initial pruning can impede the performance of a heuristic algorithm. The deletion of unnecessary edges (when pruning) may restrict the heuristic algorithm's options, and cause it to hit a dead-end and quit rather than be able to extend the path. The algorithm may temporarily need to use edges that can't be in any Hamiltonian cycle in order to rearrange the path and further extend it, replacing those edges later on.

Note that initial pruning can detect that no Hamiltonian cycle is possible on a given graph. By including such pruning in a heuristic algorithm, it gives the algorithm the ability to determine and report that certain graphs are non-Hamiltonian, rather than always fail in such cases.

3.4.4 The Search Method

In this section we examine how the search should be performed. As the heuristic algorithms extend the path, whether by normal extension or by the rotational transformation technique, the issue of which endpoint to use arises. As we discuss in Section 3.3.2 for backtrack algorithm design, we have two options: the singlepath method (use only one endpoint) and the doublepath method (use both endpoints). Heuristic algorithms using the doublepath method have implemented this type of search using two different approaches. The partial approach is to start with the singlepath method and only switch to searching from the other endpoint if the algorithm reaches a dead end. The full doublepath approach is to consider expanding from either endpoint of the path at each stage of the search.

Using an example of second-order crossover from Kocay and Li [20], we can show that certain graphs require the full doublepath approach in order for a Hamiltonian cycle to be found. See Figure 3.2 for our example. The initial path P could either be Hamiltonian or could represent a stage in the search where cycle extension is needed to extend the path. Our goal is to form a cycle to allow the use of cycle extension. We first must perform a rotational transformation on the U endpoint (to obtain path P') and then perform two rotational transformations on the other endpoint before obtaining cycle C. Inspection of the initial path P shows that if we only perform rotational transformation on one endpoint there is no way to form the cycle.

From this example, we expect that the full doublepath method will be the most successful of the different search methods. And since it does not involve much additional computation, it will most likely be efficient.

There is another possible benefit to using the doublepath approach. By expanding the path from both endpoints, the final path (even if not Hamiltonian) will tend to be longer than the paths obtained using the singlepath approach. So if obtaining long non-Hamiltonian paths is an acceptable approximation to finding a Hamiltonian cycle, then the doublepath search method is clearly superior.

Note that the search method is not relevant if the crossover extension technique is used, since this replaces the rotational transformation technique and always tries to extend the path using cycle extension. In this case, we can not consider the path as being extended from a single endpoint. However, the algorithm's actions are similar to one that uses the doublepath method with the backtrack rotational transformation technique.

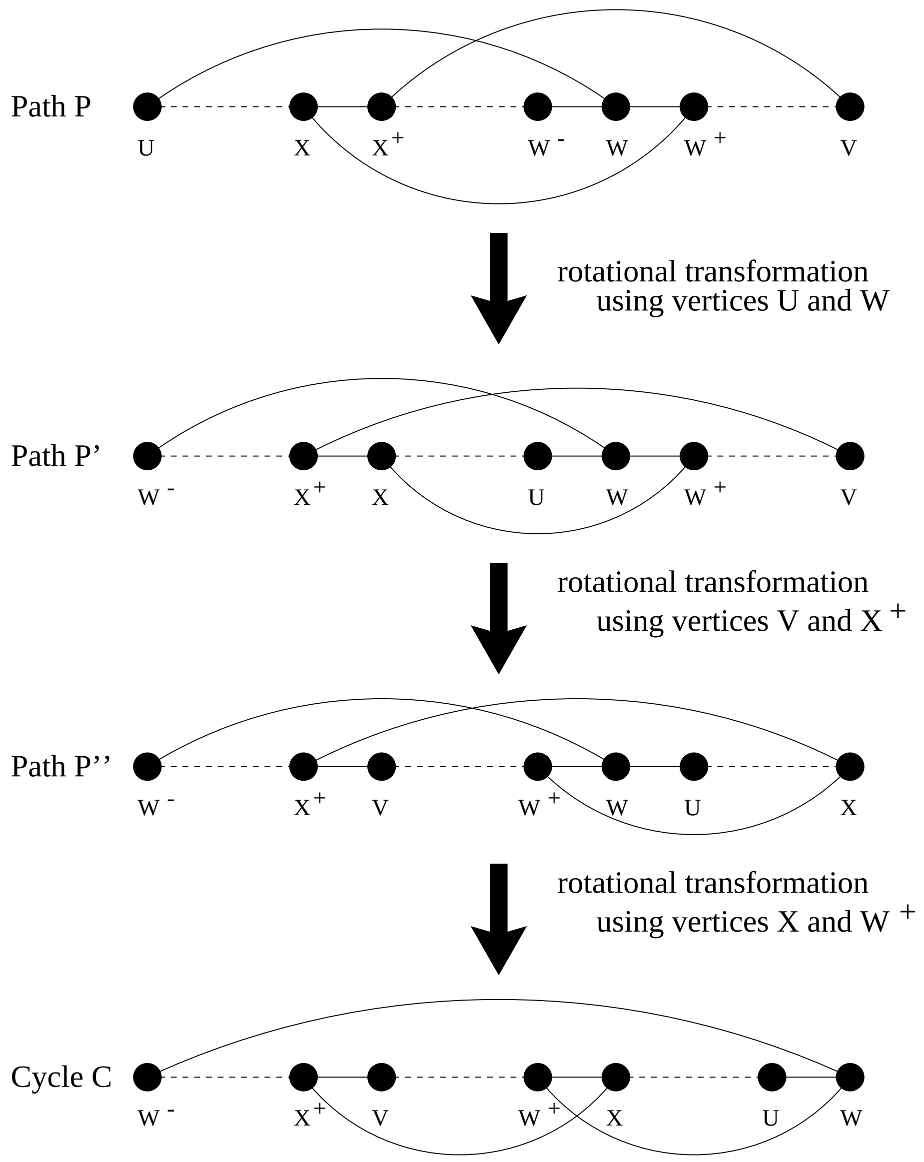

Figure 3.2: Sample graph for which the full doublepath search method is required.

3.4.5 Vertex Selection

In this section we examine the process of vertex selection. At each stage of the search, a heuristic Hamiltonian cycle algorithm has choices to make depending on the techniques it is using. We assume that the algorithm is using the singlepath search method, so the algorithm is trying to expand the path from a particular endpoint vertex. (Our discussion is easily generalized for the doublepath search method.)

With each neighbour of the endpoint, there are several possibilities. First, the neighbour is not in the path. Second, the neighbour is the other endpoint of the path. (A cycle exists which means that the cycle extension technique can be used.) Third, the neighbour is in the path, and the rotational transformation technique can be applied. The naive approach is to randomly select a neighbour of the endpoint, and perform the corresponding action. A more intelligent approach is to use heuristics to guide our selection. In the next section we discuss some of the basic heuristics that come to mind, including ones we have examined for the backtrack algorithm (see Section 3.3.3). In the subsequent section we introduce two heuristics based on a new technique of our devising which we call the non-path neighbours technique. Finally, we combine the different heuristics into a single vertex selection algorithm.

Another aspect to vertex selection is choosing the initial vertex to start a search with. One method is to choose a vertex at random. One heuristic is to choose the vertex of largest degree, to provide the most edges for forming the cycle at the end of the search. However, this is of questionable use if the cycle extension technique or the doublepath search method are used, since both these techniques can modify the starting vertex. Another approach is to try each vertex as the initial vertex by executing the algorithm n times (or until a Hamiltonian cycle is found).

Basic Heuristics

The first obvious heuristic used by many of the Hamiltonian cycle algorithms we surveyed is to always select a neighbour (of the endpoint) not part of the current path if one exists, since this will immediately increase the length of the path.

If more than one neighbour is not in the current path, then we again have a choice to make. The obvious heuristic (which amazingly is not used by many of the heuristic algorithms we surveyed) is to follow a forced edge in preference to any other. Note that there cannot be more than one forced edge if initial pruning using graph reduction was performed. If there is no forced edge, then we can use the low degree first heuristic to choose between non-path vertices.

If all the neighbours of the endpoint are part of the current path, then we will be using the rotational transformation technique to modify the path. Our goal is still to extend the path, so we should first select a neighbour that will allow us to form a cycle. If a neighbour of the endpoint corresponds to vertex v_i in the path, then we want vertex v_{i+1} to be adjacent to the starting vertex $s(P)$. After forming a cycle, we can use the cycle extension technique to expand the path.

The Non-Path Neighbours Technique

In this section we introduce two new heuristics based on a new technique we devised called the non-path neighbours (*NPN*) technique. The technique is simple and easy to implement. We denote the number of non-path neighbours of a vertex v by $NPN(v)$. Initially, for each vertex v, we set $NPN(v) = d(v)$. When a vertex x is added to the path, then for each $w \in N(x)$ we decrease $NPN(w)$ by one. Basically, the technique tracks the number of neighbours of each vertex that are not members of the current path.

We can now use the *NPN* values to guide our search in two different ways. The first way we call the *NPN* path extension heuristic. Let us assume that the current endpoint of the path is e, and $NPN(e) > 1$, which means that e is adjacent to multiple vertices $\{v_1, v_2, \ldots\}$ not in the path. Instead of just choosing a random vertex v_i as we would normally do, we first check if any v_j has $NPN(v_j) > 0$. If so, this means that the vertex v_j is adjacent to another vertex not in the path. Thus we select v_j as the next vertex in the path, knowing that we will be able to expand the path again. This heuristic thus helps delay that point in time when the algorithm must use the rotational transformation or cycle extension to further extend the path.

The second way of using the *NPN* values to guide the search is in selecting a vertex on which to perform the rotational transformation. This *NPN* rotational transformation heuristic is a bit more complicated to compute, but has the same end result as the prior heuristic: we want the new endpoint of the path to have at least one neighbour not in the path, because this will allow us to easily expand the path. We start with the endpoint e of the path P having neighbours $\{v_1, v_2, \ldots\}$ which are all members of the path. (We exclude the vertex prior to e in the path – $P^-(e)$ – from this list.) The new endpoint after a rotational transformation using neighbour n_i will be the next vertex in the path after n_i, $P^+(n_i)$. Thus the heuristic is to select a neighbour n_j for which $NPN(P^+(n_j)) > 0$.

It is unclear how useful the *NPN* path extension heuristic will be. By prefering vertices adjacent to other non-path vertices, the algorithm avoids selecting vertices adjacent only to path vertices. These isolated vertices may be difficult to introduce into the path at a later point in time, and cause the algorithm to fail. So it is possible that this heuristic will tend to produce longer paths, but be less likely to form Hamiltonian cycles. Secondly, this heuristic only gets used when there are multiple non-path vertices adjacent to the current endpoint. Such situations would seem to occur more at the beginning of the search, rather than near the end, when the choices are much more constrained. Since the algorithms tend to fail only near the end, when their choices are exhausted, the heuristic's ability to improve an algorithm's performance is questionable.

The *NPN* rotational transformation heuristic on the other hand seems much more useful, especially if the backtrack rotational transformation technique is not being used. Basically, if one of the possible rotational transformations will lead to a path extension, this heuristic will find it. So the heuristic causes the backtrack rotational transformation technique to require one less level of search (approximately). Without the backtrack rotational transformation technique, the algorithm can easily become stuck after performing a few rotational transformations, selected at random. With the heuristic, it seems that the algorithm has a much better chance of eventually extending the path again.

A Combined Vertex Selection Algorithm

Using the different heuristics described above we are able to construct a combined algorithm for vertex selection. Figure 3.3 contains the vertex selection algorithm. It is executed at each stage of the search on a current path P and on a graph G. Note that this algorithm combined with the various heuristic algorithmic techniques forms a nearly complete heuristic Hamiltonian cycle algorithm.

The only part of the algorithm that requires some additional work is the last line, the point at which the algorithm is unable to immediately extend the path. We had the algorithm randomly select a neighbour which it could use to perform the rotational transformation. If we were using the backtrack rotational transformation technique, then we would want to add the different possible paths (after the transformations using the different neighbours) to the list of paths to explore. Additionally, we need to avoid repeating parts of the search that we have already done, and we need to terminate the search if we cannot avoid repetition, which indicates the search has reached a dead-end.

```
VertSelectAlg(graph G, path P) {
    Let N = N(e(P))
    If ∃v | v ∈ N, v ∉ P then {
        Let W = {x | x ∈ N, x ∉ P}.
        If ∃v | v ∈ W and d(v) = 2 then
            Choose v (edge (e(P), v) is forced)
        Else if ∃v | v ∈ W and NPN(v) > 0 then
            Choose v (the NPN rotational transformation technique)
        Else choose v ∈ W such that d(v) is a minimum
            (the low degree first heuristic)
    }
    Else { (we must select a vertex already in the path)
        If ∃v | v ∈ V and NPN(P+(v)) > 0 then
            Choose v (the NPN rotational transformation heuristic)
        Else if ∃v | v ∈ V and (P+(v), s(P)) ∈ E(G) or v = s(P) then
            Choose v (form a cycle for the cycle extension technique)
        Else (we are unable to immediately expand the path)
            Choose a random v ∈ V (to apply the rotational transformation)
    }
}
```

Figure 3.3: The vertex selection algorithm.

3.5 Conclusions

In this chapter we have examined issues involved with the design of Hamiltonian cycle backtrack and heuristic algorithms. We first discussed some general aspects of algorithm design, and introduced our definition of an efficient algorithm as an algorithm which minimizes the time required to solve the problem. We then used this definition as the basis for our analysis of the different algorithmic techniques and heuristics available for the two different types of algorithms.

Our examination of backtrack Hamiltonian cycle algorithms involved three areas: pruning, the search method, and vertex selection. Our most important work involved pruning. We consolidated and organized the different pruning techniques taken from the algorithms and theorems reviewed in Chapter 2 by establishing two sets of categories of pruning. We defined initial pruning and search pruning as one set of categories and graph reduction and global checking as the other set of categories. We also introduced the graph collapse technique, which may be of possible use when trying to solve hard non-Hamiltonian graphs.

Our discussion of heuristic Hamiltonian cycle algorithm design dealt with many different issues: algorithm techniques, search termination methods, initial pruning, the search method and vertex selection. Our analysis indicates that there exist strong reasons to expect certain methods to perform (in general) better than others. In particular, we expect the edgeonce search termination method is the best choice of the three we examined and the full doublepath search method is the best choice of the three search methods available to heuristic algorithms. Our discussion of the issue of vertex selection (in Section 3.4.5) is where we introduce several new improvements. We develop new heuristics based on a new technique we call the non-path neighbours technique. We also develop a new vertex selection algorithm incorporating these heuristics and others which is expected to increase an heuristic algorithm's efficiency as compared to previous strategies for vertex selection.

Throughout our discussion on algorithm design we observed cases where the use of a particular technique or heuristic depended upon the nature of the graph the algorithm would be used on. For example, the use of graph reduction in initial pruning is useless if no degree 2 vertices exist in the graph. The use of global checking, and in particular detection of cutpoints, is extremely useful for random graph classes such as $G_{n,m}$ for which a high proportion of the non-Hamiltonian graphs are not biconnected. Other examples exist. The point is that algorithmic design should not be done in a vacuum: any knowledge about the types or properties of graphs that one is trying to solve can be important for the design process. This applies even to algorithms solving arbitrary random graphs, for the probability distribution used in generating these graphs will affect the probabilities of the graphs have different properties, which will affect the design.

Chapter 4

Generalized Knight's Tours

4.1 Introduction

The knight's path problem (also referred to as the knight's tour problem) [1] is based upon the game of chess [2]. It involves moving a knight from square to square on the chessboard, visiting each square only once, until all the squares have been visited. The knight's circuit problem is an extended version of the knight's path in which the knight is required to return to its starting square after visiting all the other squares. Various solutions to both of these problems have been devised over the years. See [3] for a discussion.

The knight's circuit problem is of interest to us because it is a subset of the Hamiltonian cycle problem. As such, it can serve as a method of generating problem instances to test Hamiltonian cycle algorithms. In particular, we would like to find hard instances (or hard sets of instances) that our algorithms do poorly on (see Chapter 5). In addition, insights we gain into the knight's tour problem may be of interest to mathematicians.

The knight's circuit problem can be easily translated into a specific instance of the Hamiltonian cycle problem if we consider each square on the board to be a vertex v in V, and add an edge (v_i, v_j) to E if a knight can move from square v_i to square v_j. Note that the basic knight's circuit problem is only one instance of the Hamiltonian cycle problem, and thus of limited use for testing Hamiltonian cycle algorithms. Furthermore, various solutions to the knight's path and knight's circuit problems have been presented over the years. (See [3] for a discussion.) The logical next step is to generalize the knight's circuit problem in some fashion, to get a set of related problems. Such a set could be used as a testbed of

[1] The literature is not consistent in its terminology for distinguishing between knight's tours and knight's circuits. Some authors use the term "knight's tour" to refer to both paths and cycles (i.e. [27]). Our terminology comes from [12].

[2] The game of chess involves moving pieces on an 8×8 board, with alternating squares colored black and white. The knight moves in an L-shaped pattern, 1 square horizontally or vertically, and 2 squares in the perpendicular direction.

Hamiltonian cycle instances for testing Hamiltonian cycle algorithms.

In this chapter we investigate generalizations of the knight's circuit problem. In Section 4.2 we examine previous work, which has mainly dealt with varying the size of the chessboard. We refer to this as the rectangular knight's circuit problem. In Section 4.3 we introduce the generalized knight's circuit problem, where the move of the knight is allowed to vary along with the size of the chessboard. In the remainder of the chapter we present various proofs, experimental results and observations on the existence and non-existence of knight's circuits for this generalized problem. We present our conclusions in Section 4.8.

4.2 Previous Work

The majority of the work done on generalized knight's tour problems has been on arbitrarily-sized rectangular chessboards. The rectangular knight's circuit problem is formulated as follows: find a circuit of knight's moves on a $n \times m$ chessboard (where $n \leq m$ by convention). Of interest is the set of values of n and m for which circuits exist (or do not exist).

Schwenk [27] determines for which values of n and m circuits cannot exist, and proves that circuits exist for all other values. Cull and De Curtins [12] provide a similar proof to show the existence of circuits and tours for most values of n and m. Their results can be summarized as follows. An $n \times m$ board, with $n \leq m$, has a knight's circuit unless one or more of three conditions hold [27]:

- nm is odd
- $n = 1, 2, 4$
- $n = 3$ and $m = 4, 6, 8$

A knight's path exists for $n \times m$ boards where $m \geq n \geq 5$ [12]. [3]

The arguments and proof techniques used in these papers will be briefly reviewed here. The proofs presented in the latter part of this paper are in part extensions of these techniques.

The proofs for the existence of tours or cycles presented in [12] and [27] are inductive construction proofs. The authors presented a set of tours or cycles for specific, small boards, and then showed how these solutions could be connected together to solve larger boards. An inductive argument was used to conclude the proof.

[3] The authors do not discuss the existence of knight's path on boards where one of n, m is less than 5. Obviously, those boards with cycles (such as 3×10) will also have knight's paths; boards with no circuit (such as any of size $4 \times m$) might have a knight's path as well.

More specific proofs were used to determine the conditions for which a knight's circuit could not exist (see [27]). Some of these proofs utilized various conditions from graph theory about the existence of Hamiltonian cycles, which will not be discussed here. The more significant of these proofs are presented below.

Lemma 1 *A knight's moves on a chessboard must alternate between black and white squares.*

Proof. Let us represent an arbitrary square on the chessboard as (i, j), with $1 \leq i \leq n$ and $1 \leq j \leq m$. We define the parity of the square as even if $(i + j) \bmod 2 = 0$ and odd if $(i + j) \bmod 2 = 1$. By this definition, all squares of a single color are a single parity (if all even parity squares are white, then all black squares are odd parity). If the knight starts at location (i, j) with parity $p = (i + j) \bmod 2$, then its location after one move will be $(i \pm \{1, 2\}, j \pm \{2, 1\})$, and its new parity will be $q = (i + j \pm \{1, 3\}) \bmod 2$. Clearly, $q \neq p$, which demonstrates that the knight will switch parity with each move, and thus will switch colors. □

Theorem 12 *If the $n \times m$ board of a rectangular knight's circuit problem contains an odd number of squares, no circuit exists.*

Proof. The proof follows directly from Lemma 1. Since in a circuit the knight must end on a color opposite from its starting color in order to return to the start, the knight must traverse an equal number of black and white squares, which means that the total number of squares on the board must be even. □

Theorem 13 *A $4 \times m$ board cannot have a circuit for any value of m.*

Proof. [4] Assume that a circuit C exists for the $4 \times m$ board. Let us partition the squares of the board into two sets (or colors), X and Y. The top and bottom row of the board is in set X, and the middle two rows of the board are in set Y. Note that from a square in set X, a knight can only reach a square in set Y (see Figure 4.1). In circuit C, any move to a square in set X must therefore mean the knight came from a set Y square, and will next move to a set Y square. Since $|X| = |Y|$, circuit C must therefore alternate between squares of set X and set Y. By Lemma 1 we know that any circuit must alternate between black and white squares. This implies that all the squares of set X must be just one color. But from the partition of the board, it is clear that both sets contain both black and white squares. So there is a contradiction, and therefore no circuit C exists for the $4 \times m$ board. □

[4] Schwenk [27] credits Louis Pósa for discovering this proof.

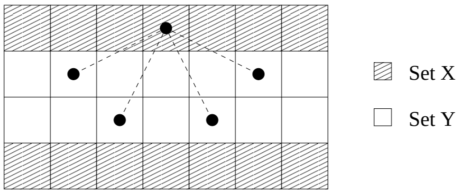

Figure 4.1: The partition of a chessboard into sets X and Y; a knight on a square in set X can only reach a square in set Y.

4.3 The Generalized Knight's Circuit Problem

Initial experiments on the rectangular knight's circuit problem using both backtrack and heuristic Hamiltonian cycle algorithms revealed that circuits were easily found, and that the difficulty (of finding circuits) increased only slowly as the problem size grew larger. In our search for harder problems, a natural step was to further generalize the knight's circuit problem by allowing the size of the knight's move to vary. We specify a generalized knight's move by a pair (A, B), $A \leq B$, which signifies that a move can be made from a square (i, j) to any square $(i \pm \{A, B\}, j \pm \{B, A\})$. As an example, the standard knight's move is $(1, 2)$. We allow the board size $n \times m$ to vary as well. We call this problem the generalized knight's circuit problem, and any instance is specified by the 4-tuple $(A, B) - n \times m$.

As has been done for the rectangular knight's circuit problem, we would like to determine for the generalized knight's circuit problem which instances or sets of instances have a circuit and which do not. We ignore certain trivial cases: $A = 0$ (piece moves in a straight line) and $n \leq B$ (piece forced to move in a straight line). We also ignore $(1, 2) - n \times m$ since it has been previously discussed.

To begin our investigation of the generalized knight's circuit we obtained empirical results using a backtrack Hamiltonian cycle algorithm [5]. To help organize our results, we define an *instance class* of the generalized knight's circuit problem to be a set of instances $(A, B) - n \times m$ with A, B, and n fixed, and with m allowed to vary [6].

Our initial experiments on different instance classes produced a variety of results. For some classes, we easily found circuits for many values of m. For other classes, our algorithm quickly indicated that no circuits existed for all values of m that we tested for. A few instance classes only seemed to have circuits for particular values of m. These preliminary results indicate that the question of a circuit existing for a particular instance (or instance class) of

[5] Our backtrack Hamiltonian cycle algorithm uses the singlepath search method, full initial pruning (graph reduction and global checking), local pruning using graph reduction. Vertex selection is done using the low degree first heuristic, and the initial vertex is selected randomly.

[6] A sample instance class is $(1, 4) - 5 \times m$.

the generalized knight's circuit problem is non-trivial and therefore worthy of investigation. In Section 4.4 we develop proofs for the non-existence of circuits for particular instance classes, using the work of Schwenk and others (as discussed in Section 4.2) for inspiration. Based on Theorem 13, we develop a more complex proof technique we call the *partition proof technique*. This technique and the proofs resulting from it are presented in Section 4.5. In Section 4.6 we perform an empirical investigation of the existence (or non-existence) of cycles on instance classes of the generalized knight's circuit problem. The results display interesting patterns in the existence of circuits for various sets of instances which we make observations and conjectures about. In Section 4.7 we apply these results to a specific instance class $(1, 4) - 5 \times m$ and develop some proofs concerning the existence and non-existence of circuits within this instance class.

4.4 Non-Existence Proofs for the Generalized Knight's Circuit Problem

We consider a $n \times m$ board with the upper left corner labelled $(1, 1)$, and the bottom right corner labelled (n, m).

Theorem 14 *If $A + B \bmod 2 = 0$, then the $(A, B) - n \times m$ instance of the generalized knight's circuit problem has no circuit.*

Proof. Consider the parity of the squares the piece can move between. If the piece starts in an arbitrary square (i, j) with parity $p = (i + j) \bmod 2$, it will move to a square $(i \pm \{A, B\}, j \pm \{B, A\})$, with parity $q = (i + j \pm A \pm B) \bmod 2$. Thus, $p = q$, and the piece can never reach a square of opposite parity. [7] □

If $A = B$, then no circuit is possible by this theorem. Therefore we only need to consider instances where $A < B$.

Corollary 14.1 *An $(A, B) - n \times m$ instance of the generalized knight's circuit problem has no circuit if $nm \bmod 2 = 1$.*

Proof. Since a piece's moves must change parity, the piece must alternate between black and white squares, just like a knight. This means that Theorem 12 holds for the generalized knight's circuit problem, independent of the values of A and B, and therefore for a circuit to exist the number of squares comprising the board must be even. □

[7] An alternate way to start the proof is to realize that a piece with an even sum of A and B can never move from square $(1, 1)$ to square $(1, 2)$.

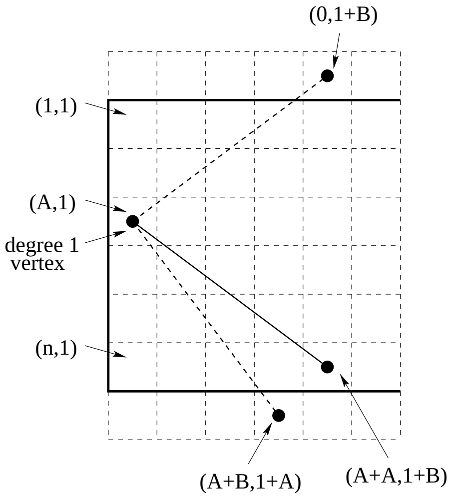

Figure 4.2: For $A + B > n$, square $(A, 1)$ has a maximum degree of 1.

Theorem 15 *An $(A, B) - n \times m$ instance has no circuit if $B = kA, A > 1, k \geq 1$*

Proof. Consider a piece with a move of the form (A, kA) on square (i, j), that ends up on square $(i + x, j + y)$ after an arbitrary number of moves. The displacements in position, x and y, must each be a linear combination of A and kA. So $x = sA + tkA = A(s + tk)$. If $A > 1$, then $x \neq 1$ for all values of s, t, k. Therefore, the piece can never reach a square adjacent to (i, j), and thus no circuit (or path) is possible for such a piece, on any sized board. □

Theorem 16 *An $(A, B) - n \times m$ instance has no circuit if $A + B > n$.*

Proof. Consider the square $(A, 1)$ of an arbitrary $(A, B) - n \times m$ instance. The only possible destinations of this square are $(A + A, 1 + B)$ and $(A + B, 1 + A)$. (Square $(A - A, 1 + B)$ is not on the board.) If $A + B > n$, then the second square $(A + B, 1 + A)$ is not on the board, and the maximum possible degree of square $(A, 1)$ is 1. From basic graph theory, each vertex in a graph must be adjacent to at least two other vertices for a Hamiltonian cycle to exist. Thus no circuit is possible. See Figure 4.2. □

Theorem 17 *An $(A, B) - n \times m$ instance has no circuit if $A + B < n \leq 3A$.*

Proof. Consider the square $S = (1 + A, 1 + B)$. We show that the square has 3 neighbours of degree 2 if the conditions hold. The 3 neighbours of S are $N_1 = (1, 1)$, $N_2 = (1 + 2A, 1)$ and $N_3 = (1 + A + B, 1 + B - A)$. Note that for N_3 to be on the board, $n \geq 1 + A + B$. If this holds, then N_2 is also on the board since $A < B$. So if $n > A + B$, then S has 3 neighbours. See Figure 4.3.

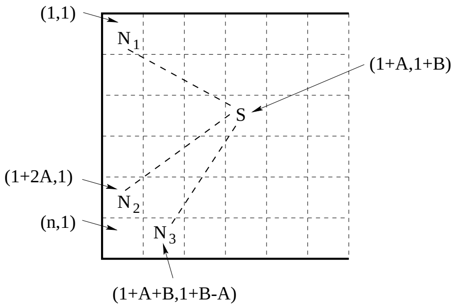

Figure 4.3: Square $(1+A, 1+B)$ has 3 neighbours of degree 2.

We now show that each of N_1, N_2, N_3 has a maximum degree of 2. Square N_1 has only two neighbours, S and $(1+B, 1+A)$ since any subtraction would leave one coordinate ≤ 0. Square N_2 has neighbours S, $(1+2A-B, 1+A)$, $(1+2A+A, 1+B)$ and $(1+2A+B, 1+A)$. If we set the condition $n < 1+2A+A$, which simplifies to $n \leq 3A$, then the last two neighbours will not be on the board, and N_2 will be of degree 2.

If we let square $N_3 = (J, K)$, so $J = 1+A+B$ and $K = 1+B-A$, then the neighbours of N_3 are $X_1 = (J+A, K+B)$, $X_2 = (J+A, K-B)$, $X_3 = (J-A, K+B)$, $X_4 = (J-A, K-B)$, $X_5 = (J+B, K+A)$, $X_6 = (J+B, K-A)$, $X_7 = (J-B, K+A)$ and $X_8 = (J-B, K-A)$. $X_3 = (1+B, 1+2B-A)$ is on the board. $X_7 = (1+A, 1+B)$ is square S, which is on the board.

$X_1, X_2 = (1+2A+B, 1+B-A \pm B)$ are not on the board, since $n \leq 3A$ and $B > A$. $X_4 = (1+B, 1-A)$ is not on the board for $A \geq 1$, and we are ignoring the $A = 0$ case as trivial. $X_5, X_6 = (1+A+2B, 1+B-A \pm A)$ are not on the board because $A+2B > 3A$ and $n \leq 3A$. $X_8 = (1+A, 1+B-2A)$, which is not on the board if $1+B-2A \leq 0$. Rearranging we get the condition $B < 2A$. If this holds, then N_3 is of degree 2. However, combining the conditions $A+B < n$ and $n \leq 3A$ gives us $A+B < 3A$, or $B < 2A$, so this third condition is already included in the prior conditions

Square S therefore has 3 neighbours which are each (at most) of degree 2 if $A+B < n \leq 3A$. By Theorem 2 no circuit is possible in such a graph, and the theorem is proven. □

4.5 The Partition Proof Technique

The proofs in this section utilize a proof technique we generalized from Theorem 13, which we refer to as the *partition proof technique*. We first explain this technique, then proceed with the proofs that are based on this technique.

The proof technique consists of three steps. We start with a graph $G = (V, E)$ for which we wish to prove that no Hamiltonian cycle exists [8]. The first step is to form a partition $\Pi(V) = \{P_1, P_2, \ldots, P_k\}$ with $P_i = \{v_{i1}, v_{i2}, \ldots, v_{ik_i}\}$ where $P_1 \cup P_2 \cup \ldots \cup P_k = V$ and $\forall i \neq j, P_i \cap P_j = \emptyset$.

The second step is to construct what we refer to as the contraction graph. We define the contraction graph of graph $G = (V, E)$ as the graph $T = (\Pi(V), E_\Pi)$ satisfying $\forall i, j \mid 1 \leq i, j \leq k, (P_i, P_j) \notin E_\Pi \Rightarrow \forall a, b \mid 1 \leq a \leq k_i, 1 \leq b \leq k_j, (v_{ia}, v_{jb}) \notin E$. (This last constraint can be restated as $\forall 1 \leq i, j \leq k, (P_i, P_j) \in E_\Pi$ if $\exists x \in P_i, y \in P_j \mid (x, y) \in E$). Note that a contraction graph may have an edge between two vertices whose corresponding partition elements are not joined by an edge, although we never actually construct a partition in which this occurs. The condition is set up in this way to allow the composition of two contraction graphs to itself be a contraction graph (see Lemma 2 below). Since vertices can have edges to themselves, the graph can be non-simple.

The third step is to prove that the contraction graph T of step two combined with the properties of the partition Π obtained in step one make it impossible for a circuit to exist. This is done by selecting a particular element of the partition as a cutset and then showing that the resulting number of components (after the cutset is removed) is greater than the size of the cutset. By Theorem 5, no Hamiltonian cycle can exist in such a graph.

We now prove two lemmas concerning partitions and contraction graphs that will be needed for our proofs. First, we consider two different partitions $\Pi_i(V) = \{P_{i1}, \ldots, P_{ik_i}\}$ and $\Pi_j(V) = \{P_{j1}, \ldots, P_{jk_j}\}$ with corresponding contraction graphs $T_i(G) = (\Pi_i(V), E_i)$ and $T_j(G) = (\Pi_j(V), E_j)$ on the graph $G = (V, E)$. We define a refined partition $\Pi_{ij}(V) = \{P_1, \ldots, P_{k_i \times k_j}\}$ such that each element of Π_{ij} is formed by the intersection of an element from Π_i and one from Π_j. We refer to the formation of a refined partition in this manner as the merger of the two original partitions.

We define the refined contraction graph $T(G) = T_i(G) \otimes T_j(G) = (\Pi_{ij}(V), E_{ij})$ where $\otimes$ is the cross product operator, $\Pi_{ij}(V)$ is the refined partition of $\Pi_i(V)$ and $\Pi_j(V)$. If we consider two arbitrary elements of $\Pi_{ij}(V)$, $A = A_i \cap A_j$ and $B = B_i \cap B_j$, where $A_i, B_i \in \Pi_i(V)$ and $A_j, B_j \in \Pi_j(V)$ then $(A, B) \in E_{ij}$ iff $(A_i, B_i) \in E_i$ and $(A_j, B_j) \in E_j$.

Lemma 2 *The refined graph $T = (\Pi, E')$ is a contraction graph.*

Proof. To show that graph T is a contraction graph we must prove that $(A, B) \notin E' \Rightarrow \forall x \in A, y \in B; (x, y) \notin E$ where A, B represent both vertices of T and elements of the partition Π, and x, y represent vertices of the original graph with edgeset E.

[8]While the technique works for all graphs in general, we apply it only to graphs constructed from a piece moving over a rectangular grid.

To prove the implication we assume an arbitrary $(A, B) \notin E'$. From the definition of the formation of T above, this implies that $(A_1, B_1) \notin E_1$ or $(A_2, B_2) \notin E_2$. Without loss of generality, we assume that $(A_1, B_1) \notin E_1$. By definition of a contraction graph, we have $\forall x \in A_1, y \in B_1, (x, y) \notin E$. Since $A = A_1 \cap A_2$, $A \subseteq A_1$ and similarly $B \subseteq B_1$. Thus $\forall x \in A, y \in B, (x, y) \notin E$, which concludes the proof. □

Lemma 3 *If the partition Π of a graph G induces a contraction graph T with a degree 1 vertex x with neighbour y (corresponding to elements X and Y), then Y is a cutset of G whose deletion will create $|X|$ components from the element X.*

Proof. By our definition of a contraction graph, a vertex in element X has no edges to any vertices other than those in element Y. When Y is removed, each of those edges is removed, isolating each vertex in X and making each an individual component, thus creating $|X|$ components. □

As an example of the *partition proof technique*, we reprove Theorem 13 (which states that a $4 \times m$ board cannot have a circuit for any value of m) using this technique. We partition the board as before: the top and bottom rows are in set X and the middle two rows are in set Y. We now consider a second partition Π_2 of the board into sets B and W corresponding to the black and white squares respectively. By Lemma 1 we know the knight must alternate between black and white squares, which restricts the the corresponding contraction graph T_2 to a single edge connecting the two vertices B and W. The contraction graphs T_1 and T_2 for partitions Π_1 and Π_2 respectively are displayed in Figure 4.4.

We merge the two partitions to get a new partition Π with elements X_B, X_W, Y_B, Y_W and a new contraction graph T according to Lemma 2 (see Figure 4.5). Since edge (X, X) does not exist in contraction graph T_1, there is no edge between X_B and X_W in the refined contraction graph even though edge (B, W) exists in graph T_2.

From inspection of the board (see Figure 4.1), it is clear that the size of each element of Π equals $\frac{1}{4}|V|$. Deleting Y_W, a cutset, produces $\frac{1}{4}|V|$ components from element X_B by Lemma 3, plus one extra component (at least) from the remainder of the graph (elements X_W and Y_B). Since $|Y_W| = \frac{1}{4}|V|$, by Theorem 5 no circuit exists, and the proof is complete.

Theorem 18 *An $(A, B) - n \times m$ instance has no circuit if $2A + B \leq n \leq 4A$.*

Proof. This proof is an extension of Theorem 13 for the generalized knight's circuit problem, and is very similar to our new version of the proof presented above. We partition an arbitrary board into 2 partitions X and Y, where X contains the top and bottom A

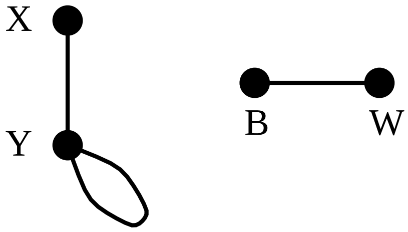

Figure 4.4: The two contraction graphs for the two partitions for Theorem 13.

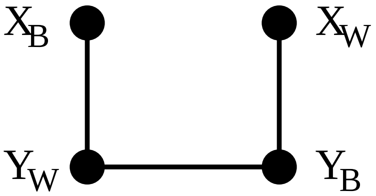

Figure 4.5: The contraction graph for the merged partition for Theorem 13.

rows, and Y contains the other $n-2A$ rows in the middle. (See Figure 4.6.) If $n-2A \geq B$ then squares in X can only reach squares in Y, and the resulting contraction graph is the same as the left one in Figure 4.4.

We make a second partition of the board into sets B and W containing the black and white squares respectively. The merger of these two partitions creates a contraction graph identical to that of Figure 4.5. Since the original X and Y partitions contain an equal number of black and white squares, $|X_B| = Am$ and $|Y_W| = (n-2A)m/2$. Using Lemma 3 with element Y_W as the cutset produces $Am+1$ components at least. By Theorem 5 no circuit can exist if $Am+1 > (n-2A)m/2$. Rearranging gives us the condition $n \leq 4A$. Combining this with the condition $n-2A \geq B$ results in the final condition $2A+B \leq n \leq 4A$. □

Theorem 19 *An $(A,B)-n\times m$ instance has no circuit if for some $k \geq 1$, $B > A(2k-1)$ and $n = B + A(2k-1)+1$.* [9]

Proof. The general idea behind this proof is to specify a partition $\Pi = \{X, Y, Z\}$ that creates a contraction graph with $(X,Y) \in E$, X degree 1, and $|X| = |Y|$. This will satisfy the requirements for Lemma 3. The structure necessary for this partition allows us to derive the conditions stated in the theorem.

Note that n must be even if $n = B + A(2k-1)+1$. $2k-1$ is always odd, which means the product $A(2k-1) \bmod 2 = A \bmod 2$. Since we want $(A+B) \bmod 2 = 1$ by Theorem

[9]For $k=1$, this reduces to $B > A$ and $n = B+A+1$. Since $B > A$ is required by Theorem 14, this means that for any instance $(A,B)-n\times m$, if $n = A+B+1$ no circuit exists.

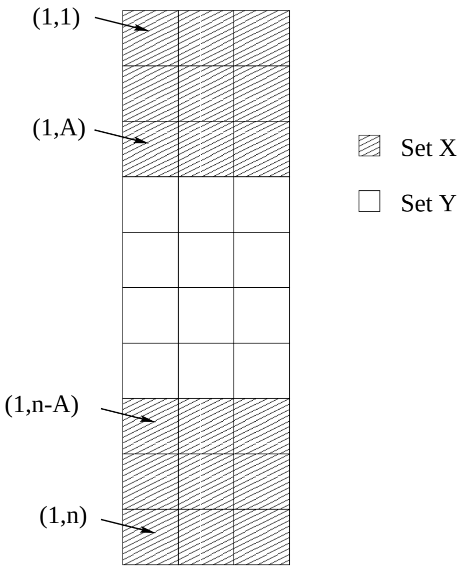

Figure 4.6: The first partition of the board for Theorem 18.

14, n must be even.

Therefore the board can be divided into two halves: the top $\frac{n}{2}$ rows and the bottom $\frac{n}{2}$ rows. The partition is arranged as follows. For the top $\frac{n}{2}$ rows, every A^{th} row alternates between being in X and Y, with all other rows being in Z. So set X contains rows $1+2A, 1+4A, \ldots, 1+2A(k-1)$. Set Y contains rows $1, 1+A, 1+3A, \ldots, 1+A+2A(k-1)$. Note that k equals the number of X and Y rows in the top half of the board. The partition of the bottom half is a mirror image of the top half. Rows $n, n-2A, \ldots, n-2A(k-1)$ are in X, rows $n-A, n-3A, \ldots, n-A-2A(k-1)$ are in Y and the rest are in Z. See Figure 4.7.

Vertex X in the contraction graph is degree 1 if the only valid move from a vertex in element X is to a vertex in Y. On the board, any move from a square in X by a distance A in the vertical direction will end up on a square in Y since in the construction the rows in X and Y are separated by a distance of A. What about moves with a vertical displacement of B? We want the construction so that for each X row q, row $q \pm B$ corresponds to a row in Y. So for row 1 (in X), row $1+B$ must be in Y. Similarly for the other rows in X in the top half of the board. The easiest way to fulfill this condition is for the set of rows in X in the top half of the board to be a distance B away from the corresponding Y rows in the bottom half of the board. So row $n-A-2A(k-1)$, the topmost row in Y in the bottom half of the board, must actually be row number $1+B$ to be a distance B from the top row in X in the top half (row 1). Since each row in X in the top half of the board is separated by a distance of $2A$ as is each row in Y in the bottom half, this means that each of the rows in X in the top half of the board are separated by a distance B from a corresponding row in Y in the bottom half of the board. (So the last row in X in the top half, row $1 + 2A(k-1)$ is a distance B from the bottommost row in Y in the bottom half, row n). See Figure 4.7. By symmetry, the same argument holds for the rows in X in the bottom half, and the rows in Y in the upper half. Therefore squares in partition X can only reach squares in partition Y, and therefore vertex X in the contraction graph is degree 1.

In addition, there are $2k$ rows in X and $2k$ rows in Y so $|X| = |Y|$. Using Lemma 3 with element Y as the cutset gives us $|X|$ components from element X and at least one component from element Z. $|X| + 1 > |Y|$ so by Theorem 5 no circuit is possible.

We must now simply derive the equations defining the relationship between A, B, n and k which allow this partition of the board to exist. First, we know that the k^{th} row in Y must remain in one half of the board. The k^{th} row in Y in the upper half corresponds to row $1+A+2A(k-1)$. This gives us the condition $1+A+2A(k-1) \leq \frac{n}{2}$. We have the other restriction that the $n - A - 2A(k-1)$ row in Y in the bottom half must be row number $B + 1$. This gives us the condition $B + 1 = n - A - 2A(k-1)$. Rearranging this second equation gives us $n = B + A(2k-1) + 1$. Plugging this into the first and manipulating gives us $B > A(2k-1)$. These are the two equations specified in the theorem, so the proof is complete. □

4.6 An Empirical Investigation of the Generalized Knight's Circuit Problem

The various theorems that were proven about the generalized knight's circuit have shown that no circuit exists for certain instance classes. A list of these classes for small values of A and B is presented in Table 4.1, along with the relevant theorem for that instance. Note that possible values for the piece moves and smallest value for n are restricted according to Theorem 14 and the trivial cases discussed in Section 4.3. Also, for several instance classes more than one theorem can be applied; only one is listed in the table.

To determine for which instances (and instance classes) circuits do exist, we turn to empirical methods. The proof techniques of the previous sections, culminating in Table 4.1, are a starting point but are not enough. We therefore generated and tested specific instances of the generalized knight's circuit problem using our backtrack Hamiltonian cycle algorithm with the goal of determining for what values of A, B, n and m circuits do or do not exist. We avoided values of A, B and n corresponding to Table 4.1, and restricted our

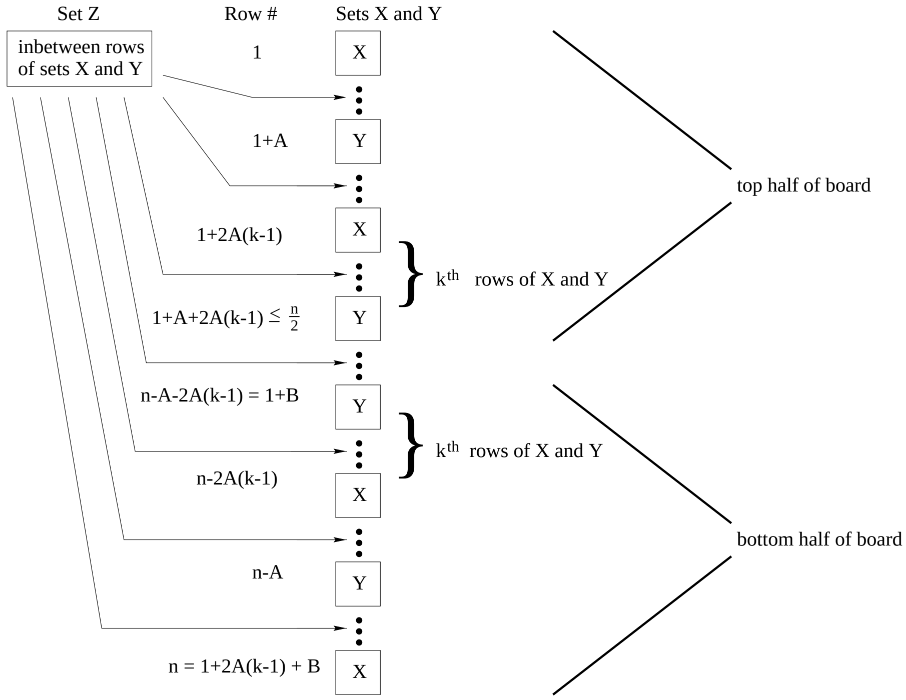

Figure 4.7: The partition of the board for Theorem 19.

Table 4.1: Instance classes of the generalized knight's circuit problem for which no circuit exists.

Instance Class	Relevant Theorem	Instance Class	Relevant Theorem
$(1,4)-6\times m$	19	$(3,4)-5\times m$	16
$(1,4)-8\times m$	19	$(3,4)-6\times m$	16
$(1,6)-8\times m$	19	$(3,4)-8\times m$	17
$(1,6)-10\times m$	19	$(3,4)-9\times m$	17
$(1,6)-12\times m$	19	$(3,4)-10\times m$	18
$(1,8)-10\times m$	19	$(3,4)-11\times m$	18
$(1,8)-12\times m$	19	$(3,4)-12\times m$	18
$(1,8)-14\times m$	19	$(3,6)-n\times m$	15
$(1,8)-16\times m$	19	$(4,5)-6\times m$	16
$(2,3)-4\times m$	16	$(4,5)-7\times m$	16
$(2,3)-6\times m$	17	$(4,5)-8\times m$	16
$(2,3)-7\times m$	18	$(4,5)-10\times m$	17
$(2,3)-8\times m$	19	$(4,5)-11\times m$	17
$(2,5)-6\times m$	16	$(4,5)-12\times m$	17
$(2,5)-8\times m$	19	$(4,5)-13\times m$	18
$(2,7)-8\times m$	16	$(4,5)-14\times m$	18
$(2,7)-10\times m$	19	$(4,5)-15\times m$	18
$(2,7)-14\times m$	19	$(4,5)-16\times m$	18

examination to small values of A, B and n. For each instance class, m was tested from $n+1$ [10] to a reasonably large value. (m was restricted to even values if n was odd.) For some instance classes, positive results were obtained for large values of m while negative results could only be obtained for much smaller values of m due to computational intractability. For certain instance classes with larger values of n, the algorithm ran to completion for only a few values of m before the time required became intractable.

Note that our backtrack algorithm was developed to solve Hamiltonian cycle problems in general, and was not optimized for the generalized knight's circuit problem. A more efficient algorithm utilizing symmetry considerations and other techniques might be able to extend these results.

Table 4.2 presents the results we obtained. For each instance class, the table lists the range of values of m for which circuits were found and for which no circuits exist. Missing values are unknown.

Our discussion of these results will be organized around the patterns exhibited in our results for the different instance classes. We describe and discuss three different classifications: *min-bounded*, *min-exist* and *periodic*. While we define these categories based on the existence of circuits over all m, we refer to certain instance classes as being in a category based solely on our limited empirical results. Further empirical or theoretical work could change this.

For some instance classes circuits were found for all values of m greater or equal to some value m_n, which we define as the minimum board size for which no larger board length with no circuit exists. We call these instance classes *min-bounded*. From Table 4.2 we can identify the following instance classes $(1,4)-5\times m$, $(1,4)-7\times m$, $(1,4)-9\times m$, $(1,4)-10\times m$, $(2,3)-5\times m$, $(2,3)-10\times m$, $(2,5)-7\times m$ and $(3,4)-7\times m$ which seem to be *min-bounded*. Note that for some of these instance classes, circuits were found for some values of $m < m_n$.

The actual existence of *min-bounded* instance classes is an open question. Constructive induction proofs showing solutions for all $m \geq m_n$ would demonstrate their existence; proving that a particular instance class is not *min-bounded* (but does still have circuits for certain values of m) would seem to be much more difficult. In Section 4.7 we provide proofs that answer some of these questions for the $(1,4)-5\times m$ instance class.

We define another category of instance classes as *min-exist* if circuits exist only for some

[10] Or from n if n was even. Note that using $m = n+1$ as a starting point was done for convenience. For several instance classes, our proofs show that no circuits exist for certain values of $m > n$.

Table 4.2: Empirical results concerning the existence of circuits for various instance classes of the generalized knight's circuit problem.

Instance Class	Values of m for which Circuits found	No circuits exist
$(1,4) - 5 \times m$	$24, 32, 34, 38 - 60$	$6 - 22, 26 - 30$
$(1,4) - 7 \times m$	$16 - 30$	$8 - 14$
$(1,4) - 9 \times m$	$10 - 20$	
$(1,4) - 10 \times m$	$10 - 20$	
$(1,6) - 7 \times m$	$36, 48, 60, 72$	$8 - 34, 38 - 46$
$(1,6) - 9 \times m$	$24, 36, 38, 48$	$10 - 22, 26 - 34$
$(1,6) - 11 \times m$		$12 - 22$
$(1,8) - 9 \times m$	$48, 64$	$10 - 46, 50$
$(1,8) - 11 \times m$	32	$12 - 30, 34 - 46$
$(1,8) - 13 \times m$		$14 - 30$
$(1,8) - 15 \times m$		$16 - 30$
$(2,3) - 5 \times m$	$16, 20 - 40$	$6 - 14, 18$
$(2,3) - 9 \times m$	20	$10 - 18$
$(2,3) - 10 \times m$	$10, 11, 13 - 20$	12
$(2,5) - 7 \times m$	$20, 30, 34 - 40, 50$	$8 - 18, 22 - 28$
$(2,5) - 9 \times m$		$10 - 30$
$(2,5) - 10 \times m$		$10 - 16$
$(2,5) - 11 \times m$	$18, 20$	$12 - 16$
$(2,7) - 9 \times m$	56	$10 - 40$
$(2,7) - 11 \times m$		$12 - 40$
$(2,7) - 12 \times m$		$12 - 30$
$(2,7) - 13 \times m$		$14 - 26$
$(3,4) - 7 \times m$	$22, 30, 36 - 40, 46$	$8 - 20, 24 - 28, 32$
$(3,4) - 13 \times m$		$14 - 40$
$(4,5) - 9 \times m$	$46, 48$	$10 - 40$
$(4,5) - 17 \times m$		$18 - 50$

$m \geq m_e > n + 1$. The value m_e represents the smallest board length permitting a circuit for that instance class. Most instance classes we examined fall into this category, which is expected. The formulation of instance classes requires that $m \geq n$ ($m > n$ if n is odd). The condition $m_e > n+1$ basically means that additional constraints prevent circuits from existing for small m. These constraints are a function of the move parameters and board size. We conjecture that the minimum board length m_e will increase as the parameters A and B increase. Interestingly, as n increases for a fixed A and B, it seems easier for circuits to exist, thus implying a drop in m_e.

Note that *min-exist* instance classes with high values of m_e might also be *min-bounded*. One example of this is the $(2,3) - 9 \times m$ instance class. Tests were executed for $m = \{10 \ldots 20\}$. A circuit was found for the $m = 20$ case only (so $m_e = 20$). Is $m_n = 20$ for this instance class? Or is there a periodic occurrence of circuits? These open questions can only be answered by theoretical investigation, since empirical calculations eventually become intractable as m grows large.

Our third category of instance classes we call *periodic* if we observed a periodic occurrence of circuits (with respect to m). To be precise, we classify an instance class as *periodic* if circuits exist only for values of $m = pk + m_e$, where $k = \{0, 1, 2, \ldots\}$, $p =$ the period of occurrence and $m_e =$ the minimum board length required for a circuit to exist. In our initial experiments, it appeared that many instance classes were periodic since instances with $m \neq pk + m_e$ never completed (within a few days or so). We performed additional experiments that used a modified form of our backtrack algorithm. We added a 10 minute time limit on the length of the execution, and restarted the algorithm if the time limit was reached. This allowed us to obtain circuits for instances with $m \neq pk + m_e$. A deeper investigation of these results showed that the initial vertex selected by the algorithm (the choice was made randomly) has a major effect upon the performance of the algorithm. Some choices for initial vertices make the algorithm finish quickly, while many other choices for $m \neq pk + m_e$ make the algorithm never finish. So for these non-periodic values, the resulting graphs are in general hard to solve for our algorithm (but did have circuits), while for the periodic values the graphs are easy to solve.

After these additional experiments, the following instance classes remain which still seem to be *periodic*: $(1,6) - 7 \times m$ and $(1,8) - 9 \times m$. We suspect these instance classes are like the others and have harder-to-find solutions for values of $m \neq pk + m_e$. When examining all these periodic and apparently-periodic instance classes, we made one highly unexpected observation.

Table 4.3: Classification of instance classes.

Instance Class	m_n	m_e	p
$(1,4) - 5 \times m$	38	24	–
$(1,4) - 7 \times m$	16	16	–
$(1,4) - 9 \times m$	10	–	–
$(1,4) - 10 \times m$	10	–	–
$(1,6) - 7 \times m$	–	36	12
$(1,6) - 9 \times m$	–	24	–
$(1,8) - 9 \times m$	–	48	16
$(1,8) - 11 \times m$	?	32	?
$(2,3) - 5 \times m$	20	16	–
$(2,3) - 9 \times m$	?	20	?
$(2,3) - 10 \times m$	13	–	–
$(2,5) - 7 \times m$	34	20	–
$(2,5) - 11 \times m$	?	18	–
$(2,7) - 9 \times m$	?	56	?
$(3,4) - 7 \times m$	36	22	–

Observation 1 *For instance classes exhibiting periodic behaviour, it was found without exception that the period $p = 2B$.*

This seems to show that the high regularity found in the generalized knight's circuit problem has some very subtle effects upon when circuits appear for graphs, and how hard it is to find these circuits.

Table 4.3 summarizes the discussion on these three categories. The table lists instance classes and their values for the parameters m_n, m_e and p. The presence of a number for one of these parameters indicates that the instance class appears to be a member of the corresponding category (*min-bounded*, *min-exist* and *periodic*, respectively). An entry of "–" indicates that the particular instance class does not appear to be a member of the corresponding category, and an entry of "?" indicates uncertainty.

The various theorems presented in the previous sections never apply when $n = A + B$ (except for Theorem 15, which we assume does not apply in this discussion). For instance, Theorem 19 proves that no circuit can exist for an instance class with $n = A + B + 1$. However as Table 4.2 shows, circuits were found for all instance classes when $n = A + B$, and in addition seem to occur periodically for many of these classes. This leads us to conjecture that there is something in the basic structure of these problems that always allows circuits to exist when $n = A + B$.

A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8
E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8

Figure 4.8: A labelled 5×8 board. The labels correspond to the elements of the partition Π_k used for the proof.

4.7 Proofs Concerning the $(1, 4) - 5 \times m$ Instance Class

In this section, we consider just one instance class, $(1, 4) - 5 \times m$, one of the $(1, B) - B+1 \times m$ instance classes (with $B = 4$), for which we have the following empirical results:

Observation 2 *For the $(1, 4) - 5 \times m$ problem, no circuit exists for $m < 32$, $m \neq 24$.*

Observation 3 *For the $(1, 4) - 5 \times m$ problem, circuits exist for $m \geq 38$, up to $m = 60$.*

In this section we develop short proofs addressing these observations. In Section 4.7.1 we provide a proof addressing Observation 2 and in Section 4.7.2 we develop a proof that extends our results of Observation 3.

4.7.1 Circuit Non-Existence Proof

Our empirical results show that for the $(1, 4) - 5 \times m$ instance class no circuits exist for $m < 32, m \neq 24$. In this section we provide a short proof for a portion of these results. We prove that no circuit exists for $m = 8k + r, r = \{2, 4, 6\}, k < 3$. This section not only shows the use of the *partition proof technique* but also might provide some insight or hint into the periodic level of difficulty of the problem.

We start by using the *partition proof technique*. The partition Π_k is defined as follows. The 40 elements of the partition correspond to the squares of a 5×8 section of board. We tile the actual board ($5 \times m$) with this section k times, and use a portion of the section to tile the remainder (since $m = 8k + r$). Figure 4.8 shows the labels we use for each of the elements of the partition. Figure 4.9 displays the resulting contraction graph T_k (for $m > 8$).

When $r = \{2, 4, 6\}$, the size of the partition elements vary. Specifically, for a particular r, all elements A_x, B_x, C_x, D_x, E_x, $x \leq r$ are of size $k + 1$, while the other elements are of

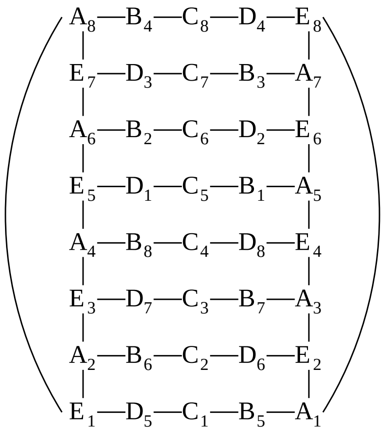

Figure 4.9: The contraction graph T_k based on the 40 element partition Π_k.

size k. The structure of the contraction graph combined with these different sized elements results in certain portions of each graph having very interesting properties that form the basis of our proof. This interesting subgraph G_i consists of a five vertex path in which the size of the element corresponding to each vertex alternates between k and $k+1$. Formally, we define the graph G_i to be a weighted graph of five vertices $v_1, \ldots, v_5$ in a path, with odd vertices of weight k and even vertices of weight $k+1$. (So $w(v_1) = k$, $w(v_2) = k+1$.)

Lemma 4 *Graph G_i is a subgraph of the contraction graph T_k for $m = 8k + r$, $k \geq 1$, $r = \{2, 4, 6\}$.*

Proof. Due to the definition of the partition Π_k, for a particular r all elements $\{A - E\}_x$, $x \leq r$ are of size $k+1$, while the other elements are of size k. So for $r = \{2, 4\}$ the subgraph A_6, B_2, C_6, D_2, E_6 of T_k corresponds to G_i. For $r = 6$ the subgraph A_8, B_4, C_8, D_4, E_8 of T_k corresponds to G_i. $\square$

Theorem 20 *No circuit exists for the $1, 4 - 5 \times m$ problem, $m = 8k + r$, $r = \{2, 4, 6\}$, $k < 3$.*

Proof. From Lemma 4 we know that the graph G_i is a subgraph of T_k for $r = \{2, 4, 6\}$ and $k \geq 1$. If we take v_1, v_3, v_5 from G_i as a cutset of size $3k$, then we get $2(k+1)$ components from v_4 and v_5 plus at least 1 component from the rest of the graph (since G_i is always smaller than T_k). Thus no cycle exists for $2(k+1) + 1 > 3k$. Rearranging, we get $k < 3$. When $k = 0$ ($m = r = \{2, 4, 6\}$), Lemma 4 does not apply. However for these small boards, by inspection at least one vertex in the middle row is of degree zero, which implies that no Hamiltonian cycle is possible. $\square$

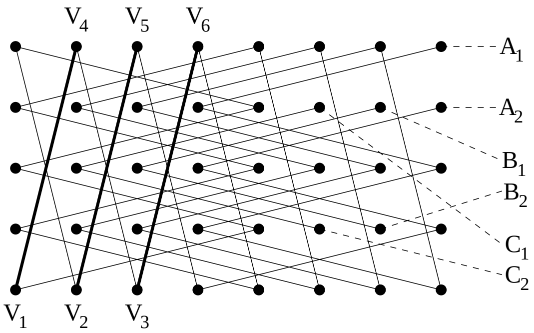

Figure 4.10: The 5×8 section of board used to extend existing cycles. Note that 3 paths cover this board, with endpoints (A_1, A_2), (B_1, B_2) and (C_1, C_2). The labelled vertices and emphasized edges on the left side are necessary to allow additional extensions.

4.7.2 Circuit Existence Proof

In this section we present an inductive proof that shows that the $(1,4) - 5 \times m$ instance class has a circuit for all $m \geq 38$ (m even). The inductive base cases for our proof are solutions to the problem for various values of m that were computer generated using our backtrack algorithm. A construction is presented which allows these circuits to be extended in length by 8 squares.

Lemma 5 *A circuit for a $(1,4) - 5 \times m$ instance with edges (V_1, V_4), (V_3, V_6) and (V_2, V_5) (where $V_1, \ldots, V_6$ are defined as shown in Figure 4.10* [11]*) can be expanded to a circuit that solves a $(1,4) - 5 \times m + 8$ instance.*

Proof. The existing circuit contains the sequence of vertices $\{\ldots, V_1, V_4, V_3, V_6, \ldots, V_5, V_2, \ldots\}$. The 5×8 board section used to extend the circuit consists of three paths $((A_1, \ldots, A_2)$; $(B_1, \ldots, B_2)$; $(C_1, \ldots, C_2)$) which must be included in the existing circuit. Figure 4.11 shows the construction necessary to accomplish this. The three paths are linked into the existing circuit to make the resulting sequence: $\{\ldots, V_1, A_1, \ldots, A_2, V_6, \ldots V_5, B_1, \ldots, B_2, V_3, V_4, C_1, \ldots, C_2, V_2, \ldots\}$. Figure 4.12 illustrates this process. This modified sequence preserves the existing circuit while including the additional board section, thus producing a circuit for the $(1,4) - 5 \times m + 8$ instance. □

Note that the newly extended circuit has the 3 edges necessary for the construction since they are a part of the 5×8 board section. Thus any circuit with those 3 edges can be extended by 8 squares an unlimited number of times.

Theorem 21 *A solution exists for every $(1,4) - 5 \times m$ instance for $m \geq 38$ (m even).*

[11] The six vertices have the following board positions, assuming that the top left corner is $(1,1)$. $V_1 = (5,1), V_2 = (5,2), V_3 = (5,3), V_4 = (1,2), V_5 = (1,3), V_6 = (1,4)$.

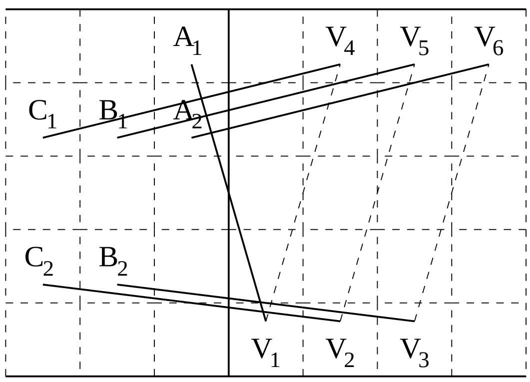

Figure 4.11: The construction to connect the 5×8 board section with an existing cycle that has vertices $V_1, V_2, \ldots, V_6$ properly connected. Dashed lines indicate edges that are removed from the original cycle.

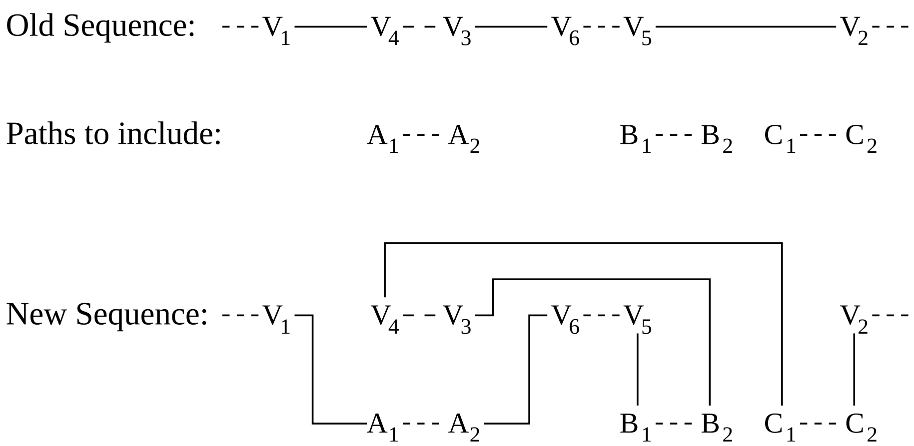

Figure 4.12: The original sequence and the modified sequence including the 5×8 board section.

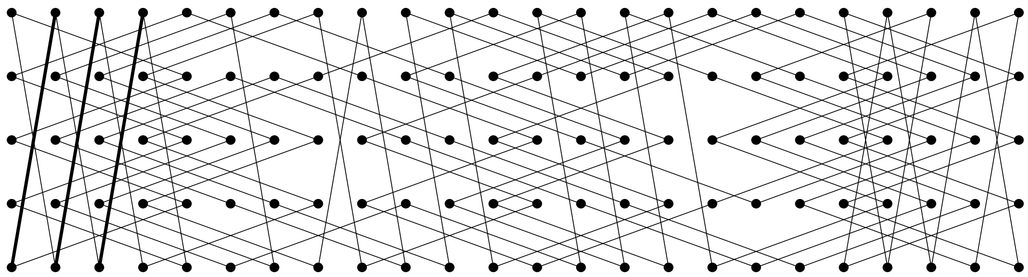

Figure 4.13: A solution to the $(1, 4) - 5 \times 24$ instance.

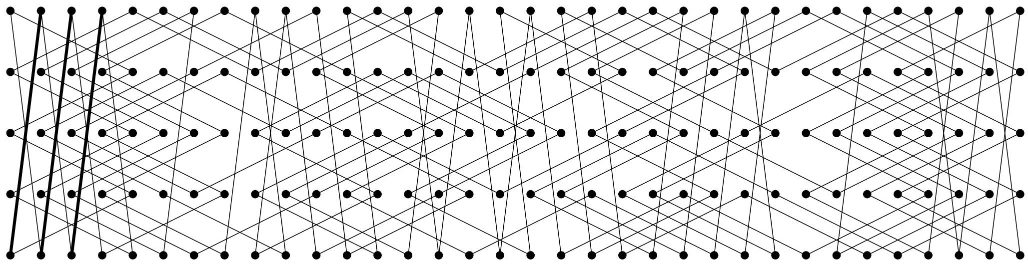

Figure 4.14: A solution to the $(1, 4) - 5 \times 34$ instance.

Proof. The proof is done by induction. We have 4 solutions which will form our base cases for $m = 24, m = 34, m = 38$ and $m = 44$. These circuits are displayed respectively in Figures 4.13, 4.14, 4.15 and 4.16. Note that the edges necessary for the construction of Lemma 5 are emphasized in each figure. Lemma 5 proves the inductive step that if a circuit exists for a certain value of $m = 8K + r$ with $r = \{0, 2, 4, 6\}$, then a circuit exists for $m = 8(K + 1) + r$. Our base cases are defined for $K = 5$ and $r = \{0, 2, 4, 6\}$ as follows: $m = 40(24 + 8 + 8), 42(34 + 8), 44$ and $46(38 + 8)$. Note that we make use of Lemma 5 to extend some of our circuits to obtain these base cases. Since the circuits produced by the lemma can be extended in turn, we have solutions for $m = 8(K) + r, K \geq 5, r = \{0, 2, 4, 6\}$. Thus we have solutions for $m \geq 40$. Including the solution we have for the 5×38 instance means we have solutions for $m \geq 38$ $\square$

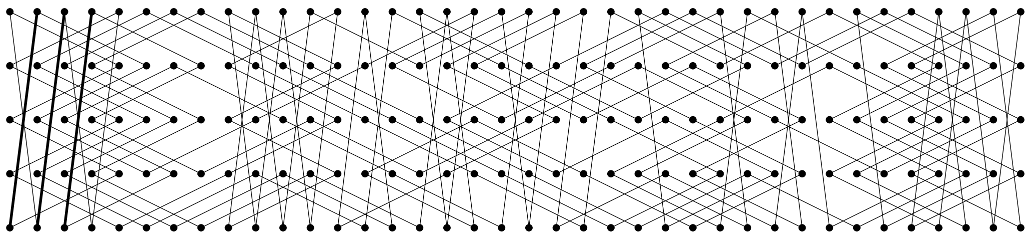

Figure 4.15: A solution to the $(1, 4) - 5 \times 38$ instance.

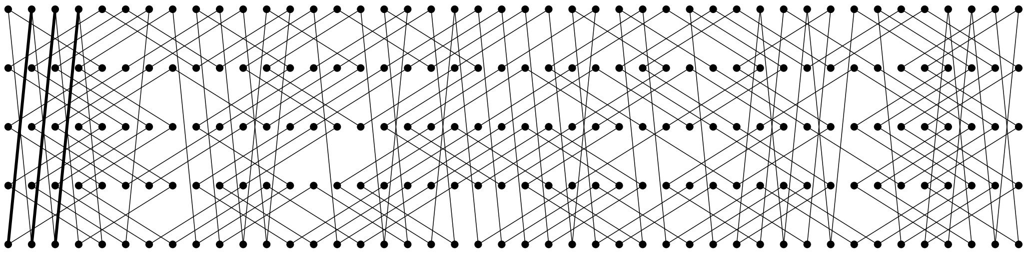

Figure 4.16: A solution to the $(1,4) - 5 \times 44$ instance.

4.8 Conclusions

In this chapter we have proposed a new generalization to the standard knight's tour problem: the generalized knight's circuit problem. Our main focus has been on determining which instances of this problem are Hamiltonian (or which instances are not Hamiltonian). Given an arbitrary instance $(A, B) - n \times m$ of this problem, we have produced various theorems showing for which values of A, B, n, m circuits cannot exist. We developed the *partition proof technique* to aid us in our development of some of these theorems. Besides our theoretical work in determining the Hamiltonicity of this problem, we also undertook an empirical investigation. We classified instances into instance classes (A, B, n fixed, m allowed to vary) and found that the Hamiltonicity of an instance class could be classified according to three classifications: *min-bounded*, *min-exist* and *periodic*. We made several observations and conjectures that suggest that more future research is needed. In particular, we observed a periodicity in many problems, either in the problem difficulty (periodic easy problems) or in terms of the existence of circuits. These periods were always equal to $2B$. We conjectured that all instance classes with $n = A + B$ (for which Theorem 15 does not apply) have circuits for one or more values of m.

Our final results involved proofs for the $(1,4) - 5 \times m$ instance class. In particular we proved that circuits exist for all even $m \geq 38$. One possibility for future research is to obtain more general proofs for the existence of circuits involving different instance classes (ideally involving arbitrary instance classes). However, such proofs would most likely require new proof techniques different from the ones employed in this chapter.

Our empirical results concerning the periodicity of certain instance classes show how different algorithms can change the difficulty of the same graphs. Our original backtrack without the time limit was unable to produce results within a few days for most of these graphs while our time-limited backtrack with random restart was able to produce results within an hour or two at most.

Chapter 5

Hard Hamiltonian Cycle Graphs

5.1 Introduction

The Hamiltonian cycle problem can be considered a subset of the travelling salesmen problem (TSP) for which all edge weights are equal. While both are NP-C problems, the TSP has received much more attention as a problem worth solving, while Hamiltonicity has been considered more a property of use to mathematicians and graph theorists. In addition, the Hamiltonian cycle problem has been shown to be solvable in polynomial time *whp* for various random graph models. Thus, the perception may exist that the Hamiltonian cycle problem is easy. In this chapter our goal is to show that the Hamiltonian cycle problem can not be considered solved (or even easy): hard graphs and hard graph sets do exist. Furthermore, finding these hard instances can only lead to improvements in our Hamiltonian cycle algorithms. Experiments using hard graph sets can help in determining which techniques and heuristics are superior in which circumstances. As we discuss below, hard graphs typically occur when our algorithms fail to perform well, which can often lead to insight into the design of better algorithms, or at the very least knowledge of which graph properties give our algorithms difficulties. In the literature, research into hard regions and hard graphs has seemed to neglect the important role the algorithm plays in determining hardness. The second goal of this chapter is to explore and characterize this interaction between graph hardness and the algorithms being used. We demonstrate repeatedly that a particular graph (or set of graphs) can be easy for one algorithm but difficult for another.

First we need to define what we mean by hard graphs. In general, we refer to two different circumstances. In the first, we have a Hamiltonian graph for which our algorithm has difficulty finding a circuit. In the second, we have a graph set for which our algorithm, in general, has difficulty determining if a graph in the set is Hamiltonian or not. In either case, we evaluate how hard a graph is by how long it takes to find a Hamiltonian cycle or show that the graph is non-Hamiltonian (for backtrack algorithms). This leads us to a comparative definition of hardness on Hamiltonian graphs: Given a Hamiltonian cycle algorithm A, Hamiltonian graphs G_1 and G_2 of equal order (so $|V(G_1)| = |V(G_2)|$) and expected time to find a solution on graph G equal to $E_A(G)$ then graph G_1 is harder than graph G_2 if $E_A(G_1) > E_A(G_2)$. We can extend this definition to non-Hamiltonian graphs if we assume a backtrack algorithm is being used.

From this comparative definition, we can try to form an objective definition of hardness. Unfortunately, this is not easy. We define a graph G to be objectively hard for an algorithm A if it takes an intractable period of time for A to find a Hamiltonian cycle (or to prove that one does not exist, if A is a backtrack algorithm). The flaw with this definition is that the determination of an intractable period of time is subjective (do we need a solution in hours, days or weeks?) and varies according to a variety of factors (efficiency of algorithm implementation, computer hardware, etc.).

Our definition of hardness was made relative to a single algorithm. This was done because hardness is a function of both the type of graph being solved and the type of algorithm being used. In other words, the hardness of a graph can depend upon the algorithm being used to solve the problem. A simple example provides evidence for this claim. We consider two graph sets. The first is the set of complete graphs (every vertex is connected to every other vertex) $K(n)$, for which every vertex ordering is a Hamiltonian cycle. The second is the set of cycle graphs (the graph is a Hamiltonian cycle; no other edges exist except those forming the circuit) $C(n)$. We now present a new Hamiltonian cycle algorithm with very bad performance: the lottery algorithm. At each stage of the search, it generates a random vertex ordering, and then tests it to see if it is a Hamiltonian cycle or not. For a graph G with a number of Hamiltonian cycles equal to $|HC(G)|$, the probability of success after a single iteration is $|HC(G)|/n!$. In terms of our definition of hardness, the graph set $K(n)$ is easy for the lottery algorithm, while the graph set $C(n)$ is hard. Most other graphs will also be hard for the lottery algorithm. However, most familiar Hamiltonian cycle algorithms (such as the ones we surveyed in Chapter 2) will find both the $K(n)$ and $C(n)$ graph sets easy. While this is clearly an extreme example, it illustrates how one graph set can be easy (or hard) for two algorithms while another graph set is easy for the one, but hard for the other. Note that the more alike two algorithms A and B are, the more likely graphs that are hard for algorithm A will also be hard for algorithm B. Note as well that the hardness of some graphs sets is affected more by the type of algorithm being used than other graph sets.

Note that our definition of comparative hardness requires that the graphs have the same number of vertices. This is not necessary or accurate from an operational viewpoint: larger graphs tend to require more time to solve, and thus will tend to be harder. However, we are most interested in discovering hard graphs that will provide insight into the Hamiltonian cycle problem and the operation of our algorithms. Since most problems become harder (require more time to solve) as the graph size is increased, this relationship is of limited interest to us.

Having defined hardness and examined some of the factors that affect it, we now consider the types of hard graphs we will explore. Due to the nature of the Hamiltonian cycle problem, it is not trivial to come up with hard sets of graphs. Random graphs of high degree tend to be easy since many different Hamiltonian cycles exist, and the algorithms seldom reach dead-ends in their search due to the larger number of choices. Low degree graphs are also easy, particularly for backtrack algorithms. First, if the graph is not biconnected, it is easily determined to have no Hamiltonian cycles. Secondly, search pruning performed by backtrack Hamiltonian cycle algorithms is efficient at reducing the search space. The algorithm can often quickly determine when a particular choice leads to a dead-end. However, the study of phase transitions can aid us in our search. We examine phase transitions in Section 5.2 and review the literature on phase transitions and the Hamiltonian cycle problem.

Before presenting the experimental results in the remainder of the chapter, we first discuss the experimental methodology we employ and describe the algorithms we use in Section 5.3. Since most work on hard graphs for various NP-C problems including the Hamiltonian cycle problem has involved standard random graph classes such as $G_{n,m}$, we examine the existence of phase transitions and hard graphs for this class in Section 5.4. In Section 5.5 we examine a low degree random graph set we call Degreebound graphs. We demonstrate the existence of a phase transition and show that while these graphs are hard for heuristic algorithms, they are easy for backtrack algorithms. In Section 5.6 we examine generalized knight's circuit graphs. In Section 5.7 we construct two different types of graphs to take advantage of limitations in the various Hamiltonian cycle algorithms. We provide theoretical and experimental evidence that these graphs are hard for some algorithms but not for others according to the properties given to these graphs.

5.2 Phase Transitions

Problems that are NP-C are commonly perceived as hard, yet many instances of various NP-C problems are easy. This is due to the fact that the NP-C complexity class characterizes worst-case behaviour. This raises the question of how to identify the subset of a particular problem which is hard. Phase transitions are one means of accomplishing this. In general, many NP-C problems can be characterized by a 'constraint' parameter which measures how constrained an instance is. Evaluation of a problem using this constraint parameter typically divides instances into two classes: those that are solvable, and those that are unsolvable. Between these two classes lies the phase transition. In this region, the problem is considered critically constrained, and instances are usually hard. The standard explanation for this behaviour is as follows. When the problem is highly constrained, it is easily determined that no solution exists. When the constraints are mostly removed, then a solution is easily found. However, near the critical constraint region there tends to be many 'almost' solutions, and very few (if any) true solutions. Thus, any algorithm must perform much work searching the greater number of near solutions before finding an actual solution. Note that the constraint parameter and hence the phase transition region are defined independent of the problem size. [9, 14]

While most researchers have studied phase transitions for problems such as graph coloring or satisfiability, some have looked at the Hamiltonian cycle problem. Cheeseman et al. [9] and Frank and Martel [14] both examined phase transitions on random graphs for the Hamiltonian cycle problem. The obvious constraint parameter is the average degree (or average connectivity) of the graph. As the degree increases, the graph becomes less constrained: it becomes easier both for a Hamiltonian cycle to exist and for an algorithm to find one. Both sets of researchers examined how Hamiltonicity changes with respect to the average degree. Frank and Martel experimentally verified that the phase transition for Hamiltonicity is very close to the phase transition for biconnectivity, which occurs when the average degree is $\log n$ (or $E = n \log n/2$) [1]. Cheeseman et al. experimentally confirmed theoretical predictions by Komlós and Szemerédi [21] that the phase transition (for the Hamiltonian cycle problem) occurs when the average degree is $\log n + \log \log n$. Both papers also provided empirical evidence that the time required by their backtrack algorithms increased in the region of the phase transition. While limited details (if any) were provided on these algorithms, from the comments made in the papers, it is likely that no pruning

[1] Note that the average degree equals $2E/n$.

was performed by the algorithms. Perhaps for this reason their reported results were only for graphs of up to 24 vertices.

The limited graph size used by these researchers and the seemingly crude algorithms employed raises questions about the accuracy of their results. Therefore in Section 5.4 we examine random graphs: firstly to verify the presence of a phase transition and secondly to determine if graphs in the phase transition region really are hard.

5.3 Experimental Methodology

In this section, we describe our experimental methodology and the algorithms that we will be using throughout this chapter. We use two different algorithms: one version of a backtrack Hamiltonian cycle algorithm, and one version of a heuristic Hamiltonian cycle algorithm (see Chapter 3 for detailed descriptions of the following techniques and heuristics). Our backtrack algorithm uses the singlepath search method, full initial pruning (graph reduction and global checking) and local pruning using graph reduction. Vertex selection is done using the low degree first heuristic, and initial vertex selection is done by selecting a random vertex. We also use a time limit of 10 minutes (600 seconds) for our backtrack algorithm. If that time period is exceeded, the algorithm stops executing and returns an incomplete result for the current graph. This time limit is needed to avoid the case of the rare graph that takes hours or days. Note that the time limit of 10 minutes is at least two orders of magnitude greater than the typical running time, so the limit is rarely used.

The heuristic Hamiltonian cycle algorithm we use has the following features. It uses the rotational transformation and cycle extension techniques, the vertexonce search termination method, full initial pruning (graph reduction and global checking) and the singlepath search method. Vertex selection is done using the vertex selection algorithm without the low degree first heuristic and without the NPN heuristics. The algorithm executes n times, trying each vertex as the initial vertex, and stops when a Hamiltonian cycle is found. While most heuristic Hamiltonian cycle algorithms cannot determine that a graph has no Hamiltonian cycle, ours can because of the initial pruning being used. However, since we are interested in the ability of the heuristic algorithm to find Hamiltonian cycles, we will not test the heuristic algorithm on non-Hamiltonian graphs. For random graph classes where Hamiltonicity is not guaranteed, we generate Hamiltonian graphs as follows. We generate a graph and run our backtrack algorithm on it. If the graph is Hamiltonian, we keep the graph for our heuristic algorithm. Otherwise we repeat the process.

Heuristic Hamiltonian cycle algorithms also cannot guarantee to find a Hamiltonian cycle in an arbitrary graph. Typically, heuristic algorithms will only succeed a certain percentage of the time. Furthermore, since most heuristic algorithms are fast (compared to backtrack), they can be run multiple times. These factors affect our testing of heuristic algorithms. Instead of just reporting the average time required, we compute the total expected time required for our heuristic algorithm to solve a particular graph (or set of graphs). For each graph or graph set, we run a number of trials. We compute the percentage of trials in which a solution was found (P_S), the average time spent on trials where a solution was found (T_S), and the average time spent on trials where the algorithm failed (T_F). The total expected time (T_T) is calculated as

$$T_T = T_S + \frac{(1 - P_S)T_F}{P_S}$$

This formula accounts for the time required to execute the expected number of unsuccessful attempts (at finding a Hamiltonian cycle) along with the time required for the successful attempt.

Note that our backtrack algorithm with the 10 minute time limit can function like a heuristic algorithm on very hard graphs. On such graphs we can calculate the expected time for our backtrack algorithm in the same way as for our heuristic algorithm.

Experiments were performed on a Sun SPARCstation 20 model 50 (50 MHz) (SPECint92 = 76.9 and SPECfp92 = 80.1). Algorithms were not optimized for performance.

5.4 $G_{n,m}$ Random Graphs

In this section we examine $G_{n,m}$, a standard random graph model, to determine the difficulty of these graphs for our backtrack Hamiltonian cycle algorithm. One of our goals is to extend the research of Cheeseman et al. [9] and Frank and Martel [14], whose experiments were performed on small graph sizes (≤ 24 vertices) using primitive algorithms.

Our experiments will be performed on graphs of 100 to 500 vertices. We use the $G_{n,m}$ random graph model, with $m = \overline{d}n/2$. From previous work both theoretical [21] and experimental [9], we expect the phase transition to occur when $\overline{d} = \log n + \log\log n$. Thus we specify the constraint parameter (or degree parameter) $k = \overline{d}/(\log n + \log\log n)$. We use the degree parameter k to measure where a graph (or set of graphs) is with respect to the phase transition independent of its size. (Note that log is the natural logarithm.) Table 5.1 shows the values of the mean degree for different graph sizes with $k = 1$.

In our experiments we use our backtrack algorithm as specified in Section 5.3. We generate 100 graphs for each data point, execute our algorithm once on each graph and

Table 5.1: Mean degree of $G_{n,m}$ graphs for k (degree parameter) = 1.

# of vertices	Mean Degree
100	6.132
200	6.966
300	7.445
400	7.782
500	8.042

average the results.

We first verify that a phase transition exists. For $G_{n,m}$ graphs of 100 to 500 vertices with the degree parameter k ranging from $0.5 - 2.0$ we determine the percentage of the graphs that are Hamiltonian, the percentage of the graphs that are biconnected and the percentage of the graphs for which $\delta(G) \geq 2$. From Cheeseman et al. [9] we expect the phase transition for biconnectivity to be very similar to the phase transition for Hamiltonicity, and from Bollobás [5] and Komlós and Szemerédi [21] we expect the phase transition for minimum degree greater than 1 to be almost identical to the phase transition for Hamiltonicity. Our experimental results matched these expectations very closely. Out of the 10,000 trials, only two non-Hamiltonian graphs were generated that were biconnected and had a minimum degree ≥ 2. (The first graph had 100 vertices with $k = 0.90$ and the second had 400 vertices with $k = 1.05$.) The remainder of the non-Hamiltonian graphs had one or more degree 1 (or 0) vertices and were not biconnected. Our results for the Hamiltonian phase transition are presented in Table 5.2.

We define the center of the phase transition as the point at which 50% of the graphs are Hamiltonian. These results show that the center of the phase transition occurs when the degree parameter is in the range of $1.05 - 1.15$. While there is some variation over the different graph sizes, the phase transition does not tend to become steeper (increase in slope) or shift in location as the graph size increases. This is due to the fact that the constraint parameter k automatically accounts for the graph size. This suggests that using the degree parameter as the constraint parameter is superior to using the mean degree as was done by Cheeseman et al. in [9], which made comparisons across graphs of different sizes difficult. Our degree parameter is similar to the $E/(n \log n)$ constraint parameter used by Frank and Martel [14].

Our next step is to investigate how our backtrack algorithm performs on the $G_{n,m}$ graphs as the degree parameter k varies across the phase transition. We measured the time required by our backtrack algorithm in the experiments detailed above, and present our

Table 5.2: Percentage of Hamiltonian graphs of $G_{n,m}$ graphs as a function of graph size and degree parameter.

Degree Parameter	Number of Vertices 100	200	300	400	500
0.50	0	0	0	0	0
0.60	0	0	0	0	0
0.70	0	0	0	0	0
0.80	1	0	1	0	0
0.90	11	9	8	6	4
0.95	16	17	12	15	7
1.00	24	30	26	24	25
1.05	40	33	41	40	41
1.10	40	59	53	48	51
1.15	49	56	62	62	61
1.20	57	71	73	64	72
1.25	73	72	77	81	76
1.30	81	82	90	83	93
1.40	88	89	90	91	90
1.50	95	98	97	95	94
1.60	90	97	100	99	97
1.70	99	99	98	98	98
1.80	100	100	100	100	99
1.90	100	99	100	100	100
2.00	100	99	100	100	100

Table 5.3: Time in seconds required by our backtrack algorithm on Hamiltonian $G_{n,m}$ graphs.

Degree	Number of Vertices				
Parameter	100	200	300	400	500
0.50	–	–	–	–	–
0.60	–	–	–	–	–
0.70	–	–	–	–	–
0.80	0.1	–	0.5	–	–
0.90	0.1	0.3	0.5	1.0	1.4
0.95	0.1	0.3	0.5	1.0	1.5
1.00	0.1	0.2	0.6	1.0	1.5
1.05	0.1	0.3	0.6	1.0	1.6
1.10	0.1	0.3	0.6	1.0	1.6
1.15	0.1	0.3	0.6	1.0	1.6
1.20	0.1	0.3	0.6	1.1	1.6
1.25	0.1	0.3	0.6	1.1	1.7
1.30	0.1	0.3	0.7	1.1	1.9
1.40	0.1	0.3	0.6	1.1	1.8
1.50	0.1	0.3	0.7	1.2	1.9
1.60	0.1	0.3	0.7	1.2	1.9
1.70	0.1	0.3	0.7	1.3	2.1
1.80	0.1	0.3	0.7	1.3	2.1
1.90	0.1	0.4	0.8	1.4	2.2
2.00	0.1	0.4	0.8	1.4	2.2

results in Table 5.3. Since almost all the non-Hamiltonian graphs had one or more vertices of degree less than 2, our backtrack algorithm quickly detected all these non-Hamiltonian graphs. Thus, our times are averaged over the Hamiltonian graphs only.

Our results clearly show no increase in difficulty in the phase transition region. As the degree parameter (and thus the mean degree) increases in size, there is a slight increase in the time required by our backtrack algorithm. This seems to be due solely to the increased computational cost of handling higher degree vertices, since the algorithm performs operations like the low degree first heuristic which requires processing all the neighbours of a vertex. Additional evidence that supports this view is that the number of nodes searched by our algorithm does not increase as the degree parameter increases. The time required by our backtrack algorithm increases as the graph size increases, as one would expect. However, the rate of increase seems to be quadratic (n^2) rather than exponential. These results suggest that $G_{n,m}$ graphs are easy for our backtrack algorithm.

We performed additional experiments using 750 and 1000 vertex $G_{n,m}$ graphs and a preliminary analysis of these results indicate that the prior observations still hold for these larger random graphs.

Both Cheeseman et al. [9] and Frank and Martel [14] found that the computational work performed by their backtrack algorithm (number of nodes searched) peaked to the right of the center of the phase transition and fell off at the extremes. Cheeseman et al. explained this increase in difficulty by stating that "On the border [between the regions of low and high connectivity] there are many almost Hamiltonian cycles that are quite different from each other ... and these numerous local minima make it hard to find a Hamiltonian cycle (if there is one). Any search procedure based on local information will have the same difficulty." [9]. Our results clearly contradict this statement. While near the center of the phase transition many local minima may exist (further research is necessary to determine this), the search pruning used by our backtrack algorithm is too efficient to become trapped by these minima. In the majority of the time our algorithm determines immediately (using pruning) when a wrong edge leading to a dead-end has been selected, and thus avoids ever following such an edge.

This illustrates the large effect a particular algorithm can have on determining if a graph (or set of graphs) is hard. Researchers such as Cheeseman et al. and Frank and Martel found that these $G_{n,m}$ graphs were hard in the phase transition region because they were using poor Hamiltonian cycle algorithms. Our algorithm, by using search pruning and other improvements, is much more efficient and thus finds these graphs to be easy.

Of the 5516 Hamiltonian graphs we tested, only 1 graph took an order of magnitude more time to solve than the average. (The remainder of the graphs all took less than 4 seconds to solve.) One 500 vertex random graph with a degree parameter of 1.30 took 14.3 seconds to solve. When further trials were performed on this graph, we found that it took 1.7 seconds to solve on average and took a maximum of 1.8 seconds to solve over 20 trials. This suggests that the 14.3 second result was a low probability event resulting from a poor choice for the initial vertex, and that this particular graph is not actually hard.

The results in this section clearly indicate that random graphs generated under the $G_{n,m}$ graph model are not hard in any way for our backtrack algorithm although there is a clear phase transition.

5.5 Degreebound Graphs

In this section we examine regular and almost regular graphs in which all vertices are of low degree. We look for phase transitions on these graphs, and then examine the difficulty of these graphs (in the region of the phase transition) for backtrack and heuristic Hamiltonian cycle algorithms.

We specify a new random graph model $G_n(d_2 = p_2, d_3 = p_3, \ldots)$ for which n is the number of vertices and $d_i = p_i$ is the percentage of vertices of degree i. Note that the percentages must sum to 100% and that the sum of the vertex degrees must be even. As an example $G_n(d_3 = 100\%)$ represents the set of 3-regular graphs, and $G_{100}(d_2 = 50\%, d_3 = 50\%)$ represents the set of graphs of 100 vertices in which 50 are of degree 2 and 50 are of degree 3. We refer to a graph generated under this model as a Degreebound graph. Note that the average degree of a Degreebound graph is easily calculated as $\sum_i d_i p_i$. We only consider graphs which have percentages defined for $i \leq 4$. Formally, we should define $G_n(d_2 = p_2, d_3 = p_3, \ldots)$ as a uniform distribution over the defined set of graphs. Unfortunately we cannot prove the uniformity of the distribution of the algorithm we use to generate Degreebound graphs.

We start our investigation in Section 5.5.1 by providing evidence that a phase transition exists for Degreebound graphs and by exploring how the transition region changes as the graph size increases. In Section 5.5.2 we present results that show that Degreebound graphs throughout the phase transition are easy for backtrack Hamiltonian cycle algorithms. In Section 5.5.3 we show that the same graphs are hard for heuristic Hamiltonian cycle algorithms and that the hardness region corresponds to the region of the phase transition.

In this section we restrict ourselves to Degreebound graphs of degree 2 and 3 vertices only. Such graphs can be specified solely by their mean degree. (The mean degree of such graphs equals 3 minus the percentage of degree 2 vertices.) One area of future work is to investigate how the phase transition region changes when changing the distribution of vertices (by allowing degree 4 vertices) while keeping constant the mean degree.

For our experiments we use our backtrack algorithm as specified in Section 5.3 except that initial vertex selection is done by selecting a random vertex of maximum degree (so a random degree 3 vertex is selected). We generate 100 graphs for each data point, execute our algorithm once on each graph, and average the results. In experiments involving our heuristic algorithm (as specified in Section 5.3), we generate 25 graphs and run the algorithm 25 times per graph for a total of 625 trials.

5.5.1 Phase Transitions on Degreebound Graphs

To perform our search for a phase transition, we use our backtrack algorithm to calculate the percentage of graphs which are Hamiltonian for Degreebound graphs of different mean degrees with 100 to 500 vertices. Figure 5.1 illustrates our results. If a phase transition

Table 5.4: Location of the center of the phase transition for different graph sizes.

# of vertices	mean degree of center
100	2.78
200	2.81
300	2.83
400	2.84
500	2.85

exists, we would expect the percentage of Hamiltonian graphs to initially be 0%, and then sharply rise until it has reached 100%. We do observe this behaviour, although there is some question as to whether the slope of the curve is steep enough. Another property of phase transitions is that as the parameter (in this case the graph size) increases, the slope of the transition should become steeper, which means that the transition should become more abrupt. Figure 5.1 clearly shows this increase in steepness as the graph size increases.

Also of interest is the 50% point for each graph size (the point at which 50% of the graphs are Hamiltonian). Table 5.4 lists the estimated mean degree corresponding to the 50% point for each graph size. Graphs that were incomplete (algorithm reached time limit) were not included in percentage calculations because we do not know if they were Hamiltonian or not. Of the 10,500 trials performed there were a total of 38 incomplete results: 2 for 200 vertex graphs, 10 for 300 vertex graphs, 11 for 400 vertex graphs and 15 for 500 vertex graphs. We define these graphs as *ultrahard* because they take at least an order of magnitude longer to solve than the average graph, and we discuss these graphs more in Section 5.5.2. Table 5.4 shows that as the graph size increases the 50% point increases. However, the 50% point seems to be approaching an upper limit since the increases get smaller as the vertex size gets larger. This upper limit seems to lie in the range of 2.86 – 2.89. The existence of this upper limit is more evidence for the existence of a phase transition, since the phase transition (for sufficiently large parameter values) must occur at a single point. We do not have a good argument for why the 50% point shifts upward as the graph size increases. We suspect that as the graph grows larger, the increased number of vertices (and edges) allows for a greater variety of graph structures that produce non-Hamiltonian graphs.

5.5.2 Hardness of Degreebound Graphs for Backtrack Algorithms

As we have discussed above, the main use of phase transitions is to help identify hard regions. According to the literature [9, 14], we should expect the critical region of the phase transition to produce the hardest graphs. However, when we examine the time required

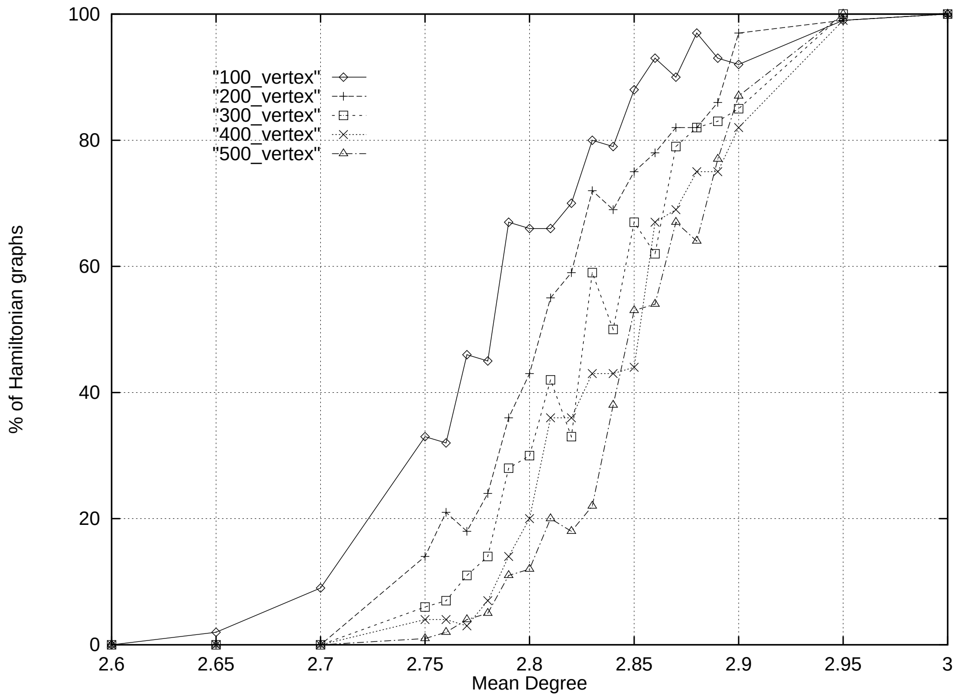

Figure 5.1: % of Hamiltonian graphs as a function of graph size and mean degree for Degreebound graphs.

by our backtrack Hamiltonian cycle algorithm on Degreebound graphs, we do not find this to be the case. Table 5.5 shows the average time (and standard deviation) required by the backtrack algorithm on 200 vertex Degreebound graphs of varying mean degree. The total time on all graphs is shown along with the average time required to solve the Hamiltonian graphs, and the average time required to solve the non-Hamiltonian graphs. Times for incomplete trials (ultrahard graphs) are not included in the Hamiltonian and non-Hamiltonian time averages, but are included in the column for average total time to give a lower bounds for this value. Out of the 2100 trials, only 2 graphs were incomplete (1 at 2.86 and 1 at 2.87).

The times for mean degrees of 2.85 and 2.95 seem to indicate an increase in difficulty, but an examination of the data indicates that the increase in time is due solely to four ultrahard graphs requiring 66.7 and 89.0 seconds for a mean degree of 2.85 and 33.4 and 85.3 seconds for a mean degree of 2.95. When these ultrahard graphs are removed, the mean time on Hamiltonian graphs (and the mean total time) drops back to the same range as the other mean degrees.

The results in Table 5.5 show several things. First, that there is no significant increase in difficulty (time required) for the backtrack algorithm even though a phase transition

Table 5.5: Time in seconds required by backtrack algorithm on 200 vertex Degreebound graphs.

mean	time on Ham.		time on non-Ham.		total time	
degree	mean	stddev	mean	stddev	mean	stddev
2.60	–	–	0.0	0.0	0.0	0.0
2.65	–	–	0.0	0.0	0.0	0.0
2.70	–	–	0.0	0.0	0.0	0.0
2.75	0.2	0.3	0.0	0.0	0.1	0.1
2.76	0.1	0.1	0.0	0.0	0.0	0.1
2.77	0.1	0.1	0.0	0.0	0.0	0.1
2.78	0.1	0.1	0.0	0.0	0.1	0.1
2.79	0.1	0.1	0.1	0.2	0.1	0.1
2.80	0.3	0.7	0.0	0.0	0.1	0.5
2.81	0.1	0.1	0.0	0.0	0.1	0.1
2.82	0.2	0.4	0.0	0.0	0.1	0.3
2.83	0.1	0.1	0.0	0.0	0.1	0.1
2.84	0.2	0.4	0.0	0.0	0.1	0.3
2.85	2.3	12.6	0.0	0.0	1.7	11.0
2.86	0.2	0.4	0.0	0.0	> 6.2	–
2.87	0.3	0.1	0.0	0.0	> 6.2	–
2.88	0.2	0.1	0.0	0.0	0.1	0.1
2.89	0.2	0.5	0.0	0.0	0.2	0.4
2.90	0.2	0.3	0.0	0.0	0.2	0.3
2.95	1.4	9.1	0.0	0.0	1.4	9.1
3.00	0.2	0.4	–	–	0.2	0.4

exists. Second, the backtrack algorithm can almost always immediately identify the non-Hamiltonian graphs. Out of the 2100 graphs examined, of which 990 were non-Hamiltonian, only 11 non-Hamiltonian graphs required searching by the backtrack algorithm and the maximum time required to find a non-Hamiltonian graph was 1.3 seconds. All other graphs could be identified as non-Hamiltonian by the initial pruning (graph reduction and graph checking). This would seem to provide an explanation why no increase in difficulty was observed in the critical region. As the critical region is approached, the number of almost-Hamiltonian graphs increases, and the theory is that the number of almost-Hamiltonian cycles to search increases as well, thus increasing the work required of the algorithm. However, the initial pruning of our backtrack algorithm is too efficient at identifying non-Hamiltonian graphs and thus has no problem with the critical region. When searching Hamiltonian graphs, the prior expectation was that in the critical region the algorithm has many almost-Hamiltonian cycles to search before finding one of the few cycles. But our algorithm quickly identifies when it has made a wrong decision leading to a dead-end, since such wrong decisions usually produce a non-Hamiltonian graph after graph reduction occurs. Thus almost no increase in difficulty occurs anywhere along the phase transition.

The accuracy of the results in Table 5.5 can be questioned because of the very short time intervals (and because of the relatively large standard deviations). We therefore perform a similar experiment for Degreebound graphs of 500 vertices. The results are shown in Table 5.6. Times for incomplete trials (ultrahard graphs) are not included in the Hamiltonian and non-Hamiltonian time averages, but are included in column for average total time to give a lower bounds for this value. Out of the 2100 trials, only 15 graphs were incomplete (1 at 2.79, 4 at 2.83, 3 at 2.86, 3 at 2.89, 2 at 2.95 and 2 at 3.00).

From Table 5.4 the center of the phase transition for 500 vertex Degreebound graphs is at a mean degree of 2.85 (approximately). Looking at Table 5.6 we see that the time required (for Hamiltonian graphs) starts very low, and increases to a peak at 2.84 and 2.85 before dropping again. While the standard deviations suggest that these results are unreliable, the fact that the standard deviations sharply increase near the 50% region (2.84 – 2.85) suggests another look at the data. The majority of the graphs still take the same amount of time, but there is an increase in the (small) number of graphs (ultrahard, but not incomplete) that take an order of magnitude more time to solve, which brings up the average and makes the standard deviation jump. This same event occurs for mean degrees of 2.79, 2.95 and 3.00. While we do not understand the nature of the ultrahard graphs, we might expect an increase in their number in the center of the phase transition. It is more

Table 5.6: Time in seconds required by backtrack algorithm on 500 vertex Degreebound graphs.

mean	time on Ham.		time on non-Ham.		total time	
degree	mean	stddev	mean	stddev	mean	stddev
2.60	–	–	0.0	0.0	0.0	0.0
2.65	–	–	0.0	0.0	0.0	0.0
2.70	–	–	0.0	0.0	0.0	0.0
2.75	0.3	0.0	0.0	0.0	0.0	0.0
2.76	0.4	0.2	2.1	20.6	2.1	20.4
2.77	0.6	0.3	0.0	0.0	0.0	0.1
2.78	1.7	1.2	0.0	0.0	0.1	0.4
2.79	3.3	8.0	0.0	0.0	> 6.4	–
2.80	0.6	0.3	0.0	0.0	0.1	0.2
2.81	1.4	2.2	0.0	0.0	0.3	1.1
2.82	1.2	1.3	0.0	0.0	0.2	0.7
2.83	0.5	0.2	0.0	0.0	> 24.1	–
2.84	4.9	23.6	0.0	0.0	1.9	14.7
2.85	4.9	21.4	0.0	0.0	2.6	15.8
2.86	4.1	17.0	0.0	0.0	> 20.2	–
2.87	1.1	2.1	0.0	0.0	0.7	1.8
2.88	1.6	5.3	0.0	0.0	1.0	4.3
2.89	2.7	7.0	0.0	0.0	> 20.1	–
2.90	0.9	0.1	0.0	0.0	0.8	1.0
2.95	2.2	5.5	–	–	> 14.2	–
3.00	6.9	29.2	–	–	> 18.8	–

unexpected that the same phenomena occurs at the extreme right of the phase transition. One possible explanation is that such graphs with few or no degree 2 vertices undergo no initial graph reduction. The amount of graph reduction during the search will likewise be reduced. Plus, the backtrack algorithm will have more choices to make due to the greater number of edges. Thus the algorithm is more likely to take a longer period of time, since the search space is larger and it is not being pruned as effectively. This increase in time is not consistently observed because often the algorithm has no problem forming a Hamiltonian cycle. The increase in time required to solve Hamiltonian graphs for a mean degree of 2.79 is harder to evaluate because of the low percentage of Hamiltonian graphs (11 %).

One observation in [9] was that "strictly 3-connected random graphs with at least one Hamiltonian cycle (guaranteed by construction) have a solution time that grows exponentially with the size of the graph (using the above backtrack algorithm)." Our results offer an refinement of this observation and verify that the algorithm used plays an important role in determining hardness. By using search pruning, our backtrack algorithm is often able to greatly reduce the search space. Thus, for a majority of the 500 vertex Degreebound graphs of mean degree 3.00, Hamiltonian graphs were quickly solved, and our algorithm does not take an exponentially greater amount of time (compared to 200 vertex graphs). However, we do observe an increase in the number (and difficulty) of ultrahard graphs as the graph size increases. We investigate these ultrahard graphs below.

Table 5.6 shows that no time was required to solve almost all the non-Hamiltonian Degreebound graphs. We checked the data to verify this, and found that out of the 1359 non-Hamiltonian Degreebound graphs of 500 vertices, only 1 required searching. But the backtrack algorithm took 205.1 seconds to prove that this graph was non-Hamiltonian, making the graph ultrahard. The rest of the graphs were all determined to be non-Hamiltonian by initial pruning (graph reduction and global checking). This result is somewhat surprising. While it is similar to the results we obtained on non-Hamiltonian 200 vertex Degreebound graphs, one would expect that more non-Hamiltonian graphs would be generated that would require searching. However, due to the presence of a significant number of forced edges due to degree 2 vertices, there are many opportunities for graph reduction to remove many edges through a cycle of removing edges, finding new forced edges and removing more edges. The initial pruning (and graph reduction in particular) thus appears to be much more efficient on Degreebound graphs than we expected. However another possibility is that the 15 incomplete graphs could be non-Hamiltonian graphs that require extensive searching to prove their lack of a Hamiltonian cycle.

Table 5.7: Histogram of time required by our backtrack algorithm on 200 and 500 vertex Hamiltonian Degreebound graphs.

	200 vertices		500 vertices	
time interval (s)	# of trials	% of total	# of trials	% of total
$t \leq 0.2$	1000	90.1	1	0.1
$0.2 < t \leq 1.0$	85	7.7	542	73.0
$1.0 < t \leq 5.0$	17	1.5	141	19.0
$5.0 < t \leq 30.0$	2	0.2	28	3.8
$30.0 < t \leq 100.0$	4	0.4	10	1.3
$100.0 < t \leq 300.0$	0	0.0	4	0.5
$300.0 < t \leq 600.1$	2	0.2	15	2.0

We now examine the ultrahard graphs that we have occasionally found in our experiments. As we mentioned above, in our examination of the experimental data we observed that as the graph size increased, the frequency of ultrahard graphs (requiring at least an order of magnitude more time) increased. In order to analyze this behaviour, we generate a frequency histogram of the number of trials for which the algorithm required time equal to a certain time interval. Since almost all non-Hamiltonian graphs took no time, we omit them from the analysis. Table 5.7 contains the histogram for 200 and 500 vertex Hamiltonian Degreebound graphs. For each time interval and each graph size we list the number of trials and the corresponding percentage of the total this number represents. There were 1110 trials for 200 vertex Hamiltonian Degreebound graphs and 741 trials for 500 vertex Hamiltonian Degreebound graphs.

From these results we make two key observations. Firstly, the time required to solve the majority of the graphs increases from ≤ 0.2 seconds to ≤ 1.0 seconds as the graph size increases. Since the graph size increases by a factor of 2.5, we expect this result. Secondly, we observe on the 500 vertex graphs a higher frequency of graphs that require more time, compared to the 200 vertex graphs. Both frequency distributions have a similar shape (like a decaying exponential), but the slope appears to be less steep for the 500 vertex graphs. This demonstrates that there is an increase in the frequency of harder graphs as the graph size increases.

Note that the trials included in the above table came from points all along the phase transition. We also want to determine if the frequency of harder graphs increases as we move towards the center of the phase transition. In Table 5.8 we examine the histogram of the trials of graphs for various ranges of mean degree: 2.60–3.00, 2.81–2.89, and 2.84–2.86 (which have 741, 418 and 146 trials each). Note that each range is centered on the center

Table 5.8: Histogram of the percentage of trials within a specific time interval (using our backtrack algorithm) on 500 vertex Hamiltonian Degreebound graphs of varying ranges of mean degree.

	range of mean degrees			
time interval (s)	2.60 – 3.00	2.81 – 2.89	2.84 – 2.86	2.75 – 2.80
$t \leq 0.2$	0.1	0.0	0.0	2.8
$0.2 < t \leq 1.0$	73.0	73.9	76.7	69.4
$1.0 < t \leq 5.0$	19.0	18.2	15.1	22.2
$5.0 < t \leq 30.0$	3.8	3.3	2.7	2.8
$30.0 < t \leq 100.0$	1.3	1.7	2.1	0.0
$100.0 < t \leq 300.0$	0.5	0.5	1.4	0.0
$300.0 < t \leq 600.1$	2.0	2.4	2.1	2.8

of the phase transition (mean degree 2.85), and the limits of the range approach the center point. We also look at another range, 2.75 – 2.80, which avoids the center of the phase transition entirely (this range has only 36 trials). For this histogram, only the percentages are given.

The results of Table 5.8 show that the frequency of appearance of harder graphs is not affected by the phase transition. All four ranges have similar distributions. (The distribution of the 2.75 – 2.80 range is of limited accuracy due to the low number of trials.) This is perhaps the best evidence that despite the existence of a phase transition for Degreebound graphs, the graphs are not hard for our backtrack algorithm. Furthermore, the locations of the hard (or harder) graphs we do find do not correspond to the location of the phase transition.

Our backtrack algorithm randomly selects a degree 3 vertex as the initial vertex to being the search. It is possible that on the ultrahard graphs, the selection of a different initial vertex would cause the algorithm to quickly solve the graph. Such graphs are of course not as hard as graphs which require much time given any starting vertex. Thus, we investigate the hardness of the incomplete ultrahard graphs by executing our backtrack algorithm 10 times per graph (with the time limit of 10 minutes still in place). In this experiment, our backtrack algorithm is performing like a heuristic algorithm (success is not guaranteed and we have multiple trials) so we measure the probability of success (probability of finding a Hamiltonian cycle or proving none exist within the time limit) and calculate the expected time for the backtrack algorithm to find a solution. We examine the 15 incomplete 500 vertex Degreebound graphs of various mean degrees. Table 5.9 contains our results, with the graphs sorted by increasing mean degree. (We do not list the failure times on the table because these times are always 600 seconds.)

Table 5.9: Performance of our backtrack algorithm on 500 vertex incomplete, ultrahard Degreebound graphs.

graph #	mean degree	Hamil-tonian?	% solutions found	success time (s) mean	success time (s) stddev	expected time (s)
1	2.79	unknown	0	–	–	> 600.0
2	2.83	unknown	0	–	–	> 600.0
3	2.83	unknown	0	–	–	> 600.0
4	2.83	unknown	0	–	–	> 600.0
5	2.83	yes	60	121.0	146.6	521.0
6	2.86	yes	100	1.2	1.3	1.2
7	2.86	yes	90	34.9	89.9	101.5
8	2.86	yes	80	1.0	0.8	151.0
9	2.89	yes	100	1.2	0.9	1.2
10	2.89	yes	100	4.4	6.5	4.4
11	2.89	yes	100	0.8	0.5	0.8
12	2.95	yes	100	9.0	24.0	9.0
13	2.95	yes	100	39.1	111.6	39.1
14	3.00	yes	100	0.7	0.2	0.7
15	3.00	yes	100	0.7	0.3	0.7

The results of this experiment confirm that many of the graphs we thought to be ultrahard are in fact easy. Of the 15 graphs, 7 have expected times of less than 10 seconds, 1 less than 40 seconds, 2 less than 3 minutes and 1 less than 9 minutes. Only 4 were not solved in any of the 10 trials. Most of the graphs have high solution rates, which suggests the reason they were detected as ultrahard initially was due to the low probability event of choosing an initial vertex which was very bad for the search. In examining the data for the graphs that have large average success times, we observe that this result is due almost always to a single long time (> 60 seconds) averaged with the other short times (< 3 seconds). This indicates that for these graphs there is an approximate probability of 10% that a poor initial vertex will be chosen. For graph # 5 however most of the successful runs took over 10 seconds, and the success rate was only 60%. While good choices for initial vertices exist, there seem to be many more poor or bad choices. Thus these graphs (tend to) remain hard despite the choice of initial vertex, and therefore these are the truly ultrahard graphs. Another observation from Table 5.9 is that the graphs tend to become easier as the mean degree increases. In particular, the unknown graphs are all located below the center of the phase transition. Unfortunately, our test size is too small to draw any firm conclusions about this.

Note that it is also possible for graphs we found to be easy initially to actually require much time for most starting vertices, which would in actuality make them hard. However, it seems likely that the probability of this would be approximately equal to the probability of finding a graph hard initially. Since this probability is low, we do not investigate this possibility.

We conclude that Degreebound graphs in general are easy for our backtrack Hamiltonian cycle algorithm. However, rarely we do encounter harder graphs, and a few of these seem extremely hard (or ultrahard) for our backtrack algorithm. Future work could entail investigating these graphs.

5.5.3 Hardness of Degreebound Graphs for Heuristic Algorithms

We now examine how hard Degreebound graphs are for heuristic Hamiltonian cycle algorithms. Unlike backtrack Hamiltonian cycle algorithms, we expect heuristic algorithms to have problems with these graphs. The reason for this is based on the difference in operation between the two types of algorithms. The Hamiltonian cycle heuristic algorithms extend a path to unvisited vertices until they reach a dead-end. At this point the algorithms use non-path edges from the endpoint(s) to other vertices in the path to transform the path such that further extension of the path is possible. The more non-path edges that exist, the more possibilities are available to the algorithm, and the less likely it will become stuck and be forced to give up.

Table 5.10 shows the results for our heuristic Hamiltonian cycle algorithm on Degreebound graphs of 200 vertices that are guaranteed to be Hamiltonian. As was discussed in Section 5.3, we calculate the total expected time to account for the number of failed trials we expect the algorithm would have to execute before finding a Hamiltonian cycle in a successful trial.

The results of Table 5.10 match our expectations. Our first observation is that the average success time and average failure time remain almost constant across the phase transition (with little variation). This is as expected. Initially, the heuristic algorithm has as many possible paths to follow as a backtrack algorithm. But since the heuristic algorithm does not back up, the addition of a vertex to the path eliminates (in a sense) the other choices, and reduces the space the algorithm could possibly search. Furthermore, when the algorithm starts applying rotational transformations, it can perform no more than n (and will usually apply only a few) before reaching a dead-end or further expanding the path. This represents a further constriction of the search space. (Contrast this with an heuristic algorithm using the crossover extension technique, which takes longer but explores more of the search space.) In essence, the heuristic ignores large (hopefully unimportant) sections of the search space in order to improve its performance. Thus, there is less variation in the

Table 5.10: Results for our heuristic algorithm on 200 vertex Hamiltonian Degreebound graphs.

mean degree	% solutions found	success time (s)		failure time (s)		expected time (s)
		mean	stddev	mean	stddev	
2.725	15.2	0.2	0.1	0.4	0.0	2.6
2.75	15.8	0.2	0.1	0.4	0.0	2.4
2.775	2.7	0.2	0.1	0.4	0.0	16.1
2.80	1.3	0.2	0.1	0.5	0.0	36.5
2.81	0.5	0.3	0.1	0.5	0.0	96.6
2.82	0.6	0.3	0.1	0.5	0.0	73.0
2.83	0.8	0.2	0.1	0.5	0.0	61.6
2.84	1.0	0.3	0.1	0.5	0.0	52.6
2.85	0.3	0.4	0.2	0.5	0.0	162.5
2.875	1.3	0.4	0.2	0.6	0.0	47.1
2.90	1.6	0.3	0.1	0.6	0.0	39.1
2.95	18.7	0.5	0.2	0.8	0.0	4.1
3.00	99.7	0.3	0.2	1.6	0.0	0.3

execution times of such a heuristic algorithm as compared to a backtrack algorithm. Note that there is a slight increase in success and failure time as the mean degree increases. The increase in mean degree means there are more degree 3 vertices and therefore more extra edges that can be used by the heuristic algorithm. With these additional choices available, the algorithm will take longer before hitting a dead-end. And if it takes longer to fail, this gives it the opportunity to take a longer period of time before finding a Hamiltonian cycle.

Our second observation is that the solution rate (% of solutions found) varies greatly across the phase transition, and is near zero except at the extreme ends of the transition. The solution rate only jumps over 50% when all the degree 2 vertices are gone (at a mean degree of 3.00). While we expected our heuristic algorithm to do better as the number of options (i.e. non-path edges) increased, this result seems to indicate that the algorithm has a possible weakness in dealing with degree 2 vertices. The rise of the algorithm's success rate at the lowest mean degrees is possibly due to the fact that there are fewer degree 3 vertices for the algorithm to make a 'wrong' choice on compared to the center of the phase transition region. Since the success time and failure time experience little change, the change in solution rate is what causes the sharp increase in expected time. Note that it is difficult to tell if the lowest solution rates correspond exactly to the center of the phase transition. The differences between the various low rates are of almost no significance, since a rate of 0.5% represents only 5 successes out of 625 trials. However, it is apparent that there is a strong correlation between hard Degreebound graphs and the phase transition.

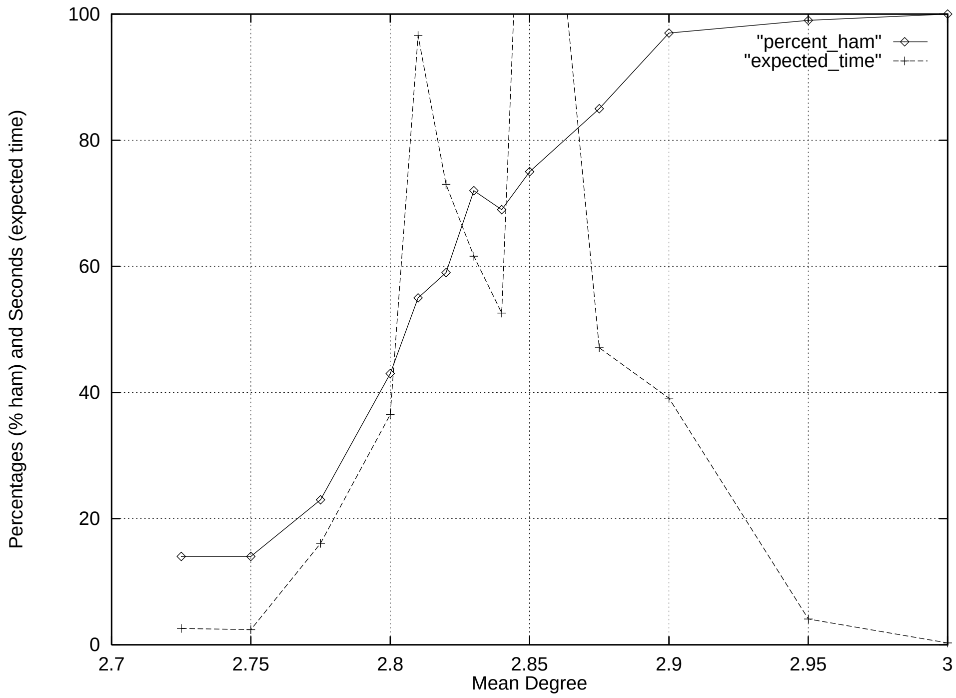

Figure 5.2: Expected time (s) and % of Hamiltonian graphs for our heuristic algorithm versus mean degree for 200 vertex Degreebound graphs.

Figure 5.2 graphs both the expected time and the percentage of Hamiltonian graphs as a function of mean degree. This makes more apparent the strong correlation between the hard region and the phase transition.

Since 200 vertex Degreebound graphs are clearly hard for our heuristic algorithm, we repeat the same experiment on 100 vertex Degreebound graphs. On these graphs the heuristic algorithm performs much better. The solution rate decreases only slightly as it approaches the center of the phase transition (mean degree of 2.79), falling from near 100% at both extremes (2.60 and 3.00 mean degree) to a low of 48.3%. Execution times drop by somewhat more than 50% compared to the times on 200 vertex Degreebound graphs, as we would expect. Due to the much higher solution rates, the expected time increases only marginally from the extremes to the center of the phase transition. Thus, our heuristic algorithm does not find these 100 vertex graphs hard.

The sharp increase in difficulty in going from 100 vertices to 200 is a strong indicator that Degreebound graphs are hard for our heuristic algorithm. Another point of evidence is the correlation between the drop in solution rate and the phase transition region for 200 vertex Degreebound graphs. Finally, the large expected times (of 30 seconds or more) on these 200 vertex graphs demonstrates that these graphs really are hard. Therefore we conclude that 200 vertex Degreebound graphs are hard for our heuristic algorithm, which contrasts greatly with the results we obtained using our backtrack algorithm. This demonstrates again that the algorithm used can have a large influence upon the difficulty of a graph or set of graphs.

5.6 Generalized Knight's Circuit Graphs

In Chapter 4 we examined the knight's tour problem as a source of hard Hamiltonian cycle graphs. In Section 4.3 we introduced the generalized knight's circuit problem as a generalized form of the knight's tour problem. Instances of this problem are specified by the 4-tuple $(A, B) - n \times m$. We defined instance classes to be sets of instances with parameters A, B and n fixed, and with parameter m allowed to vary. In the following sections of Chapter 4 we presented our theoretical and experimental results showing which instance classes had no circuits (no Hamiltonian graphs) and which instance classes did. Table 4.1 in Section 4.6 lists the non-Hamiltonian instance classes we identified and Table 4.2 shows our experimental results in searching for circuits in other instance classes. In this section, our goal is to show that the generalized knight's circuit problem is a good source of hard graphs for our backtrack Hamiltonian cycle algorithm.

In the previous sections on $G_{n,m}$ random graphs and Degreebound graphs, we were able to make use of phase transitions to help identify whether those graphs were hard or not for our algorithms. Unfortunately, there is no good way to use phase transitions with the generalized knight's circuit set of graphs. There is no clear constraint parameter which separates the Hamiltonian graphs from the non-Hamiltonian graphs. Instead, we will compare the expected time required by our algorithm to solve the different instances of the various instance classes. In our analysis of Degreebound graphs in the previous section, we defined ultrahard graphs to be graphs that took an order of magnitude more time to solve than the average graph. Thus, we will define hard generalized knight's circuit graphs to be graphs of similar size to the Degreebound graphs that take as long to solve as the hard or ultrahard graphs. To help avoid the problem of poor initial vertex selection, we perform 10 trials for each graph (problem instance). We report the success rate (% of the graphs for which the algorithm found a circuit or proved no circuit was possible) and the expected time (as calculated using the formula described in Section 5.3). We use our backtrack algorithm as described in Section 5.3. (The 10 minute time limit is in effect.)

Thus, if we find generalized knight's circuit graphs of up to 500 vertices which have an expected time of more than 30.0 seconds, these would certainly qualify as hard graphs

Table 5.11: List of instance classes examined.

instance class	range of m	instance class	range of m
$(1,2)-3\times m$	$10-40$	$(2,3)-5\times m$	$6-40$
$(1,2)-5\times m$	$6-50$	$(2,3)-9\times m$	$10-20$
$(1,4)-5\times m$	$8-60$	$(2,3)-10\times m$	$10-20$
$(1,4)-7\times m$	$8-30$	$(2,5)-7\times m$	$8-40$
$(1,4)-8\times m$	$8-10$	$(2,5)-9\times m$	$10-30$
$(1,4)-9\times m$	$10-20$	$(2,5)-10\times m$	$10-20$
$(1,4)-10\times m$	$10-20$	$(2,5)-11\times m$	$12-20$
$(1,6)-7\times m$	$8-50$	$(2,7)-9\times m$	$10-30$
$(1,6)-9\times m$	$10-40$	$(2,7)-11\times m$	$12-30$
$(1,6)-11\times m$	$12-20$	$(2,7)-12\times m$	$12-30$
$(1,8)-9\times m$	$10-50$	$(2,7)-13\times m$	$14-30$
$(1,8)-11\times m$	$12-40$	$(3,4)-7\times m$	$8-40$
$(1,8)-13\times m$	$14-30$	$(3,4)-13\times m$	$14-30$
$(1,8)-15\times m$	$16-30$	$(4,5)-9\times m$	$10-30$
$(1,8)-15\times m$	$16-30$		

as compared to Degreebound and $G_{n,m}$ random graphs. Note that most of the "hard" Degreebound graph results were merely artifacts of poor initial vertex selection, whereas this effect is minimized in these experiments by performing multiple trials. Thus any generalized knight's circuit graphs of up to 500 vertices that had an expected time greater than 600.0 seconds would definitely qualify as ultrahard.

Table 5.11 shows a list of generalized knight's circuit problems we examined. It is organized by instance class, and gives the range of m examined for each instance class. We examined a total of 360 graphs. (Remember that for any $(A,B)-n\times m$ instance, if n is odd m must be even.) The $(1,4)-8\times m$ instance class is the only instance class for which we know that all the instances are non-Hamiltonian (by Theorem 19). Some of the other instance classes we know to have many Hamiltonian instances ($(1,2)-3\times m$, $(1,2)-5\times m$, $(1,4)-5\times m$) (see Chapter 4 for details).

Of the 360 instances examined, 123 graphs (34 %) were found to be Hamiltonian and 221 graphs (61 %) were found to be non-Hamiltonian. For the remaining 16 graphs (4.4 %) our backtrack algorithm failed (reached the 10 minute time limit) every trial, thus making these graphs ultrahard (since the expected time to solve these instances is >600 seconds). These instances are listed in Table 5.12. The number of vertices for each instance is provided in the table. The $(1,4)-8\times 10$ instance is particularly interesting. It is known to be non-Hamiltonian by Theorem 19. The graph contains only 80 vertices yet our algorithm was never able to finish within 10 minutes.

Table 5.12: Generalized knight's circuit instances for which our backtrack algorithm never succeeded.

problem instance	# of vertices	problem instance	# of vertices
$(1,4)-5\times 30$	150	$(2,5)-10\times 18$	180
$(1,4)-5\times 36$	180	$(2,5)-10\times 19$	190
$(1,4)-8\times 10$	80	$(2,5)-10\times 20$	200
$(1,6)-7\times 50$	350	$(2,5)-11\times 14$	154
$(1,6)-9\times 40$	360	$(2,7)-13\times 28$	364
$(2,5)-7\times 28$	196	$(2,7)-13\times 30$	390
$(2,5)-7\times 32$	224	$(3,4)-7\times 32$	224
$(2,5)-10\times 17$	170	$(3,4)-7\times 34$	238

Table 5.13: Histogram of the expected time required by our backtrack algorithm on 221 non-Hamiltonian generalized knight's circuit instances.

time interval (s)	# of trials	% of trials
$t = 0.0$	206	93.2
$0.0 < t \leq 0.2$	3	1.4
$0.2 < t \leq 1.0$	3	1.4
$1.0 < t \leq 5.0$	3	1.4
$5.0 < t \leq 30.0$	3	1.4
$30.0 < t \leq 100.0$	0	0.0
$100.0 < t \leq 300.0$	1	0.5
$300.0 < t \leq 600.1$	0	0.0
$t > 600.1$	2	0.9

We now examine the generalized knight's circuit instances which our backtrack algorithm found to be non-Hamiltonian. Table 5.13 contains a histogram of the expected time required on these non-Hamiltonian instances.

These results support our previous findings on random $G_{n,m}$ and Degreebound graphs: our backtrack algorithm is highly efficient at finding non-Hamiltonian graphs. Despite the use of proofs to eliminate non-Hamiltonian instance classes, a majority of the non-Hamiltonian graphs were still easily solved by our algorithm. However, a few hard and ultrahard non-Hamiltonian graphs were found. Table 5.14 has a list of these. Note that the $(1,4)-8\times 10$ instance is included; while our algorithm failed on all 10 trials on this instance, we know it to be non-Hamiltonian by Theorem 19 and thus include it here. Since only a few hard non-Hamiltonian graphs were found, we conclude that our generalized knight's circuit problem is not a good source of hard non-Hamiltonian graphs. However it is still much better than the Degreebound and $G_{n,m}$ graphs.

We now examine the generalized knight's circuit instances for which our backtrack al-

Table 5.14: Backtrack algorithm results on hard non-Hamiltonian generalized knight's circuit graphs.

problem instance	# of vertices	success rate	expected time (s)
$(1,4)-8\times 10$	80	0	> 600.0
$(2,3)-10\times 12$	120	60	913.7
$(2,5)-9\times 30$	270	100	142.2
$(2,5)-10\times 16$	160	30	1519.7

Table 5.15: Histogram of the expected time required by our backtrack algorithm on 123 Hamiltonian generalized knight's circuit instances.

time interval (s)	# of trials	% of trials
$t \leq 0.2$	37	30.1
$0.2 < t \leq 1.0$	9	7.3
$1.0 < t \leq 5.0$	6	4.9
$5.0 < t \leq 30.0$	4	3.3
$30.0 < t \leq 100.0$	8	6.5
$100.0 < t \leq 300.0$	29	23.6
$300.0 < t \leq 600.1$	6	4.9
$t > 600.1$	24	19.5

gorithm found a Hamiltonian cycle. Table 5.15 contains a histogram of the expected time required on these Hamiltonian instances. These results clearly indicate that the generalized knight's circuit problem is a good source for hard (and ultrahard) Hamiltonian graphs. The histogram shows a much greater percentage of non-easy graphs as compared to Degreebound graphs (see Table 5.7). Of particular interest is the large number of ultrahard graphs (ones which had an expected time of more than 600 seconds). Table 5.16 lists these 24 ultrahard Hamiltonian generalized knight's circuit instances. Note that since we only performed 10 trials, we expect a high variance in the algorithm success rate, which produces a correspondingly high variance in the expected time.

The experiments performed in this section indicate that the generalized knight's circuit problem is a good source of hard graphs for the Hamiltonian cycle problem. A natural question to ask is why this is so. This is clearly an area for future research, but we have some preliminary comments and observations. First, we observe that the question of which instances are Hamiltonian or not is much harder to answer than for random $G_{n,m}$ or Degreebound graphs. The random non-Hamiltonian graphs were either not biconnected or could be pruned to the point where non-Hamiltonicity was obvious. This still seems to be true for the majority of the non-Hamiltonian generalized knight's circuit graphs. However, for generalized knight's circuit graphs, only vertices near the corners (and sometimes

Table 5.16: Backtrack algorithm results on 24 ultrahard Hamiltonian generalized knight's circuit graphs.

problem instance	# of vertices	success rate	expected time (s)
$(1,2)-3\times 34$	102	20	2400.1
$(1,2)-3\times 38$	102	40	900.1
$(1,2)-3\times 40$	102	30	1400.1
$(1,4)-5\times 34$	170	20	2583.5
$(1,4)-5\times 38$	190	30	1405.0
$(1,4)-5\times 42$	210	40	901.9
$(1,4)-5\times 44$	220	30	1499.2
$(1,4)-5\times 46$	230	40	900.9
$(1,4)-5\times 52$	260	40	906.0
$(1,4)-5\times 54$	270	20	2418.4
$(1,4)-5\times 60$	300	50	606.6
$(1,4)-9\times 12$	108	10	5401.3
$(1,4)-9\times 16$	144	50	680.6
$(1,4)-9\times 18$	162	40	900.3
$(1,4)-9\times 20$	180	10	5405.0
$(1,4)-10\times 11$	110	10	5500.8
$(1,4)-10\times 14$	140	50	600.9
$(1,6)-9\times 38$	342	30	1400.5
$(2,3)-9\times 20$	180	50	634.2
$(2,5)-7\times 36$	252	50	600.7
$(2,5)-7\times 38$	266	20	2400.8
$(2,5)-11\times 18$	198	20	2412.6
$(2,5)-11\times 20$	220	50	667.5
$(3,4)-7\times 36$	252	40	981.8

both sides, if the board is narrow) are of degree 2. Thus, as m increases the number of forced edges remains constant. This means the amount of pruning that can be applied remains constant as the graph size increases, which means the graph difficulty will increase. Furthermore, for many instance classes (with large enough m) we expect the graph to be biconnected (although we have no proofs concerning this). Thus, many graphs will have limited pruning and be biconnected. Some of these graphs however will be non-Hamiltonian due to other factors most likely due to the structure or regularity of these graphs. And these non-Hamiltonian graphs will most likely be hard for our backtrack algorithm.

A high percentage of the Hamiltonian generalized knight's circuit instances were hard for our backtrack algorithm to solve. We feel that the reason for this is as follows. First, as we mentioned above, the forced edges are concentrated near the ends and sides of the graph, limiting the amount of pruning that can be done. Second, there is a certain number of higher degree vertices (degree 4 and 8). As we saw with the Degreebound graphs, degree 3 vertices are able to propagate edge deletions very well, since the deletion of one edge creates two new forced edges. Thus, the presence of higher degree vertices in the generalized knight's circuit graphs will reduce this pruning propagation. Furthermore a higher average degree for these graphs means a higher branching factor and thus a larger search space for the backtrack algorithm to explore. Normally, we would expect an increase in mean degree to increase the number of solutions, therefore allowing the algorithm to take the same or less time even though the search space is larger. However, in the generalized knight's circuit graphs the structure and regularity of the problem seems to constrain the possible solutions. We suspect that the high level of regularity creates many local minima in the search tree that our backtrack algorithm must search through.

5.7 Hard Constructed Graphs

In this section we construct special Hamiltonian graphs. By utilizing our knowledge of various Hamiltonian cycle algorithms, we can form graphs which produce a poor performance for a particular algorithm.

One basic concept we use in constructing these graphs is the non-Hamiltonian edge, which we define as an edge which cannot be in any possible Hamiltonian cycle. If we can construct a graph so that the algorithm chooses to follow a non-Hamiltonian edge many times throughout the graph, then it will be very difficult for the algorithm to form a Hamiltonian cycle. Note that since the graphs are Hamiltonian, each vertex must be incident on at least two edges which are not non-Hamiltonian.

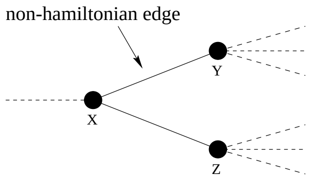

Figure 5.3: A portion of a graph with a non-Hamiltonian edge.

An important issue when constructing these graphs is the degrees of the vertices incident on the non-Hamiltonian edge. Let us consider a portion of a constructed graph containing vertices X, Y and Z. Let us assume that the Hamiltonian cycle algorithm A we are using has extended the path to vertex X, and can choose between the non-Hamiltonian edge (X, Y) and the other edge (X, Z). (See Figure 5.3.) If the algorithm uses a particular vertex selection heuristic, then we want to design the graph to exploit the heuristic and force the algorithm to follow edge (X, Y). For example, if algorithm A uses the low degree first heuristic, then we want $d(Y) < d(Z)$, which will guarantee that the algorithm will follow the non-Hamiltonian edge. If we do not want to make any assumptions about the vertex selection heuristic used by the algorithm, then we want to balance the degrees to leave the choice a random one. With $d(Y) = d(Z)$, algorithm A (and most other algorithms) will choose an edge at random. Thus, for many different choices, in approximately half of them the non-Hamiltonian edge will still be selected.

We explore several different graphs, which we label with specific names to differentiate between them. The following sections examine each of these graphs.

5.7.1 The Crossroads Graph

The crossroads graph was designed to produce extremely terrible performance for a backtrack algorithm using the low degree first heuristic and standard local search pruning, but not using component checking in local pruning. The crossroads graph $CR(k)$ is composed of k identical subgraphs (CR_S) arranged in a circle. If the subgraph has a Hamiltonian path, then the graph has a Hamiltonian cycle. Figure 5.4 contains a sample crossroads graph and Figure 5.5 contains the crossroad subgraph.

The basic idea behind the design of the crossroads subgraph is to have a choice between two edges, one being a non-Hamiltonian edge and the other being a forced edge. Vertex A of one subgraph connects to vertex E of the previous subgraph in the circle, while vertex E connects to vertex A of the next subgraph in the circle. Edge (A, B) is the non-Hamiltonian edge and edges (A, C) and (B, D) are forced because of the crossroads component subgraph

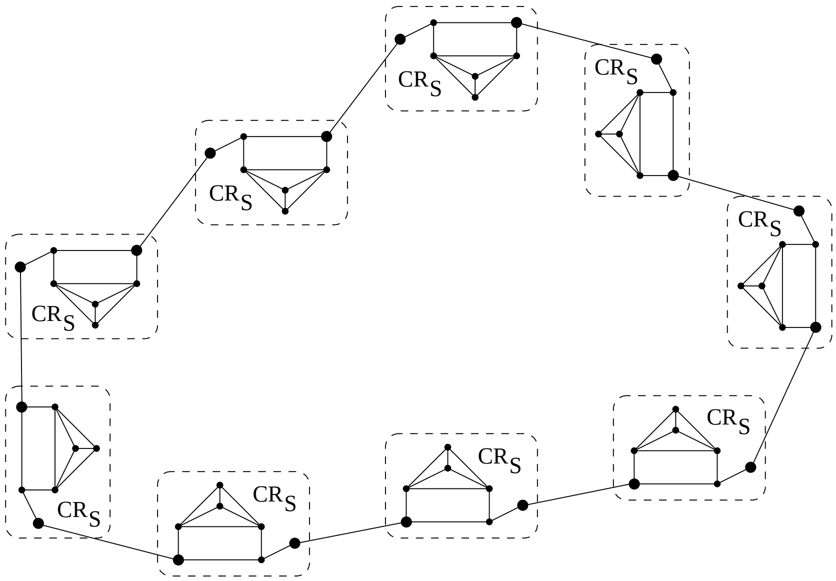

Figure 5.4: A sample crossroads graph made of 9 crossroads subgraphs.

(vertices C, D, F, G), which can only be included in the cycle if both forced edges are followed.

If the backtrack algorithm comes from an E vertex to an A vertex, it must then choose between the non-Hamiltonian edge to the degree 3 vertex B, or the other edge to the degree 4 vertex C. The low degree first heuristic will cause it to choose B. From B it will then choose to follow the forced edge to the degree 2 vertex E, and then on to the next crossroads subgraph. The algorithm will have left behind unvisited the crossroads component. The local search pruning employed by the algorithm will not change the component since its four vertices are all degree 3. Note that the crossroads subgraph is symmetric, and thus the backtrack algorithm will perform identically if it comes from an A vertex to an E vertex.

The crossroads component we show in Figure 5.5 is of the smallest possible size (since vertices C and D must have a degree ≥ 4.) There are two different valid (Hamiltonian) paths through this component (C, F, G, D and C, G, F, D). The edge (C, D) is a non-Hamiltonian edge, but will not be selected due to the low degree first heuristic (local pruning would immediately detect a dead-end and backtrack anyways). We denote the crossroads graph using k crossroads subgraphs each with this minimum sized component by $CR_m(k)$. We could instead use a larger component that has more vertices, all connected, which will increase the number of possible (Hamiltonian) paths through the component. As we discuss below, this can increase the amount of backtracking the algorithm must do.

We now analyze the performance of the backtrack algorithm on a crossroads graph. Our

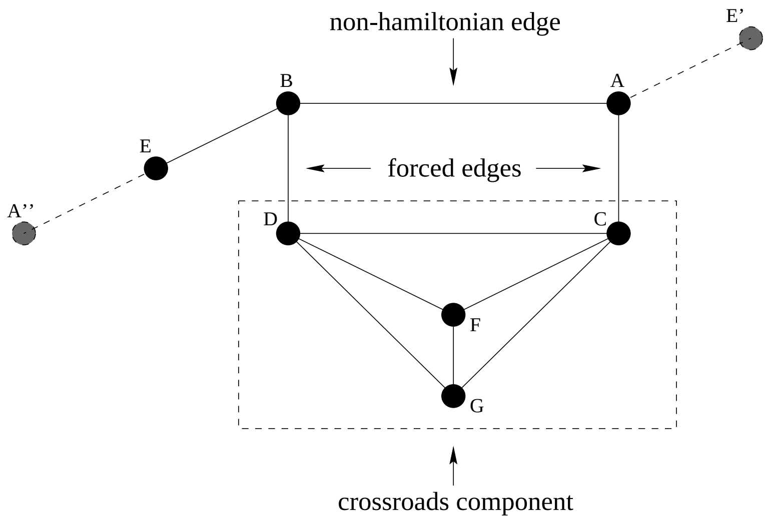

Figure 5.5: The crossroads subgraph CR_S

goal is to get an estimate on the amount of work performed by the algorithm. We observe that the algorithm only backtracks when the algorithm has gone through all the subgraphs and returned to the subgraph it started in. When the next (adjacent) vertex is already in the path, it has reached a dead-end if the algorithm has followed one or more non-Hamiltonian edges (otherwise we have a Hamiltonian cycle). The algorithm will backtrack to the point at which it had another edge to choose from, and then start to extend along this new edge. We thus analyze the amount of work done by the algorithm by calculating the number of paths $N_p(n)$ that the algorithm will generate while searching for a Hamiltonian cycle on the graph $CR(n)$. Note that if the algorithm's starting vertex is in the crossroads component, the algorithm will automatically bypass one of the non-Hamiltonian edges (and follow both forced edges). For our analysis, we assume the worst case: the algorithm starts on an A, E or B vertex and thus follows all the non-Hamiltonian edges.

We start our analysis by ignoring the crossroads component and examining the algorithm's performance. It has n decision points, and at each it can choose from 2 different edges. Due to the low degree first heuristic, it will always try the wrong edge first. Thus, the algorithm must explore all possible combinations of choices until the last combination, which produces a Hamiltonian cycle. The algorithm will therefore generate 2^n paths.

We now consider the crossroads component in our analysis. We denote the number of paths through the component as k. (For the minimum sized component, $k = 2$.) For a particular subgraph i, if we take the forced edges (not the non-Hamiltonian edge) and are forced to backtrack, we will first try the k different paths of the component of i before backtracking to subgraph $i - 1$ (and promptly choosing the non-Hamiltonian edge). For a graph with n crossroad subgraphs, we denote the number of paths generated by taking the non-Hamiltonian edge of the first subgraph as W_n, and the number of paths generated by taking the forced edges of the first subgraph as R_n. $N_p(n) = W_n + R_n$. If the algorithm follows the forced edges of the first subgraph, it is essentially reducing the problem to $n-1$ subgraphs. Thus $R_n = N_p(n-1)$. If the algorithm follows the non-Hamiltonian edge, then it will generate all possible paths, since no Hamiltonian cycle is possible without the first component. For each of the $n-1$ subgraphs, it has $1+k$ choices, the non-Hamiltonian edge or the forced edges and the k paths through the component. Thus $W_n = (1+k)^{n-1}$. $N_p(n) = N_p(n-1) + (1+k)^{n-1}$ for $n \geq 2$. For $n = 1$, $N_p(n) = 2$. Solving this recurrence equation gives us:

$$N_p(n) = \frac{(1+k)^n + k - 1}{k}$$

For the $CR_m(n)$ graph with the minimum sized component ($k = 2$) $N_p(n) = (3^n + 1)/2$. Clearly, as n increases, the backtrack algorithm must perform an exponentially increasing amount of work. Increasing the number of paths through the crossroads component will increase the rate of exponential increase. Note also that a crossroads component could be constructed from another crossroads subgraph (add two more vertices adjacent to both vertex F and G). In any case, from our analysis we see that the basic crossroads graph is hard for the backtrack algorithm.

We now consider the performance of two other algorithms on the crossroads graph. We assume that both these algorithms use the low degree first heuristic, and thus always follow the non-Hamiltonian edge first. A backtrack algorithm that uses component checking as part of its search pruning will find the graph easy. This is because instead of backtracking after proceeding through all the subgraphs, the algorithm will backtrack after following a non-Hamiltonian edge and proceeding to the next degree 2 vertex. This will cause a component to form (the crossroads component), and the algorithm will backtrack. Since the algorithm backtracks before reaching the next subgraph, the number of paths k in the component is irrelevant, and the total number of paths formed by this algorithm is $2n-1$. A heuristic Hamiltonian cycle algorithm that utilizes the cycle extension technique will also find the graph easy. Like the first backtrack algorithm, it will follow all the non-Hamiltonian edges until it returns to the first subgraph. But instead of reaching a dead-end, the algorithm will be able to form a cycle. Cycle extension will cause the cycle to be broken at one of the crossroads components not in the graph. The component will be added to the path, and a new cycle formed. This will be repeated until all the components are in the path, at which point we have a Hamiltonian cycle. Thus, we expect the heuristic algorithm to

Table 5.17: Time in seconds required by different algorithms on Crossroads graphs of varying size.

# of subgraphs	basic backtrack	component backtrack	heuristic
7	0.9	0.1	0.0
8	2.7	0.1	0.0
9	8.6	0.0	0.0
10	27.4	0.1	0.0
11	88.5	0.1	0.0
12	279.5	0.1	0.0
13	705.1	0.1	0.0
14	2277	0.1	0.0

have a 100% success rate. Furthermore, its choices for performing the cycle extension and rotational transformation are relatively straight-forward, and thus we expect the amount of work performed by the algorithm to be directly proportional to the number of crossroads subgraphs making up the graph.

We now present experimental results to confirm our analysis above. We use three Hamiltonian cycle algorithms: basic backtrack, component backtrack and heuristic. The basic backtrack and heuristic algorithms correspond to the algorithms discussed in Section 5.3. (Note that the heuristic algorithm we use does not use the low degree first heuristic and will instead choose randomly, so it will have a slightly easier time than it otherwise would.) The component backtrack algorithm is the basic backtrack algorithm with the difference that local pruning includes component checking as well as graph reduction. We modify the initial vertex selection of both backtrack algorithms to choose a random vertex of maximum degree. This will cause the algorithms to always select either a B or A vertex. Due to symmetry, there will be no difference in performance between these two selections, and therefore the backtrack algorithms will perform identically each time. Thus we only run each backtrack algorithm once per Crossroads graph. The heuristic algorithm is executed 25 times per graph, and the average time is reported. In our experiment we examine the execution times of the three algorithms as the number of subgraphs in the Crossroads graph is increased. The results are presented in Table 5.17.

As can be seen from the table, the results match our analysis. The component backtrack and heuristic algorithms took very little time irrespective of the size of the Crossroads graph, while the basic backtrack algorithm took an exponentially increasing amount of time. Assuming that the time required (T) is a constant (K) times the number of paths the algorithm must construct ($N_p(n)$), we obtain the formula $T = K(3^n + 1)/2$. Solving

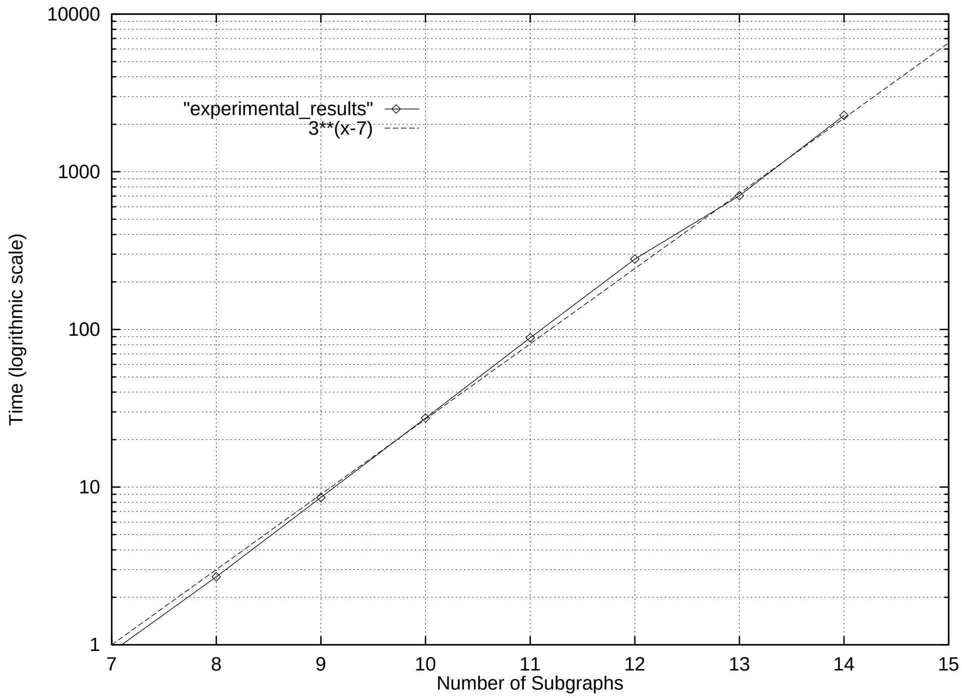

Figure 5.6: Experimental and theoretical times to solve Crossroads graphs.

for K given the initial result that 7 subgraphs takes about 1 second, we get $K = 2/3^7$, and simplify the equation to $T = 3^{n-7}$. Figure 5.6 shows our experimental values and the line corresponding to our theoretical equation. While the fit is not perfect, the experimental and theoretical rates of exponential increase are quite similar.

On an absolute scale, compared to other graphs, the Crossroads graphs are very hard indeed. A crossroads graph with 14 subgraphs has 98 vertices, and it takes our basic backtrack algorithm 47 minutes to solve. No 100 vertex Degreebound graph we generated takes this long to solve. Even most 200 vertex Degreebound graphs (except for the 2 incomplete graphs) take much less time, even for the heuristic algorithm which found 200 vertex Degreebound graphs hard (expected times were less than 3 minutes). Only the hardest generalized knight's circuit instances take similar amounts of time (using expected time) or more (2 graph of 108 and 110 vertices had expected times of approximately 90 minutes). However only on the Crossroads graphs do we observe an exponential increase in the time required with respect to the size of the graph.

5.7.2 The Interconnected-Cutset Graph

The crossroads graph of the previous section was only hard if a backtrack algorithm did not use component checking during the search. In this section, we present a constructed graph,

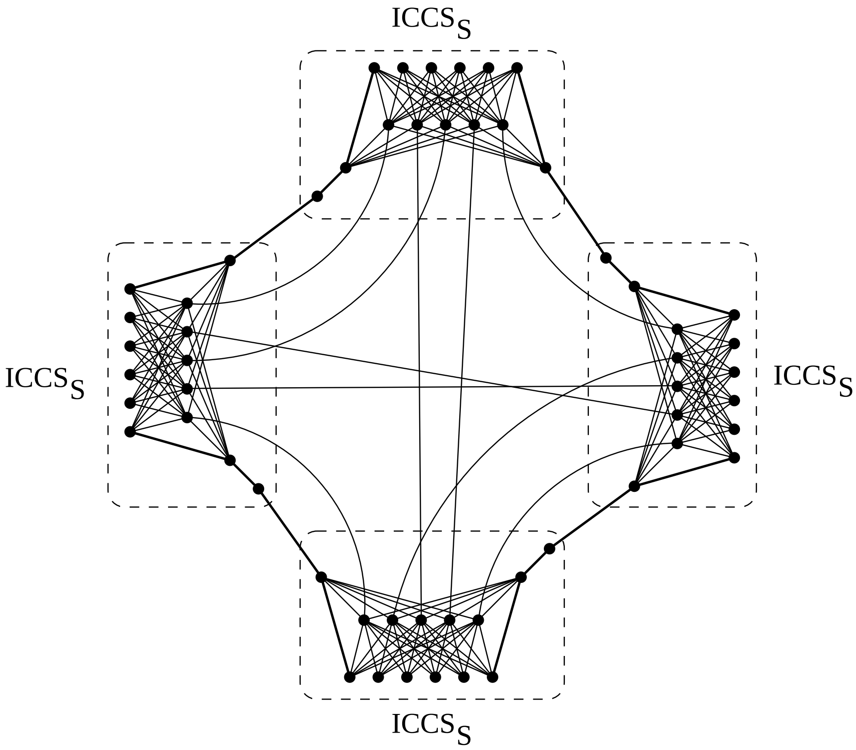

Figure 5.7: A sample *ICCS* graph.

the interconnected-cutset graph (or *ICCS*) which will not become easier when component checking search pruning is used. Due to space constraints, we only introduce the graph and provide a short discussion on it.

We designed the *ICCS* graph for a backtrack algorithm using no degree selection heuristic (so the neighbours of the current endpoint are visited in a random order). Like the crossroads graph, the $ICCS(k)$ graph is composed of k identical subgraphs $ICCS_S$ arranged in a circle. The subgraph has a Hamiltonian path between the connecting vertices and therefore the *ICCS* graph is Hamiltonian. Due to the construction of the *ICCS* subgraph, extra non-Hamiltonian edges can be added between different subgraphs. These edges help prevent components from forming during the search, which greatly reduces the effectiveness of the component checking search pruning. See Figure 5.7. Heavy lines are forced edges that must be in any Hamiltonian cycle.

The basic idea behind the design of the *ICCS* subgraph is to have a connecting set of vertices S_C, which when removed as a cutset produces $|S_C| - 1$ isolated vertices from an independent set of vertices S_I. (So $|S_I| = |S_C| - 1$.) Note that according to Theorem 5 a Hamiltonian cycle can still exist in such a graph. However, in any possible Hamiltonian cycle we must alternate between the vertices in S_C and the vertices in S_I. If from a vertex

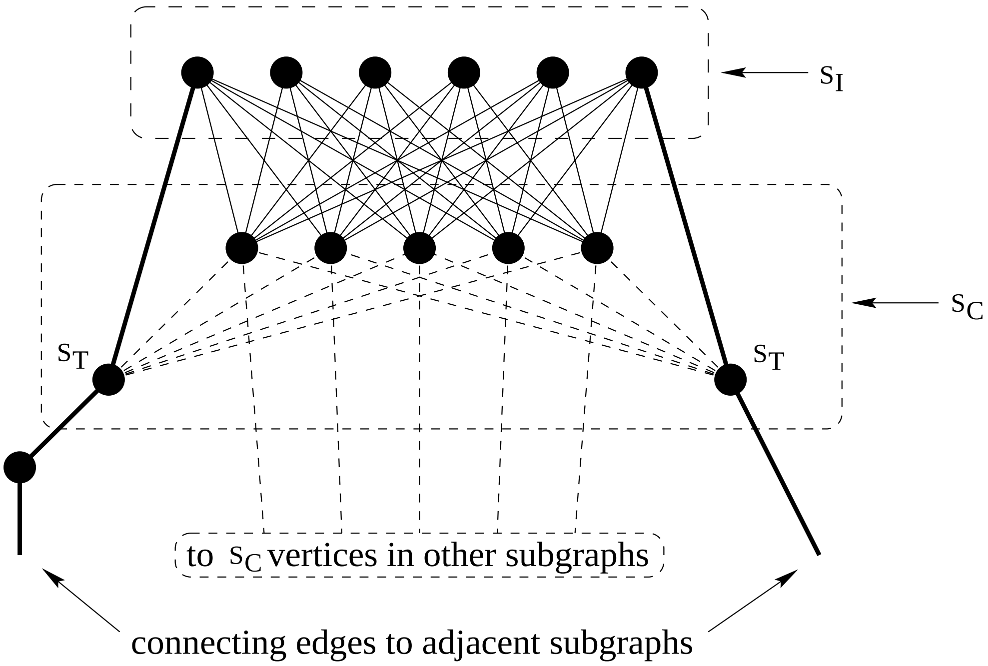

Figure 5.8: A sample *ICCS* subgraph $ICCS_S$.

in S_C we visit a vertex not in S_I then there is no way to visit all the remaining vertices in S_I and form a cycle.[2] Thus, all edges leading from a vertex in S_C to a vertex not in S_I are non-Hamiltonian edges. This allows us to interconnect subgraphs by connecting vertices of S_C of two different subgraphs. Figure 5.8 contains a sample subgraph. Non-Hamiltonian edges are denoted by dashed lines, and forced edges are denoted by heavy lines. While the size of the subgraph can be adjusted upwards, smaller subgraphs will quickly become easy due to the use of local search pruning by a backtrack algorithm.

We now discuss why the *ICCS* subgraph is hard for an algorithm to traverse entirely without taking a non-Hamiltonian edge. Two of the vertices in S_C are designated as terminating vertices S_T that are used to connect the subgraph to each of the two subgraphs adjacent to it. The S_T vertices only have one edge to a (different) vertex in S_I. This is a forced edge. Thus the other edges from S_T vertices to the other vertices in S_C are all non-Hamiltonian edges. An algorithm employing random vertex selection has a probability of $1/|S_C|$ of choosing the right edge when entering a subgraph. Thus, the algorithm will most likely choose a non-Hamiltonian edge. There are also two other ways the algorithm can go wrong. First, the algorithm can also follow a non-Hamiltonian edge from a S_C vertex in one subgraph to another subgraph. A second possibility is that after entering a subgraph

[2] We do not provide a proof, but one can be easily constructed. The idea behind the *ICCS* subgraph is very similar in concept to our work on the generalized knight's circuit problem in Chapter 4.

via one S_T vertex, the algorithm reaches the other S_T vertex and follows the forced edge to the next subgraph before having visited all the S_C and S_I vertices in this current subgraph. The non-Hamiltonian edges connecting S_C vertices in different subgraphs help prevent a component from forming, and thus component checking will not be of assistance.

Another reason why the *ICCS* subgraph is hard for a backtrack algorithm is that there are many possible paths between the two S_T vertices. A backtrack algorithm will generate many different paths trying all the different combinations before backing up to S_T vertex and trying a different edge. As we saw in the crossroads graph with respect to the crossroads component, this large number of paths will further increase the difficulty of the *ICCS* graph.

5.8 Conclusions

In this chapter we have examined the graphs that are hard for various Hamiltonian cycle algorithms. One key issue we explored is the idea that hardness of a graph depends upon the algorithm being used. This was most obvious in the performance differences between backtrack and heuristic Hamiltonian cycle algorithms. By examining which graphs were hard for which algorithm, we have gained a better understanding of how the algorithms work, and when one might be preferred over another. Our backtrack algorithm has difficulty with highly structured graphs such as those of the generalized knight's circuit problem and the Crossroads graph. However most random graphs are easy. Our backtrack algorithm had little or no problem with the $G_{n,m}$ and Degreebound graphs. The performance of our heuristic Hamiltonian cycle algorithm was considerably different. Since these algorithms have as their goal to extend the path, they require connecting vertices. Thus, the low degree Degreebound graphs are difficult for them because of the many dead-ends in the search. However, structured graphs such as the Crossroads graph in which a cycle can always be easily formed are easy due to the use of the cycle extension technique. The importance of the algorithm for hardness was also observed in work on random $G_{n,m}$ graphs. The results for our backtrack algorithm were significantly different from the results of previous researchers who used primitive Hamiltonian cycle algorithms instead. Thus, what they concluded to be hard we concluded to be easy because of the vast improvement in performance we obtained through the use of pruning.

Our search for hard graphs was successful. A small number of the Degreebound graphs proved to be very difficult for our backtrack algorithm. The generalized knight's circuit problem we devised in Chapter 4 proved to be an excellent source of hard graphs, especially hard Hamiltonian graphs. One area of future work is to investigate the reasons behind why these graphs are hard. We hope that new algorithm techniques (for the Hamiltonian cycle problem) can be developed from the findings of such research.

As we mentioned in the introduction to this chapter, one reason for investigating hard Hamiltonian cycle graphs is to learn more about the algorithms we use. We have accomplished this throughout the chapter. Our results with our heuristic algorithm show that it has difficulty with lower degree graphs due to fewer connecting edges. Future research could be performed investigating ways to improve the heuristic algorithms performance. Implementing a more sophisticated algorithm using more of the heuristic algorithm techniques discussed in Chapter 3 would be a good first step. Our results with our backtrack algorithm show that it is highly effective at identifying non-Hamiltonian graphs and also highly effective at searching low degree graphs. We have also identified some graph constructions that will cause problems for this algorithm.

One surprising discovery we made was that the performance of our backtrack algorithm can widely vary for a single graph due to the selection of the initial vertex. This effect was noticed on both Degreebound and generalized knight's circuit graphs. Due to this effect, we observed that multiple restarts of our backtrack algorithm after a time limit was reached often resulted in better performance than a single unlimited execution. This suggests a new algorithm technique for backtrack algorithms, which we call the incremental time restart technique. We define a maximum time limit t_m for the algorithm and initially set t_m to be small (a few seconds). Each time the backtrack algorithm reaches the time limit without finishing, we increase t_m by some multiple (i.e. double t_m) and restart the algorithm. After a certain number of these increases, when t_m starts becoming fairly large (and the algorithm has never finished) then we restart one last time with no maximum time limit. Our experimental results in this chapter indicate that most graphs are quickly solved at least a certain percentage of the time. Thus with this technique added to our backtrack algorithm, we would quickly solve most graphs. If we double t_m at each iteration, then the time wasted (the time spent on all the previous iterations in which the time limit was reached) is no greater than the amount of time available in the current iteration. Ultrahard graphs, particularly those that are non-Hamiltonian, would tend to not be solved within any of the time limits, and thus require that eventually the algorithm abandons the time limit and runs to completion, however long that takes. Future research can be done to investigate just how useful this technique is. In particular, we observe that it can be applied to any backtrack algorithm, and thus may be of use for other NP-C problems such as graph coloring where initial vertex selection also plays an important role.

Chapter 6

Conclusions and Future Work

This thesis has been an investigation of the Hamiltonian cycle problem, in which we have explored and characterized the interaction between the algorithms, the graphs, and the achieved performance. We have obtained and presented much evidence detailing this complex interaction. In Chapter 3 we argued that some design decisions in the creation of an Hamiltonian cycle algorithm can be made only if we have some knowledge about the properties of the graphs we are trying to solve. However, we also did show that some algorithms (or algorithm techniques and heuristics) can be expected to obtain better performance independent of the type of graph being solved. In Chapter 4 we briefly discussed the first experimental evidence for the interaction between the algorithm used and the hardness of the graph. Using time limits on our backtrack algorithm with multiple restarts allowed us to find cycles for many graphs that previously tended to take an intractable period of time.

In Chapter 5 our research into hard Hamiltonian cycle graphs produced a much greater level of insight into how the apparent difficulty of particular graphs can change greatly depending upon the algorithm being used. In particular, we found important differences in the performance of our Hamiltonian cycle backtrack and heuristic algorithms. Our experiments on Degreebound graphs clearly indicated that the majority of these almost-regular low degree graphs were easy for the backtrack algorithm, but starting at 200 vertices became quite difficult for our heuristic algorithm. To contrast with these results we presented a graph construction we called Crossroad graphs. These graphs forced our standard backtrack algorithm to take many non-Hamiltonian edges and thus required exponential time to solve. However because of the difference in its basic operation, our heuristic algorithm was able to solve the graphs easily (in linear time). By adding component checking to our backtrack algorithm we were able to get the backtrack algorithm to take linear time as well. Our second construction, the Interconnected-Cutset graph, was designed to defeat our backtrack algorithm with component checking. This graph is interesting because there seems no easy way to modify our backtrack algorithm to quickly solve the graph. This shows that not all graphs must exhibit the easy - hard difficulty with different algorithms. Just as some algorithms are always better than others, independent of the graph, some graphs are more difficult than others, independent of the algorithm (assuming a general algorithm). Another point of evidence for our thesis arose from our work on phase transitions and random graphs. Our results on the $G_{n,m}$ graph class clearly indicated that almost all of these graphs are easy for our backtrack algorithm, which contrasted greatly with the results of previous researchers. The reason for this variance in results was due to the difference in the backtrack algorithms being employed (the other researchers failed to use pruning). This clearly indicates the importance of our thesis: the interaction between algorithm and graph cannot be ignored.

Our research clearly supports our thesis: that there exists a complex interaction between the algorithms being used, the graphs being solved, and the performance obtained. Note that our research on this interaction dealt only with the Hamiltonian cycle problem, and our specific results of course only apply to that problem. However, we expect a similar kind of interaction to exist for many other NP-C problems.

Besides supporting our primary thesis, we produced other important results. Our overview of Hamiltonian cycle algorithms in Section 2.4 summarizes and classifies the different Hamiltonian cycle algorithms, techniques and heuristics presented in the literature. Chapter 3 uses this classification summary as a starting point for a discussion on design issues of Hamiltonian cycle algorithms. We introduce a classification scheme for backtrack pruning operations. And it is the consolidation of the various pruning operations scattered throughout the literature that causes our backtrack algorithm to obtain the high level of performance we observed in Chapter 5. Our analysis of heuristic algorithm techniques, methods and heuristics shows which ones are to be preferred. In addition to the analysis of previously seen techniques and heuristics, we introduce some new ones. The graph collapse technique, a method of pruning the search space by shrinking the graph to prove non-Hamiltonicity, is presented for backtrack Hamiltonian cycle algorithms. The non-path neighbour technique for heuristic Hamiltonian cycle algorithms is a simple idea that led to the development of two new heuristics for vertex selection, one of which is clearly superior to some previously used heuristics. We also combined the different vertex selection strategies for heuristic algorithms into a single vertex selection algorithm.

In Chapter 4 we presented a new generalization of the standard knight's tour problem:

the generalized knight's circuit problem. We presented various theorems concerning the non-Hamiltonicity of instances of this problem, and in the process devised a new proof method: the *partition proof technique*. Our empirical investigation of the Hamiltonicity of instances of the generalized knight's circuit problem led to several interesting observations and conjectures, which in turn led to additional proofs dealing with a specific instance class $((1,4), 5 \times m)$.

In Chapter 5 we demonstrated that despite the sophisticated Hamiltonian cycle algorithms available, there remain sets of graphs which are difficult for these algorithms. In particular our experimental results revealed that Hamiltonian generalized knight's circuit graphs tend to be quite difficult for our backtrack Hamiltonian cycle algorithm. Another important finding in this chapter concerned our investigation of phase transitions on random graphs. We found that the $G_{n,m}$ graph class was extremely easy for our backtrack algorithm despite the existence of a phase transition. Similar results were obtained on Degreebound graphs. These findings are important because they differ from those of earlier researchers and because they show that finding phase transitions is not the 'holy grail' of hard graph research.

Despite (or more realistically because of) the results presented in this thesis, there are many avenues for future research. In Chapter 3 we did not present any experimental evidence confirming our analysis of various design issues: a full-scale experimental evaluation of the different techniques and heuristics is one possible line of work. In addition, the new techniques we introduced should be implemented and evaluated. Our work on the generalized knight's circuit graphs shows the potential benefit of a time limit restart scheme for backtrack algorithms (we discuss one such technique in Section 5.8). This is another technique that should undergo an experimental investigation.

Our work on the generalized knight's circuit problem in Chapter 4 could be expanded in many directions. One possibility is to produce additional theoretical results concerning the Hamiltonicity of various instances classes of the problem. A second area of research is to investigate the structure hidden in these graphs to discover the reasons behind some of our observations (like the periodicity in certain instance classes). This research would most likely combine with an investigation of why generalized knight's circuit problems are hard to solve. A third possibility is to further extend the generalized knight's circuit problem, such as by adding a third dimension (or more) to the board, or by giving the piece multiple move steps rather than just one.

Similarly in Chapter 5 there are many possibilities for future research. One straight-forward area of study is to implement a more sophisticated heuristic Hamiltonian cycle algorithm and determine its performance on the various graphs we investigated, particularly on Degreebound graphs. Another area of study is to develop additional sets of graphs (whether randomly generated or constructed by hand) that produce hard graphs for our backtrack algorithm. A related topic of research is to investigate the hard graphs we did find to try and determine what exactly made them hard. Hopefully this additional insight in the Hamiltonian cycle problem would lead to improved Hamiltonian cycle algorithms.

Bibliography

[1] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *J. Comput. System Sci.*, 18(2):155–193, 1979.

[2] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley Publishing Co., 2nd edition, 1991.

[3] W. W. Rouse Ball and H. S. M. Coxeter. *Mathematical Recreations and Essays*. Dover Publications, Inc., New York, 13th edition, 1987.

[4] B. Bollobás, T. I. Fenner, and A. M. Frieze. An algorithm for finding Hamilton paths and cycles in random graphs. *Combinatorica*, 7(4):327–341, 1987.

[5] Béla Bollobás. The evolution of sparse graphs. In Béla Bollobás, editor, *Graph Theory and Combinatorics*, pages 35–57. Academic Press, Toronto, 1984.

[6] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Elsevier, Amsterdam, 1976.

[7] Andrei Z. Broder, Alan M. Frieze, and Eli Shamir. Finding hidden Hamiltonian cycles. *Random Structures and Algorithms*, 5(3):395–410, 1994.

[8] Francesco A. Brunacci. DB2 and DB2A: Two useful tools for constructing Hamiltonian circuits. *European Journal of Operational Research*, 34:231–236, 1988.

[9] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In J. Mylopoulos and R. Reiter, editors, *Proceedings of IJCAI-91*, pages 331–337. Morgan Kaufmann, 1991. San Mateo, CA.

[10] Nicos Christofides. *Graph Theory: An Algorithmic Approach*. Academic Press, New York, 1975.

[11] Colin Cooper and Alan Frieze. Hamilton cycles in random graphs and directed graphs. *On-line pre-print*, 1995.

[12] Paul Cull and Jeffery De Curtins. Knight's tour revisited. *Fibonacci Quarterly*, 16(3):276–286, 1978.

[13] L. Euler. *Mémoires de Berlin*. Berlin, 1766.

[14] Jeremy Frank and Charles Martel. Phase transitions in the properties of random graphs. In *CP'95 Workshop: Studying and Solving Really Hard Problems*, pages 62–69, September 1995.

[15] A. M. Frieze. An algorithm for finding Hamilton cycles in random directed graphs. *Journal of Algorithms*, 9:181–204, 1988.

[16] A. M. Frieze. Finding Hamilton cycles in sparse random graphs. *Journal of Combinational Theory, Series B*, 44:230–250, 1988.

[17] Alan Frieze, Mark Jerrum, Michael Molloy, Robert Robinson, and Nicholas Wormald. Generating and counting Hamilton cycles in random regular graphs. *Journal of Algorithms*, 21:176–198, 1996.

[18] J. N. Hooker. Needed: An empirical science of algorithms. *Operations Research*, 42:201–212, 1994.

[19] William Kocay. An extension of the multi-path algorithm for finding Hamilton cycles. *Discrete Mathematics*, 101:171–188, 1992.

[20] William Kocay and Pak-Ching Li. An algorithm for finding a long path in a graph. *Utilitas Mathematica*, 45:169–185, 1994.

[21] M. Komlós and E. Szemerédi. Limit distribution for the existence of a Hamilton cycle in a random graph. *Discrete Mathematics*, 43:55–63, 1983.

[22] Silvano Martello. Algorithm 595: An enumerative algorithm for finding Hamiltonian circuits in a directed graph. *ACM Transactions on Mathematical Software*, 9(1):131–138, 1983.

[23] Edgar M. Palmer. *Graphical Evolution: an introduction to the theory of random graphs.* John Wiley & Sons, Toronto, 1985.

[24] L. Pósa. Hamiltonian circuits in random graphs. *Discrete Mathematics*, 14:359–364, 1976.

[25] Elaine Rich and Kevin Knight. *Artificial Intelligence.* McGraw-Hill, Inc., Toronto, 2nd edition, 1991.

[26] R. W. Robinson and N. C. Wormald. Almost all regular graphs are Hamiltonian. *Random Structures and Algorithms*, 5(2):363–374, 1994.

[27] Allen Schwenk. Which rectangular chessboards have a knight's tour? *Mathematics Magazine*, 64(5):325–332, 1991.

[28] Eli Shamir. How many random edges make a graph Hamiltonian? *Combinatorica*, 3(1):123–131, 1983.

[29] Jefferey A. Shufelt and Hans J. Berliner. Generating Hamiltonian circuits without backtracking from errors. *Theoretical Computer Science*, 132:347–375, 1994.

[30] Andrew Thomason. A simple linear expected time algorithm for finding a Hamilton path. *Discrete Mathematics*, 75:373–379, 1989.

[31] D. B. West. *Introduction to Graph Theory.* Prentice Hall, 1996.