ABSTRACT

NARAYANASAMY, PRABHU. A New Heuristic for the Hamiltonian Circuit Problem. (Under the direction of Dr. Matthias F Stallmann.)

In this research work, we have discussed a new heuristic for the Hamiltonian circuit problem. Our heuristic initially builds a small cycle in the given graph and incrementally expands the cycle by adding shorter cycles to it. We added features to our base heuristic to deal with the problems encountered during preliminary experiments. Most of our efforts were directed at cubic Cayley graphs but we also considered random, knight tour and geometric graphs.

Our experimental results were mixed. In some but not all cases the enhancements improved performance. Runtime of our heuristic was generally not competitive with existing heuristics but this may be due to inefficient implementation. However, our experiments against geometric graphs were very successful and the performance was better than the Hertel's SCHA algorithm, even in terms of runtime.

A New Heuristic for the Hamiltonian Circuit Problem

by
Prabhu Narayanasamy

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2009

APPROVED BY:

_____	_____
Dr. Carla Savage	Dr. James Lester

_____

Dr. Matthias Stallmann
Chair of Advisory Committee

DEDICATION

To my most lovable family and friends....

BIOGRAPHY

Prabhu was born on June 1, 1982 in Coimbatore, India. He received his Bachelors degree in Computer Science and Engineering from Pondicherry Engineering College in 2003. After his undergraduate studies, he worked for Cognizant Technology Solutions for a period of four years from June, 2003 to June, 2007. In Fall 2007, he began his graduate studies in Computer Science at North Carolina State University.

ACKNOWLEDGEMENTS

I express my sincere thanks to my advisor, Dr. Matthias Stallmann, for his invaluable guidance and support during this research work. He has been always there for me whenever i needed help and guidance. He has always been patient in answering all my questions and explaining each and every concept in great detail.

I thank Dr. Savage for her support and suggestions. I immensely thank Ian Shields for his assistance. Finally I thank my committee members for their precious time in reviewing my thesis.

TABLE OF CONTENTS

LIST OF TABLES vii
LIST OF FIGURES viii
1. Introduction 1
2. Literature Survey 6
2.1. Fundamental Notations and Definitions 6
2.2. Theorems 8
2.3. Existing Heuristics 9
2.4. Conclusion 21
3. Heuristic Design 23
3.1. Basic Heuristic 23
3.2. Edge Pruning 32
3.3. Forced Edges and Propagation of Forced Edges 39
3.4. Farthest Edge Removal 48
3.5. Versions of the Heuristic 52
3.6. Time Complexity 53
3.7. Conclusion 54
4. Classes of Graphs 55
4.1. Cubic Cayley graphs 55
4.2. Random Graphs 59
4.3. Knight tour graphs 61
4.4. Geometric graphs 64
4.5. Conclusion 65
5. Experimental Results 66
5.1. Heuristic Settings and Machine Configuration 66
5.2. Cayley Graphs 68

5.3. Random Graphs 71
5.4. Knight Tour Graphs 74
5.5. Geometric Graphs 75
5.6. Interesting Results 76
5.7. Comparison with other heuristics 86
5.8. Conclusion 89
Bibliography 92

LIST OF TABLES

Table 3.1: Expanding the cycle 25
Table 5.1: Performance of our heuristic on Cayley graphs 69
Table 5.2: Performance of our heuristic on random graphs 72
Table 5.3: Performance of our heuristic on knight tour graphs 74
Table 5.4: Experimental Results on Cayley graphs 77
Table 5.5: Experimental Results on Random graphs 78
Table 5.6: Experimental Results on Knight-tour graphs 79
Table 5.7: Experimental Results on Geometric graphs 80
Table 5.8: Experimental Results Runtime/Trial 82
Table 5.9: Comparison for our heuristic vs SCHA algorithm 88

LIST OF FIGURES

Figure 1.1: Dodecahedron and its Hamilton Cycle 1
Figure 1.2: Path rotation sample graph 2
Figure 1.3: Illustration of path rotation on sample graph 3
Figure 1.4: Path rotation example where backtracking is needed 4
Figure 1.5: Our heuristic for path rotation sample 4
Figure 2.1: Vertex-transitive graphs 8
Figure 2.2: Examples of Platonic Solids 11
Figure 2.3: Illustration of sequential backtracking approach 12
Figure 2.4: Illustration of Pósa's heuristic – input graph G 13
Figure 2.5: Illustration of Pósa's heuristic – initial path 13
Figure 2.6: Illustration of Pósa's heuristic – path rotated 14
Figure 2.7: Pósa's Algorithm 15
Figure 2.8: DHC-1 transformation 16
Figure 2.9: DHC-2 transformation 17
Figure 2.10: DB2 Algorithm: Simultaneous Construction of cycles and paths 19
Figure 2.11: DB2 Algorithm: Merging of cycles and paths 19
Figure 3.1: Basic heuristic demonstration sample 24
Figure 3.2: Basic heuristic cycle expansion demonstration sample 24
Figure 3.3: Basic heuristic sample 25
Figure 3.4: Basic Heuristic sample with initial cycle 26
Figure 3.5: Basic heuristic sample after first cycle expansion 27
Figure 3.6: Basic heuristic sample after second cycle expansion 28
Figure 3.7: Hamiltonian Search function pseudo code 28
Figure 3.8: Start Cycle function pseudo code 29
Figure 3.9: Cycle Expansion function pseudo code 29
Figure 3.10: Symmetric graph example 30
Figure 3.11: Symmetric graph example with initial cycle 31
Figure 3.12: Symmetric graph example with final cycle 31
Figure 3.13: Symmetric graph example with edge pruned 33
Figure 3.14: Edge pruned graph with initial cycle 34
Figure 3.15: Edge pruned graph with expanded cycle 34
Figure 3.16: Edge pruned graph and the detected Hamiltonian cycle 35
Figure 3.17: Edge pruning function pseudo code 35

Figure 3.18: Hamiltonian Search function pseudo code with call to prune edges 36
Figure 3.19: Symmetric graph with starting vertex as 12 37
Figure 3.20: Symmetric graph with a different edge pruned 37
Figure 3.21: Graph to demonstrate isolated vertices 38
Figure 3.22: Graph with forced edges 40
Figure 3.23: Successful detection of Hamiltonian cycle on graph with forced edges 41
Figure 3.24: Failure of heuristic on a graph with start vertex as 6 42
Figure 3.25: Graph with marked forced edges. 42
Figure 3.26: Heuristic's attempt with start vertex as 4 43
Figure 3.27: Demonstration of propagation of forced edges 44
Figure 3.28: Detected Hamiltonian cycle with start vertex as 5 and 6 and 4 as neighbors 45
Figure 3.29: Mark forced edges function pseudo code 45
Figure 3.30: Propagate forced edges function pseudo code 46
Figure 3.31: Dense components sample 47
Figure 3.32: Dense components sample with partial cycle 47
Figure 3.33: Farthest edge removal function pseudo code. 49
Figure 3.34: Hamiltonian Search function pseudo code with call to farthest edge removal 50
Figure 3.35: Farthest edge removal - initial edge pruning 50
Figure 3.36: Farthest edge removal - edge pruning selections 51
Figure 3.37: Farthest edge removal - couple of edges pruned 51
Figure 4.1: Cayley graph 57
Figure 4.2: Petersen Graph 58
Figure 4.3: Possible moves for a knight in a chessboard 62
Figure 4.4: Hamiltonian Circuit in an 8 by 8 standard chess board 63
Figure 5.1: Edge Pruning vs Farthest Edge (both with bi-connectivity restarts) 70
Figure 5.2: Edge pruning w/o restart vs Farthest edge with bi-connectivity restart 71
Figure 5.3: Edge pruning heuristic performance 73
Figure 5.4: Average Search Length vs Number of Vertices 81

1. Introduction

A Hamiltonian Cycle or a Hamiltonian Circuit is a cycle which visits every vertex of an undirected graph exactly once and returns to the starting vertex. The problem of finding whether such a cycle exists in a graph has been proved to be NP-Complete (Karp, 1972). This famous problem is named after the mathematician *Sir William Rowan Hamilton*, who described the problem as a mathematical game (*Icosian Game*) of finding the cycle in a dodecahedron (Figure 1.1 shows a dodecahedron and Hamiltonian cycle). Hamilton's discovery can also be claimed as an inspiration from the Euler's famous knight's tour problem (Euler, 1759). Knight's tour is a mathematical game in which a knight, placed on an empty chessboard, must visit all the squares once.

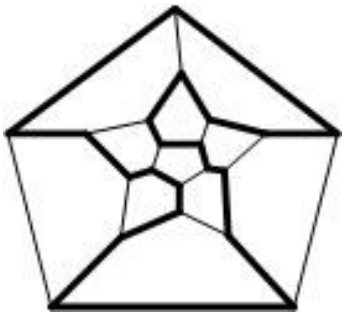


Figure 1.1: Dodecahedron and its Hamilton Cycle

Many algorithms were developed to solve the Hamiltonian cycle problem out of which the

most important is the path rotation algorithm by Pósa (Pósa, 1976). Many improvements were done on the base version of the path rotation algorithm, each of which made the path rotation technique more effective.

Pósa's algorithm constructs a path by adding vertices until it encounters a dead end, a point at which there is no possibility of adding more vertices to the path. From that juncture, it tries to do path rotation and if that is not possible it has to backtrack.

Let's look into a simple example of a path rotation technique. Consider the following graph in Figure 1.2.

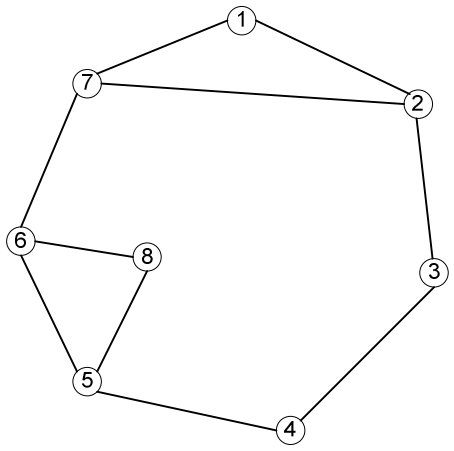
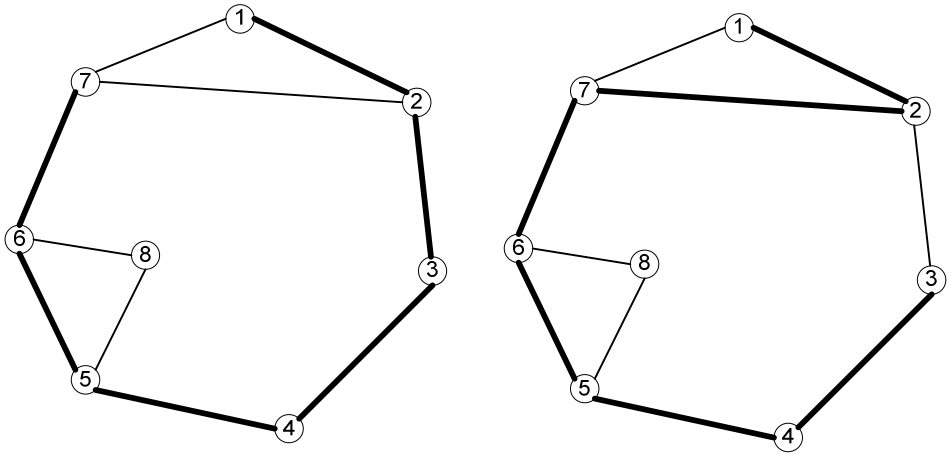


Figure 1.2: Path rotation sample graph



(a) Before path rotation *(b) After path rotation*

Figure 1.3: Illustration of path rotation on sample graph

Each vertex starting from 1 gets added to the path till it reaches the end vertex 7 and we get a resultant path as in Figure 1.3 (a). This is a dead end because the neighbors of vertex 7 are already on the path and neither of them finishes a Hamiltonian cycle. As all the vertices are not in the path, the path rotation algorithm tries to identify a potential point for rotation. Here the vertex 7 has an edge to vertex 2. Thus the path 1-2-3-4-5-6-7 can be rotated to get 1-2-7-6-5-4-3 and vertex 3 becomes the new end point as shown here as in Figure 1.3 (b).

What if the last vertex 7, as in Figure 1.3 (a), doesn't have an edge back to vertex 2? The algorithm has to backtrack till it finds a potential point for rotation. If it cannot find any point even after backtracking then the path has to be reconstructed with a new starting point. To visualize this scenario consider the Figure 1.4. As you see, there is no back edge (to the vertices that are already in the path as we had in previous example) here and the path rotation algorithm needs to backtrack. In fact Pósa's algorithm didn't have any backtracking feature.

Various implementations of the algorithm have backtracking feature built in them.

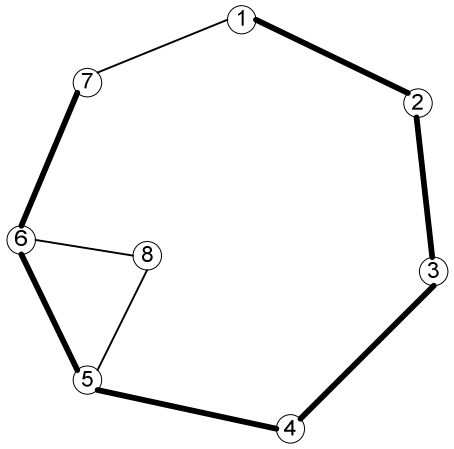
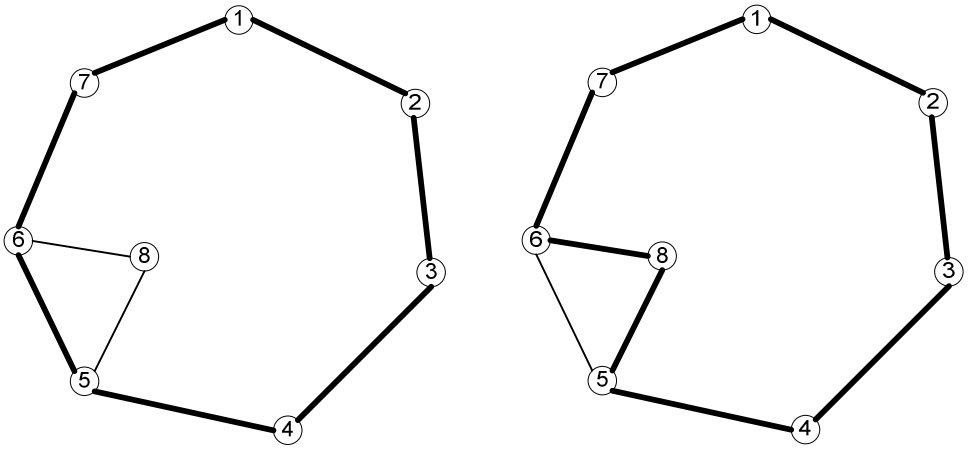


Figure 1.4: Path rotation example where backtracking is needed



(a) Initial Cycle *(b)After Cycle Expansion*

Figure 1.5: Our heuristic for path rotation sample

Our heuristic is a novel attempt to find whether the given graph is Hamiltonian. The heuristic attempts to find a smaller cycle initially and the cycle progressively gets expanded until all the vertices are included. Our heuristic works very efficiently on this example. Initially a cycle is formed and then we expand the cycle as in Figure 1.5 to include all the vertices.

Having the idea of uniting smaller cycles to make a larger cycle, our main focus in this research has been to work on this heuristic and identify its potential challenges.

We begin in Chapter 2 with a brief survey of the existing work. There we have discussed various theorems and lemmas about the necessary and sufficient conditions for a graph to be Hamiltonian. Then we describe in detail the existing algorithms and some promising results that they have managed to achieve.

In Chapter 3, we propose the base version of our heuristic and come up with various modifications to enhance the base version of the heuristic so that it can be used on various classes of graphs.

In Chapter 4, we discuss on various families of graphs, Cayley graphs, knight tour graphs, random graphs and geometric graphs, against which we ran our heuristic. We document the method of generation for each class of graph and we list the important properties that hold on the classes of graphs with respect to Hamiltonian circuits.

In Chapter 5, we document all the results that we have achieved with different versions of our heuristic against various classes of graphs described in Chapter 4. We also compare our heuristic performance against other existing heuristics.

Chapter 6 summarizes our work and describes the various suggestions that can be taken up in future to improve the heuristic's performance.

2. Literature Survey

This section is divided into three main topics Section 2.1 gives the necessary fundamental notations and definitions on graphs and other terms which are important in the context of this thesis; Section 2.2 briefly describes relevant theorems and conjectures on Hamiltonian cycles and Section 2.3 gives with a brief survey of the work that has already been put forth with respect to heuristics for the Hamiltonian circuit problem.

2.1. Fundamental Notations and Definitions

A ***graph*** consists of a vertex set $V(G)$ and an edge set $E(G)$. Each edge is a pair of vertices which are called end points of edge. A graph can be either directed or undirected. An undirected graph is denoted by $V(G) = \{v_1, v_2, v_3, \ldots v_n\}$ and $E(G) = \{u_1v_1, u_2v_2, \ldots, u_mv_m\}$. The graph here is undirected, meaning that the edge uv is same as the edge vu whereas in a directed graph the edge $uv \neq vu$.

A vertex u is ***adjacent*** to a vertex v if the graph's edge set contains the tuple uv. The ***degree*** of a vertex v is the number of vertices adjacent to v. Usually the minimum degree of a graph is denoted by $\delta(G)$ and maximum degree of a graph is denoted by $\Delta(G)$. A graph is ***regular*** if the degree of all vertices is exactly equal to some constant k.

Let n represent the cardinality of the vertex set $V(G)$. Similarly the cardinality of the edge set $E(G)$ is denoted by m. The time complexity of the algorithms is usually represented in terms m and n, the number of vertices and edges, respectively.

Another term for the number of vertices is the ***order*** of *G*.

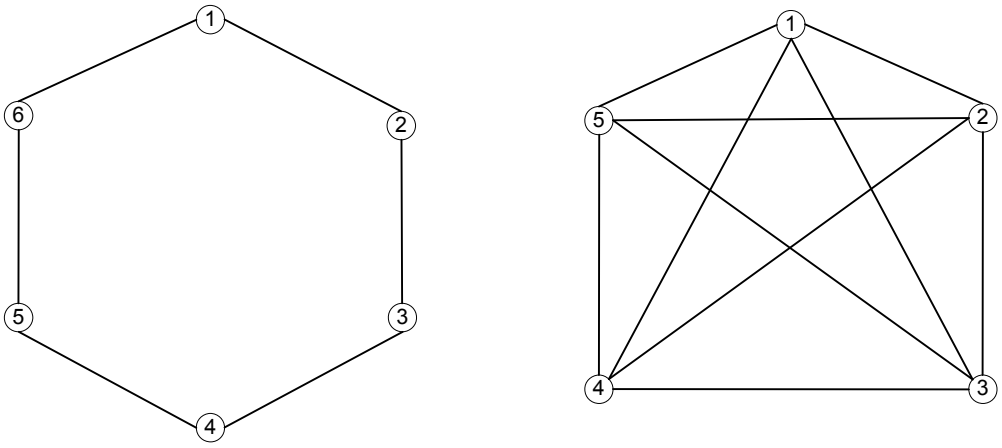
If $uv \in E(G)$ for every pair $u,v \in V(G)$ and $n=|V(G)|$, then G is the ***complete graph*** K_n. As each vertex is connected to all other vertices in the graph, all complete graphs are Hamiltonian.

A ***subgraph*** of a graph *G* is a graph *H* such that *V(H)* is a subset of *V(G)* and *E(H)* is a subset of *E(G)*. The ***components*** of a graph *G* are the maximal connected subgraphs of *G*. A ***cut edge*** or ***cut vertex*** of a graph is an edge or vertex whose deletion increases the number of components. The ***connectivity*** of *G*, written *k(G)*, is the minimum size of a vertex set *S* such that *G-S* is disconnected or has only one vertex. A graph *G* is ***k-connected*** if its connectivity is at least *k*.

An ***isomorphism*** from a simple graph *G* to a simple graph *H* is a bijection $f : V(G) \rightarrow V(H)$ such that $uv \in E(G)$ if and only if $f(u)\, f(v) \in E(H)$. An ***automorphism*** of *G* is an isomorphism from *G* to *G*.

A graph *G* is ***vertex-transitive*** if, for every pair $u, v \in V(G)$, there is an automorphism that maps *u* to *v*. An important property is that all vertex-transitive graphs are regular. As we are going to see more of these vertex-transitive graphs throughout this thesis work, let's look at a couple of examples. The 6-Cycle graph in Figure 2.1 (a) and the complete graph K_5 in Figure 2.1 (b) are common examples of vertex transitive graphs. Informally speaking, these graphs are vertex-transitive as no vertex can be distinguished from any other based on the edges surrounding it. In the Figure 2.1 (a) automorphism that maps, e.g. vertex 3 to vertex 5 are 4 to 6; 6 to 2 etc.,.

Similarly ***edge-transitive*** graph is a graph *G* such that, given any two edges e_1 and e_2 of *G* there is an automorphism of *G* that maps e_1 to e_2. The graphs in Figure 2.1 (a) and (b) are both vertex-transitive as well as edge-transitive as well.



(a) 6-Cycle graph *(b)Complete graph on 5 vertices (K_5)*

Figure 2.1: Vertex-transitive graphs

2.2.Theorems

Beyond being an NP Complete Problem (Garey & Johnson, 1979), there are various theorems and conjectures regarding the conditions under which graphs contain Hamilton cycles. In this section, we will review some sufficient and necessary conditions for a graph to be Hamiltonian.

Theorem 1*:* In a graph with a Hamiltonian cycle the degree of each vertex must be ≥ 2.

It is obvious that if the graph is not bi-connected, a path can only exist.

Theorem 2*:* If a vertex has 3 neighbors of degree 2, that graph cannot contain a Hamiltonian Cycle.

Proof: From the theorem 1, we know that the two edges incident on a degree 2 vertex must both be in the cycle. Assume such a cycle exists in the graph. The degree 3 vertex must have

two of its incident edges included in the cycle. This leaves another degree 2 vertex, the remaining neighbor of degree 3 vertex, isolated. This is a contradiction. Hence no Hamiltonian cycle may exist.

Theorem 3*:* (Dirac, 1952) If *G* is a graph of order n ≥ 3 such that $\delta(G) \geq n/2$ then G is Hamiltonian.

This theorem gives us the lower bound on the minimum degree for a graph to be Hamiltonian.

We have presented only the theorems that are relevant to this thesis work. Readers can refer to the texts (West, 2001), (Bondy & Murty, 2008) for a comprehensive coverage of all the necessary and sufficient conditions for a graph to be Hamiltonian.

2.3. Existing Heuristics

We begin this section with some early algorithms that existed for solving the Hamiltonian cycle problem and then we move on to some of the latest algorithms. Readers should consider reading surveys by Gould (Gould, Updating the Hamiltonian problem—a survey, 1991) (Gould, Advances on the Hamiltonian problem-a survey, 2003) and by Kawarabayashi (Kawarabayashi, 2001) for comprehensive coverage of all the important results regarding the Hamiltonian cycles in graphs.

As mentioned in the introduction of this thesis, Hamiltonian cycle discovery can be regarded as an inspiration from the knight's tour problem proposed by Euler (Euler, 1759). A knight's tour (refer to chapter 4 section 4.3 for detailed description of knight's tour) is a path taken by a knight in the game of chess. A knight starting from any square on a chess board must visit all the squares once and return back to the starting square. Many mathematicians De Moivre,

Roget, Vandermonde (Vandermonde, 1774), Warnsdroff (Warnsdorff, 1823), Pratt (Pratt, 1825), Lengendre (Legendre, 1830), DeLavernede (DeLavernede, 1839) got attracted towards this problem during the late 1700's and early 1800's.

First and foremost discovery of an algorithm for the knight's tour problem was by H.C. Warnsdorff in 1823. Like other mathematicians, Warnsdorff described the knight's tour as a mathematical puzzle. Later it was generalized into a Hamiltonian cycle problem.

Warnsdorff proposed to select the moves with the fewest number of outgoing edges. In other words, the vertex has to be selected such that it has a minimum degree. In the case of ties, an arbitrary choice can be made. This rule, though simple, proved to be highly successful in detecting knight tours. This is a general rule which can be applied to chess boards of any sizes. There were failures in detecting cycles in the cases when ties were broken at random.

Pohl (Pohl, 1967) in 1967 enhanced the Warnsdorff rules. Experiments conducted by Pohl based on the raw Warnsdorff rule failed on many cases due to the wrong first move and tie breaking rules. Hence Pohl came up with the following rule for breaking ties. "*For each tie move, sum the number of moves available to it at the next level and pick the one yielding a minimum"*. Pohl generalized the first move selection rule and tie breaking rule which can be summarized as follows. The vertex with highest degree has to be selected for the first move and in the case of ties the next level has to be examined. These rules proved to be highly successful and were able to find knight tours from any vertex on an 8 by 8 standard chess board. Further they were able to find tours on 20 by 20 and 40 by 40 boards.

This technique of choosing higher degree neighbors first for general graphs might not yield good results. If an algorithm chooses always the higher degree neighbors initially then at the end it might end up at dead ends and it has to backtrack eventually. In fact this procedure is exactly converse of the DB2 algorithm that we are going to describe later in this section Gardner (Gardner, 1957) in 1957 claimed that all the platonic solids (also known as regular solids) were Hamiltonian.

They are solids with equivalent faces composed of congruent regular Polygons as shown in the Figure 2.2.

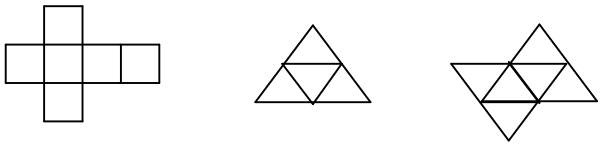


Figure 2.2: Examples of Platonic Solids

There were many interesting discoveries in finding Hamiltonian circuits in late 1960's (Danielson, 1968) (Kamae, 1967) (Kroft, 1967) (Parthasarathy, 1964) (Rao & Murti, 1969) (Roberts & Flores, 1966) (Yau, 1967).

Most of these algorithms were sequential and exhaustive. Sequential algorithms had a straightforward strategy of adding vertices to a path till it reached a dead end. Then simple backtracking approach is applied to backtrack to a vertex where further expansion is possible and the procedure continues until all the vertices get added to the path. These approaches were exhaustive and the time complexities of these approaches were exponential. Figure 2.3 shows a summarized version (original flowchart modified for better understandability) of a typical sequential search and backtracking approach by Kroft (Kroft, 1967) in finding all the Hamiltonian paths in a maze.

Another notable work was from Hakimi (Hakimi, 1966), in 1966, who constructed **deduction rules** to eliminate partial paths early during the search for a circuit. Rubin (Rubin, 1974), in 1974, came up with a search procedure which extended Hakimi's work. He added more deduction rules to eliminate partial paths. In addition to this, he extended his method to both directed and undirected graphs. The deduction rules primarily concentrated on pruning

the edges and selecting the best possible path from the given possible paths of expansion.

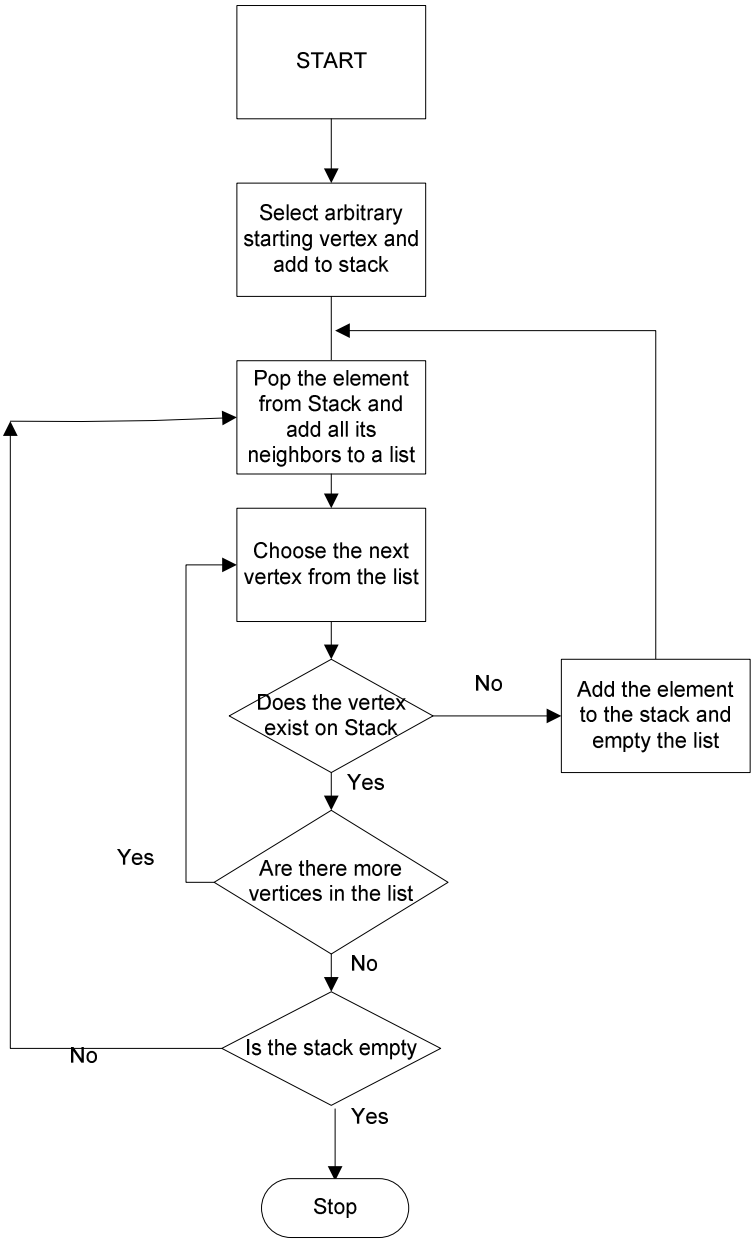


Figure 2.3: Illustration of sequential backtracking approach

In 1976, Pósa (Pósa, 1976) put forth a heuristic based on **rotational transformation**. This heuristic served as a milestone and a forerunner of many leading heuristics. Note that Euler's discovery in 1759 (Euler, 1759) also included rotational transformations while finding knight-tours in chess boards. Pósa doesn't present any implementation of his approach in his paper. Hence there are various implementations for this heuristic. His heuristic goes like this. Let *G* be a graph. An arbitrary starting point is chosen. From that starting vertex, vertices are added till we reach a dead end. Let the path thus formed be $U(x_1, x_2,\ldots\ldots,x_k)$. If *G* contains an edge x_1x_j (1<j<k) then *G* should contain a path $U`(x_{j-1}, x_{j-2},\ldots..x_1, x_j, x_{j+1},\ldots.,x_k)$. Note that both these path have a common end point $x_k$. Hence Pósa's heuristic transforms (or rotates) the path. He terms it as an allowable transformation from $U \rightarrow U`$. Care should be taken that only one end point can be changed and not both at the same time. In his version no backtracking is ever done. Many implementations of this heuristic do have backtracking along with the rotational transformation. Figure 2.4 – 2.6 illustrates a simple rotational transformation. Figure 2.4 shows the input graph *G*.

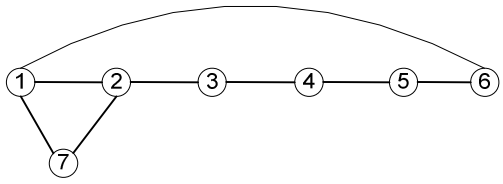


Figure 2.4: Illustration of Pósa's heuristic – input graph G

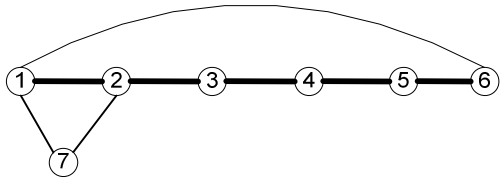


Figure 2.5: Illustration of Pósa's heuristic – initial path

Let the start vertex be 1 and Figure 2.5 shows a path *1-2-3-4-5-6* in the graph with sequential addition of vertices to 1.At this juncture, there is no further possible ways to expand the path to include the vertex 7. Hence Pósa's heuristic looks for possible rotational transformation. Here in this graph, we have an edge from 1 to 6. Fixing one end point as 1, a rotational transformation can be performed. Refer to Figure 2.6.

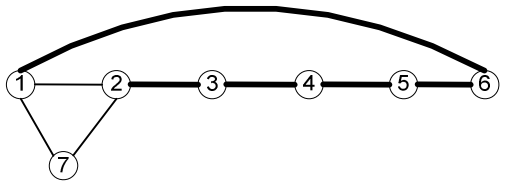


Figure 2.6: Illustration of Pósa's heuristic – path rotated

Now the vertex 7 can be added easily to the path. Algorithm for Pósa's approach is given in Figure 2.7.

The probabilistic approach by Angulin and Valiant (Angluin & Valiant, 1977), in 1977, is an interesting initiative in this area. The paper introduces two algorithms **UHC** (for undirected graphs) and **DHC** (for directed graphs). Though experimental results are not presented, the paper claims and proves that the algorithms (UHC and DHC) almost certainly find Hamiltonian circuits in random graphs having at least cn logn edges in $O(n (\log n)^2)$.

The UHC/DHC algorithm chooses an arbitrary vertex to start with. From the initial vertex it constructs a path similar to that of Pósa's heuristic. The core difference here is that the edges are deleted from the graph as they are being traversed. This is done to ensure that those edges are already part of the cycle and they should never be chosen again. The DHC algorithm has two important modes for directional rotational transformation (Note that rotational transformations for undirected and directed graphs are not the same).

Input: Graph G

Output: Success/Failure

PósaSearch(G)

1. Select an arbitrary vertex s and let it be starting vertex of path P.
2. $P \leftarrow s$
3. endpoint of $P=s$
4. While ($|P| \leq n(G)$) do
5. Select a neighbor, x, of the current endpoint at random such that x is not in P
6. If a neighbor x could be found then append x to the path P and endpoint of P is x
7. else do rotation if possible
8. Select a neighbor of the current endpoint, w, such that w is in P
9. If such a w could be found then do rotation at w
10. else return failure.
11. End while
12. If $|P| = n(G)$ then if the end points of P are connected by an edge then return success
13. else return failure

Figure 2.7: Pósa's Algorithm

Let the partial path be P. Let x be the current end point in P. DHC-1 (transformation 1) has to occur if the following conditions are met. The new vertex, v, visited (from the current end point x) is already in P and if there are at least n/2 nodes in P inclusive of v and x. If u is the predecessor of v in P then the edge uv is deleted from P. The edge xv is added. Hence the endpoint of the path now is u and the initial path P gets transformed into a path P` and a cycle C.

The DHC-2 (transformation 2) is the inverse of the DHC-1. It converts a cycle and a path to a new path. This occurs when the following conditions are met. If the new vertex v (predecessor u in cycle) to be visited (by extension of path) is in the cycle then connect the path (current end point e) and the cycle by adding an edge xv and delete the edge uv to transform into a new path p'.

The Figure 2.8 and Figure 2.9 shows DHC-1 and DHC-2 transformations. The authors of this paper have concluded with the statement "*It is likely that the particular algorithms we give can be made more efficient in practice by various "heuristic" modifications. (For example, some preliminary experiments suggest that it might be better not to delete edges as they are explored.)*".

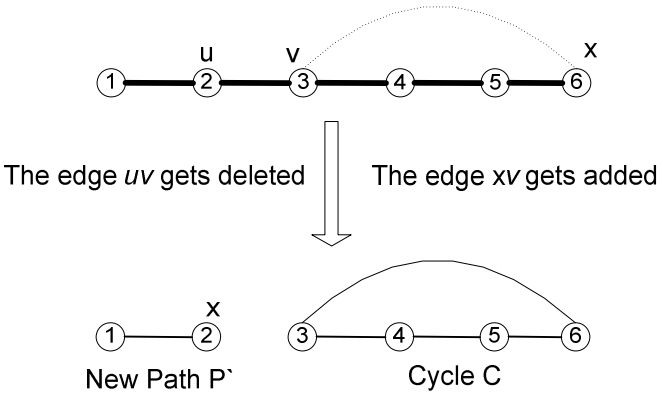


Figure 2.8: DHC-1 transformation

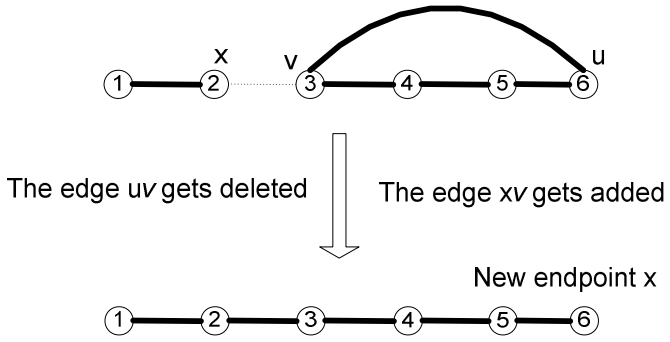


Figure 2.9: DHC-2 transformation

Interested readers can refer to the paper for the detailed algorithm and proofs regarding the run times.

Bollabas, Fenner and Frieze (Bollabas, Frieze, & Fenner, An algorithm for finding Hamilton paths and cycles in random graphs, 1987) in 1987 put forth their **Ham algorithm** for finding Hamiltonian circuits in undirected random graphs. Rather different from previous algorithms, Ham algorithm uses cycle extension technique. Two arbitrary vertices w_1 and w_2 from the bi-connected graph G are chosen such that $w_1\ w_2\ \varepsilon\ E(G)$. Let the path P contain the two vertices. The path extension is done from both the vertices w_1 and w_2 (such that all the vertices are added between them). If the new vertex to be added is not on the current path, the path gets extended. If the new vertex to be added is already in the path then immediately a cycle extension is done. Cycle extension is nothing but a rotational transformation. Let cycle C be the path $P + w_1\ w_2$. Let u be the first vertex (with neighbors u_1 and u_2 in the path; $u_1<u_2$) along the path P such that it contains a lowest numbered neighbor v that is not in C. Then cycle extension is done by adding the edge uv to the cycle C and deleting the edge uu_1 from the cycle. Later many heuristics used this idea of cycle extension. This algorithm is deterministic and doesn't make randomized choices like the previous algorithms. Frieze (Frieze, 1987) came up with a **sparse HAM algorithm** in 1987.

With his algorithm he proved that with a high probability (*whp*) his algorithm can find cycles in two classes of random graphs with constant average degree. It had minor variations from HAM algorithm, like using a depth first search for backtracking instead of breadth first and extending the path from only one end and not from both the ends. Other notable extensions of HAM algorithm are Hide HAM (Broder, Frieze, & Shamir, 1991) and Linear HAM (Thomason, 1989)

Then in 1988, Brunacci (Brunacci, 1988) came up with his **DB2 algorithm** for undirected graphs. We review this algorithm here in this survey as this is an entirely new approach rather than doing mere path extensions. DB2 algorithm uses a priority list of vertices where it prioritizes the vertices based on their degree. The degree two vertices get the top most priority. Then the priority list is filled by the edges which have degree greater than two in the increasing order. The basic idea here is to use the lower degree edges first and then use higher degree at the last so as to have many options at the end for exploring rather than encountering dead ends. The algorithm starts simultaneously from every point and adds vertices to the path. In cases of vertex having multiple neighbors, choose a vertex with lower degree based on the edge priority list. When the degree two vertices are "incompatible" (this term is used to indicate a situation of any two (or more) of the degree two vertices going to the same vertex) then Hamilton cycle cannot exist in the graph. The algorithm quits right here. If this isn't the case, the DB2 procedure adds edges, till all vertices get included and all the components remain connected. The Figure 2.10 and Figure 2.11 show how the DB2 algorithm will work. (This figure is same as the figure available in the Brunacci's paper).

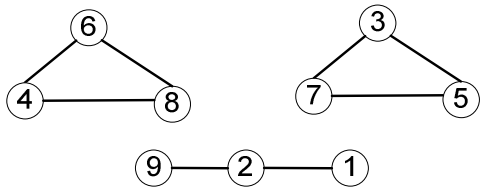


Figure 2.10: DB2 Algorithm: Simultaneous Construction of cycles and paths

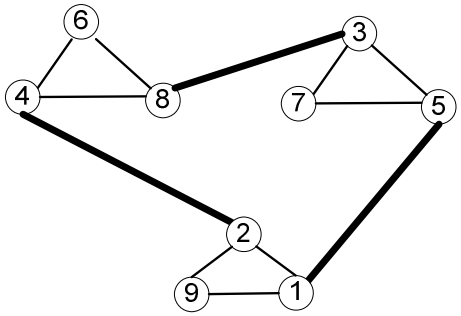


Figure 2.11: DB2 Algorithm: Merging of cycles and paths

In 1998, Vandegriend (Vandegriend, 1998) in his research work presents a detailed survey of various algorithms for finding Hamiltonian circuits. Vandegriend tries to categorize all the methods based on various strategies like backtracking, pruning (much work has been done on pruning), vertex selection.

The research work tries to combine all the strategies and apply it efficiently for the random graphs, knight tour graphs, cross road graphs, degree bound graphs, geometric graphs etc.

Alexander Hertel (Hertel, 2004) in 2004, has come up with a heuristic for detecting Hamiltonian cycles in sparse random graphs. All the work which we have reviewed so far concentrated on extending the paths, rotating the paths, extending the cycles etc. In this work,

Hertel's Hamiltonian cycle program calls another subroutine, *"probability mill"*, which basically prunes edges until a circuit is detected. The program considers edge pruning to be the most important activity and prunes the edges initially at random (if there are no available edges that can be pruned) then it goes into a deterministic mode of pruning.

A forced edge is an edge which has to be in the Hamiltonian circuit. For example, for a degree two vertex both the edges are forced. The pruning rules are as follows

1. Odd forced cut is a cut across odd number of edges. There cannot be odd number of forced cuts in a graph.
2. No Hamiltonian graph contains a hub. Hub here is a vertex with more than two forced edges.
3. Forced edges incident on the two vertices of a 2-vertex cut is termed as barricade and no Hamiltonian graph can have a barricade.
4. If a graph turns out to be non Hamiltonian after deleting an edge that edge can be safely forced. Similarly if a graph becomes non Hamiltonian after forcing an edge that edge can be safely pruned.
5. Both edges across any 2-edge cut can be safely forced.
6. If a vertex has 2 forced edges, all the rest of its edges can be safely pruned.
7. If a graph contains an un-forced edge between two vertices of a two-vertex cut then that edge can be safely pruned.
8. If a graph contains a chain (a path which has all its internal edges forced) which has unforced edges between its two heads then the edge can be safely pruned.

All of these rules govern the heuristic run time. Hertel doesn't claim that his heuristic is the fastest but the practical experiments show that it is one of the better heuristics in terms of detecting cycles in random graphs.

Shields (Shields, 2004) in 2004, in his doctoral thesis put forth an algorithm which is an improvement over the Pósa's path rotation heuristic. The research work highlights the cases

where the Pósa's search would run indefinitely. The research work suggests a fine refinement over the technique which will make the search to terminate successfully in many cases. The basic idea here is to evaluate the end points before performing the sequence of rotations. After evaluation, the most promising candidate is chosen and sequence of rotations are performed and the path is extended. If no such sequence exists the heuristic terminates.

Let P be a partial path and let *end(p)* be the endpoint in the partial path. The heuristic tries to construct a tree at the end point and determine the possibilities of rotation. Initially the *end(p)* is added to a first in first out queue and all of its neighbors are examined. Let *u* be *end(p)*. If *v* is a neighbor of *u* and if *v* is on P, then the heuristic determines the endpoint *w* that would result after the rotation of P at *v*. If *w* has not been visited then *w* is marked visited and added to the tree as well as the queue meaning that it would no longer be a vertex which is unreachable. If *v* is not on P then the heuristic has succeeded and the search terminates there by performing the rotation at the endpoint *u* to get a new path R.

To solve cubic Cayley graphs efficiently, Shields used the HAM's cycle extension algorithm (described earlier in this section) and its variants. Shields algorithm was successful in finding cycles in some of the Cayley graphs in which no algorithm found a cycle before. The algorithm was fine tuned to run against graphs with millions of vertices.

2.4. Conclusion

In this chapter we began by reviewing some fundamental theorems on Hamiltonian cycles. We discussed few of the popular algorithms and heuristics which primarily focused on techniques like path rotation, backtracking, cycle extensions and edge pruning. Interested readers can refer to the references for more detailed procedures and running times of the algorithms presented.

Most of the algorithms which were presented were specific to certain classes of graphs (mostly random and regular graphs). There is hardly any experimental evidence that a single algorithm could work for all classes or most of the classes of graphs. There are still numerous algorithms and heuristics that exist which are mostly the variants of the algorithms presented here. On a different note there are number of parallel algorithms for finding Hamiltonian circuits in graphs. There are procedures related with artificial intelligence concepts for finding circuits.

3. Heuristic Design

This chapter gives the reader the basic idea behind our heuristic with many examples. Further this chapter is divided into various sections such that each section will be an enhancement to the previous section. Under each section we list out the issues that we faced while testing and our proposed solutions to it.

3.1. Basic Heuristic

Recall that a Hamiltonian cycle is a cycle that starts at a certain vertex and gets back to the initial vertex after visiting all the vertices with the condition that no vertex is visited more than once. We see the resultant large cycle as a union of smaller cycles. We initially try to form a small cycle and incrementally enlarge the cycle by adding smaller cycles to it.

The first step in the heuristic is to form the initial cycle. An arbitrary vertex *s* of the graph is chosen as the starting vertex. Two neighbors of *s*, call them *x* and *y*, are then selected. Having three vertices in hand, now a shortest path is found from vertex *x* to vertex *y* and the initial cycle is completed by extending the path towards the starting vertex *s* at both the ends. Let's call the cycle thus formed as *C*. This cycle *C* has to be expanded until we include all the vertices.

The core idea here is that the heuristic selects two adjacent vertices in the cycle and finds a shortest path between them. We call a vertex *open* if it is already part of *C* and if it has neighbors which are not part of *C*. The cycle expansion function starts at vertex *s*. It finds two adjacent open vertices in *C* (call them *v* and *w*). The cycle expansion function then finds a shortest path between *v* and *w* and grafts it to the cycle. Let's consider the sample graph in

Figure 3.1. The starting vertex *s* is vertex 1 in our sample.

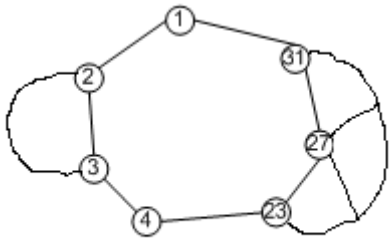


Figure 3.1: Basic heuristic demonstration sample

The initial cycle C is 1-2-3-4-23-27-31-1. The vertices 1 and 4 fall into the category of closed vertices as they don't have any neighbors that are outside C. All the vertices which are not closed are open. The cycle expansion function starts with the first set of adjacent open vertices from starting vertex *s* (in our case it is vertex 1). The heuristic selects vertices 2 and 3 as first set of open vertices and finds a shortest path which expands the cycle as shown in Figure 3.2. Now the vertices 2 and 3 can be closed. This process of selecting first set of adjacent open vertices start from the starting vertex *s* for each cycle expansion step.

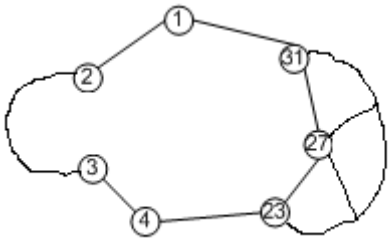


Figure 3.2: Basic heuristic cycle expansion demonstration sample

As discussed in the previous example, the cycle expansion function then scans for two adjacent open vertices from the starting vertex s and tries to expand the cycle through their neighbors. If the function ends up in not finding a path then the second open neighbor can be considered. If there are no open neighbors left then the function skips this vertex and goes to the next vertex for a possible expansion. This function continues till all the vertices are included in the cycle. Refer to Table 3.1 for all the above mentioned rules. On failure, the whole heuristic is repeated from the next starting vertex (here in our case it is another arbitrary choice).

Table 3.1: Expanding the cycle

Let C be s – v – w – x – y – z – s. In selecting two adjacent open vertices there are 3 possibilities 1. s and v are open. Then expand the cycle by finding a shortest path between them 2. s is closed but v is open. Move ahead and select v and w and repeat the same process 3. s is open but v is closed. Move and choose w and x and repeat the same process.

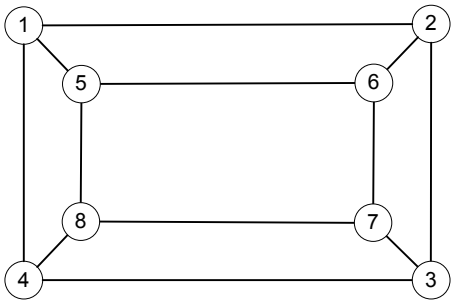


Figure 3.3: Basic heuristic sample

The heuristic can be well explained with another example. Let's consider the sample graph Figure 3.3.

Consider the starting vertex *s* to be 1. Assume that the two neighbors *x* and *y* to be 5 and 4, respectively. A shortest path is computed from vertex 5 to vertex 4 and the cycle is completed by extending the path towards vertex 1 at both ends. Refer to the Figure 3.4. Cycle *C* becomes 1-5-8-4-1. Now the cycle expansion function has to be called, as all the vertices of the graph aren't part of the cycle *C*. Scan from the starting vertex *s*. As vertex 1 has a neighbor 2, which is not part of the cycle, vertex 1 is still considered to be open. On similar lines the next adjacent vertex 5 in the cycle is also open. As we have two adjacent open vertices there is a possibility of expanding the cycle. Consider an arbitrary neighbor of vertex 1 and arbitrary neighbor of vertex 5 which are not part of the cycle. Fortunately both the vertices 1 and 5 have only one neighbor open. So now vertex 2 becomes the new *x* and vertex 6 becomes the new *y*. A shortest path between *x* and *y* is found and the cycle is expanded as shown in Figures 3.5 (a) and 3.5 (b).

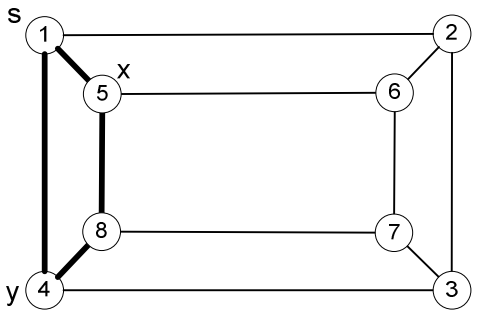
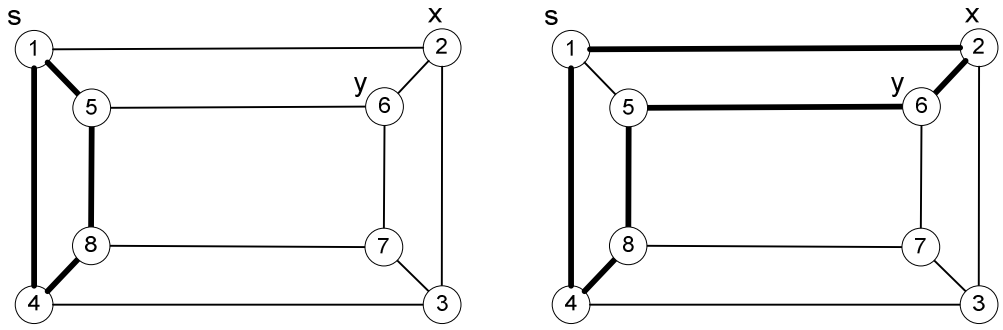


Figure 3.4: Basic Heuristic sample with initial cycle

Now the cycle C gets updated to 1-2-6-5-8-4-1. After each cycle expansion step, all the open vertices are checked to see whether they can be marked closed. In our graph, Vertices 1 and 5 can be closed.

Again as the length of the cycle is not equal to number of vertices in the graph, scan for adjacent open vertices from starting vertex. As the vertex 1 is closed, skip to the next pair of adjacent open vertices. In our case they are vertices 2 and 6. Here the vertices 3 and 7 can be marked as *x* and *y* as they are the only open neighbors of the vertices 2 and 6.



(a) x and y are marked *(b) Cycle expansion and path 1-5 removed*

Figure 3.5: Basic heuristic sample after first cycle expansion

Here the vertices 3 and 7 can be marked as *x* and *y* as they are the only open neighbors of the vertices 2 and 6. Upon finding a path from *x* to *y* and expanding the cycle we get the resultant expanded cycle as 1-2-3-7-6-5-8-4-1 as shown in Figure 3.6 Now the length of the cycle is same as the number of vertices and the heuristic has detected a Hamiltonian cycle.

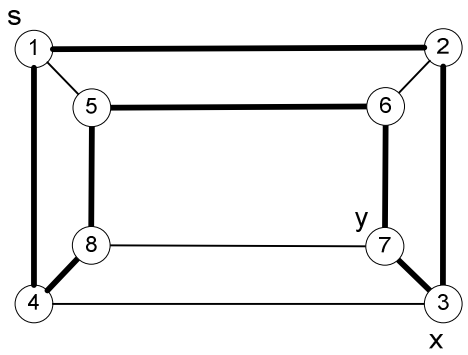


Figure 3.6: Basic heuristic sample after second cycle expansion

The pseudo code of the above described heuristic is given in Figures. 3.7 – 3.9.

```
Input: Input Graph G
Output: Success and print resultant Hamiltonian cycle or failure

HamiltonianSearch (G)
1  check whether the given graph is bi-connected
2  for each vertex s ε G
3        do C← StartCycle(s,G) //start cycle returns a cycle C
4           if |C| == |n|
5                then return success and print cycle
6           else if |C|>1
7                        then C ← CycleExpansion(C,G) // expanded cycle gets returned
8                          if |C|==|n|
9                                  then return success and print cycle
10 return Failure as heuristic is not able to detect Hamiltonian cycle.
```

Figure 3.7: Hamiltonian Search function pseudo code

```
Input: Starting vertex s and input graph G
Output: Cycle C or 0

StartCycle (s, G)
1 for each neighbor x ε s
2       do for each neighbor y ε s and y>x
3             do path ← BFS (x, y, G) //returns the shortest path from x to y
4                 if |path| > 1  //if the length of the path is greater than 1
5                     then   C ← FormInitialCycle(s,path) //Cycle C gets returned
6                            return C
7  return 0; //No Cycle could be formed
```

Figure 3.8: Start Cycle function pseudo code

```
Input: Cycle C and Input Graph G
Output: Expanded Cycle C

CycleExpansion(C,G)
1 while C is not a Hamiltonian cycle do
2       do let v,w be the first edge in C such that neither vertex v nor vertex w is closed
3          for each neighbor x ε v and x not in C
4             do for each neighbor y ε w and y not in C
5                   do  path ← BFS (x,y,G)
6                       if  |path| > 1
7                           then expand the cycle by inserting the path from x to y
8                                break and return the expanded cycle
9 return C //no possibility of expanding the cycle. Hence return the cycle
```

Figure 3.9: Cycle Expansion function pseudo code

Encountered Issues

The heuristic was performing well on random graphs, though the heuristic had to be restarted with different random seeds. There were some random choices made in choosing the start vertex and later sections of this chapter will cover them in great detail. We faced issues when we tried to run the heuristic against the symmetric graphs. Recall that a symmetric graph is a graph which is both edge transitive and vertex transitive. Refer to the chapter 2 section 2.1 for detailed definitions of edge transitivity and vertex transitivity.

Consider the graph in Figure 3.10. Let's see how the heuristic works here in this example. Consider the heuristic starts with vertex 2 as its starting vertex. Vertex 3 and 1 are its neighbors. The shortest path from vertex 1 to vertex 3 leaves us with the initial cycle 2-3-11-10-9-1-2 as shown in Figure 3.11.

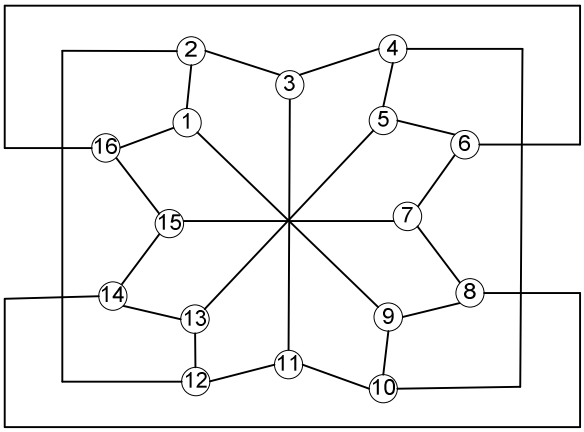


Figure 3.10: Symmetric graph example

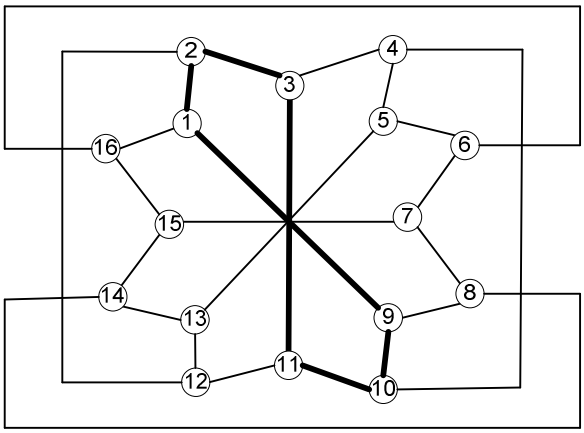


Figure 3.11: Symmetric graph example with initial cycle

Expanding the initial cycle through the open vertices 2 and 3, leaves us with the cycle 2-12-13-5-4-3-11-10-9-1-2 and further expanding on the vertices 13 and 5 yields the cycle 2-12-13-14-15-7-6-5-4-3-11-10-9-1-2. The cycle in the graph looks like in Figure 3.12.

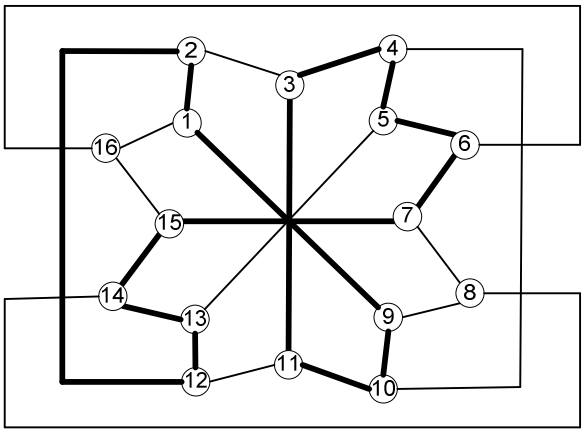


Figure 3.12: Symmetric graph example with final cycle

As you notice, the vertices 16 and 8 are not included. Though there are adjacent open vertices 15 and 7 in the cycle, further expansion is not possible without including the vertices that are

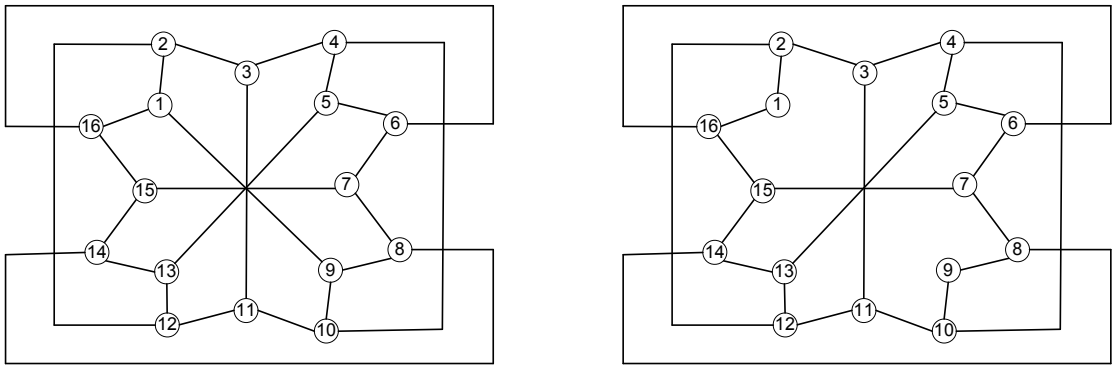
already present in the cycle. Hence we encounter a situation where the heuristic has failed in detecting a Hamiltonian cycle. Even if the heuristic had chosen a different starting vertex and neighbors for the initial cycle, the result would have been the same. The heuristic fails to find a Hamiltonian cycle in this case, though many cycles exist. One such cycle is 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-1.

Proposed Solutions

In order to overcome the symmetric graph issue, we came up with the solution of breaking the graph's symmetry. One of ways to break the graph's symmetry is edge pruning. Our main intuition here is that when we happen to delete an edge we break the symmetry and that helps us the heuristic to visit the vertices which it could not visit before. Our proposed solution worked pretty well and the next section 3.2 deals with this enhancement in more detail.

3.2.Edge Pruning

Consider the example in the Figure 3.10 (reproduced as Figure 3.13 (a) in this section). Choosing which edge to be pruned in entirely random. Figure 3.13(b) shows the same graph with edge from vertex 1 to 9 pruned. By removing this edge we break the symmetry of the graph and it aids in finding the Hamiltonian cycle easily.



(a) Original symmetric graph *(b) Edge pruned graph*

Figure 3.13: Symmetric graph example with edge pruned

As in the previous section, let the heuristic start with vertex 2 and vertices 3 and 1 are its selected neighbors. The initial cycle looks 2-3-4-5-6-16-1-2 looks like in Figure 3.14.

Now vertex 2 has an open neighbor 12 and vertex 3 has an open neighbor 11 which expands the initial cycle to 2-12-11-3-4-5-6-16-1-2. As the vertex 2 is closed, we move on to next set of adjacent open vertices. Vertices 12 and 11 have vertices 13 and 10 respectively. Further expanding the cycle through those vertices leaves us with a cycle 2-12-13-14-8-9-10-11-3-4-5-6-16-1-2 as shown in Figure 3.15.

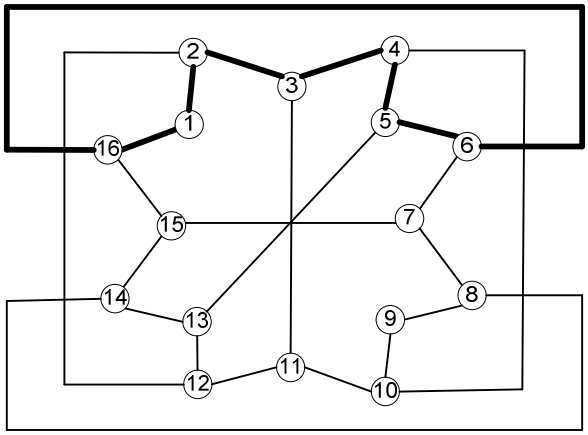


Figure 3.14: Edge pruned graph with initial cycle

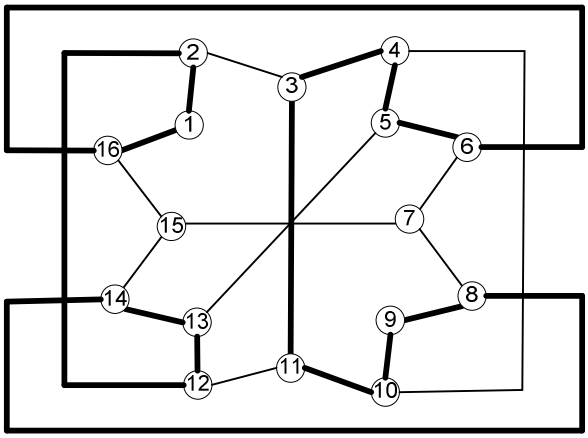


Figure 3.15: Edge pruned graph with expanded cycle

Now vertices 14 and 18 are adjacent open vertices with neighbors 15 and 7 open. Hence the heuristic finds the cycle by expanding through those vertices. The final cycle is 2-12-13-14-15-7-8-9-10-11-3-4-5-6-16-1-2 and it looks like in Figure 3.16.

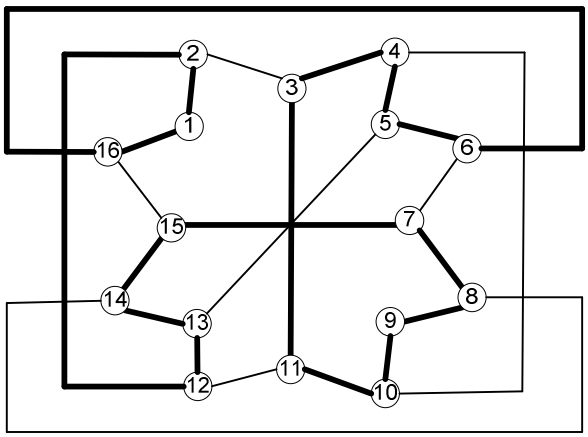


Figure 3.16: Edge pruned graph and the detected Hamiltonian cycle

If the heuristic ends up in an unsuccessful attempt then it prunes an additional edge in the graph and same procedure of attempting to find a Hamiltonian cycle continues. The heuristic further takes care that the graph remains bi-connected even after pruning the edge. This edge pruning continues until the graph loses its bi-connectivity and no further edge could be pruned. The pseudo code of edge pruning procedure is given in the Figure 3.17. The modified version of the "HamiltonianSearch" function is given in Figure 3.18

```
Input: Input Graph G
Output: Edge pruned graph G′

TrialsWithEdgePruning(G)
1  while G is bi-connected
2        Make a working copy of graph G and select an edge e to be pruned
3        G′ ← G - e
4        if  bi-connectivity (G′) = success
5             return G′  //with this graph as input, Hamiltonian Search  function  continues
```

Figure 3.17: Edge pruning function pseudo code

```
Input: Input Graph G
Output: Success and print resultant Hamiltonian cycle or failure

HamiltonianSearch (G)
1  check whether the given graph is bi-connected and make a copy of G
2  Let s be an arbitrary starting vertex
3  While G is not Hamiltonian
4          do C← StartCycle(s,G) //start cycle returns a cycle C
5          if |C| == |n|
6                  then return success and print cycle
7          else if |C|>1
8                  then C ← CycleExpansion(C,G) // expanded cycle gets returned
9          if |C|==|n|
10                 then return success and print cycle
11         else
12                 if (TrialsWithEdgePruning(G) == success)
13                         then continue with the new edge pruned graph
14                 else
15                         restore the original graph and start with a new attempt
```

Figure 3.18: Hamiltonian Search function pseudo code with call to prune edges

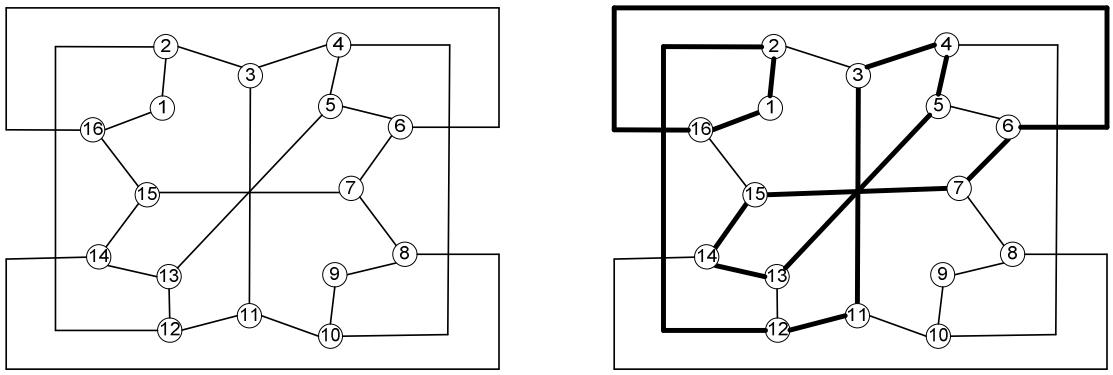
Decisions on Edge Pruning

The heuristic makes couple of major decisions (arbitrarily) which contributes much towards the success rate of finding a Hamiltonian cycle. First and foremost decision is the starting vertex and next critical decision is the selection of an edge that has to be pruned. As we don't determine which edge is better to prune and which starting vertex to use in many cases, we do let the heuristic to arbitrarily make a choice.

Different starting vertices

In our symmetric graph example in the previous section, heuristic did start from vertex 2. Assume that it had started from the vertex 12. With 12 as the starting vertex and 13 and 11 as the neighbors heuristic ends up with a cycle 12-2-1-16-6-7-15-14-13-5-4-3-11-12 as shown

in Figure 3.19 (b). This cycle is not Hamiltonian and the remaining vertices with open neighbors, 7 and 11, are not adjacent in the cycle.

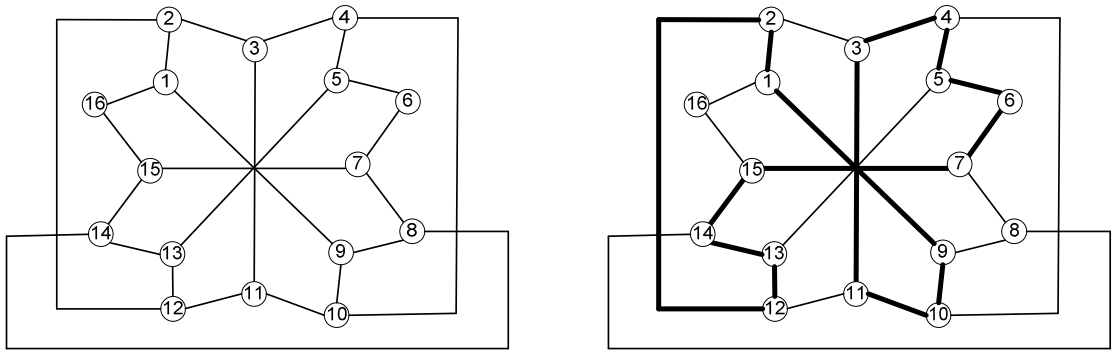


(a) Original graph *(b) unsuccessful attempt*

Figure 3.19: Symmetric graph with starting vertex as 12

Pruning different edges

In our symmetric graph example in the previous section, heuristic did prune the edge from



(a) Original graph *(b) unsuccessful attempt*

Figure 3.20: Symmetric graph with a different edge pruned

vertex 1 to vertex 9. If that wasn't the case, assume our heuristic did prune the edge from vertex 16 to vertex 6. The graph looks like in Figure 3.20(a).Consider that the heuristic started with the vertex 2 as in Figure 3.14. The heuristic ends up with the cycle 2-12-13-14-15-7-6-5-4-3-11-10-9-1-2 as shown in Figure 3.20 (b). Though the heuristic pruned the edge and broke the symmetry of the graph, it wasn't successful in detecting the Hamiltonian cycle. This cycle is not Hamiltonian and the remaining vertices with open neighbors, 8 and 16, are not adjacent in the cycle.

.

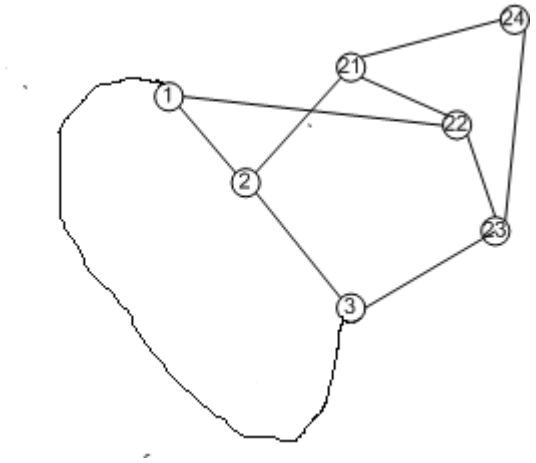


Figure 3.21: Graph to demonstrate isolated vertices

Encountered Issues

The edge pruning proved to be very successful among various instances of symmetric graphs. As discussed in previous sections, edge selection for pruning and starting vertex becomes more critical. The edges in the shortest path become very significant, as the whole heuristic depends on the adjacent open vertices.

Figure 3.21 shows a small part of a bigger graph. Assume that there is a path from vertex 1 to vertex 3 as shown by the hand-drawn line. The vertex 24 is a degree-two vertex as shown in

the Figure 3.21.

Now if the shortest path goes along the path *"- 21-22-23 –"*, then the vertex 24 will never get included; it is an isolated vertex with respect to the cycle. Usually such scenarios happen with edge pruning. The heuristic may convert a vertex into a degree two vertex in course of pruning the edges in the graph. The underlying problem is that both edges incident on vertex 24 must be included in every Hamiltonian cycle, a fact not considered by the heuristic.

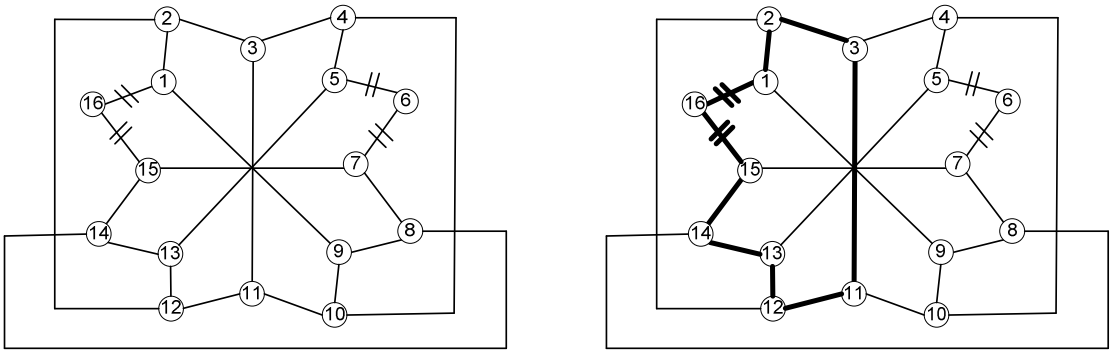
Proposed Solutions

To overcome such issues of isolated vertices, we introduce the concept of *forced edges*. A forced edge is any edge that must be included in every Hamiltonian cycle. Note that there can be only two forced edges for a vertex. The reason behind this is that there can be only two edges for any vertex in the cycle. One is the incoming edge and the other is the outgoing edge. In Figure 3.21 vertex 24 will have couple of forced edges. Vertex 21 and vertex 23 will have one forced edge (to vertex 24). In all future figures we indicate forced edges by crossed lines across the edges. The most critical operation is the propagation of forced edges in the graph and next section deals with them in detail.

3.3.Forced Edges and Propagation of Forced Edges

Consider the symmetric graph in Figure 3.20(a) (reproduced in this section as Figure 3.22(a)). The heuristic didn't successfully detect a Hamiltonian cycle with the starting vertex as 2. Let us see how the concept of forced edges makes it easier for the heuristic to detect the cycle in the same graph with the same starting vertex. Before moving into the example, one other important feature how distance in the shortest paths is handled in the presence of forced

edges. On encountering a forced edge, we traverse through the forced edges until we end up in a vertex which doesn't have any more outgoing forced edges and the distance from the initial vertex to this destination vertex is considered to be one.



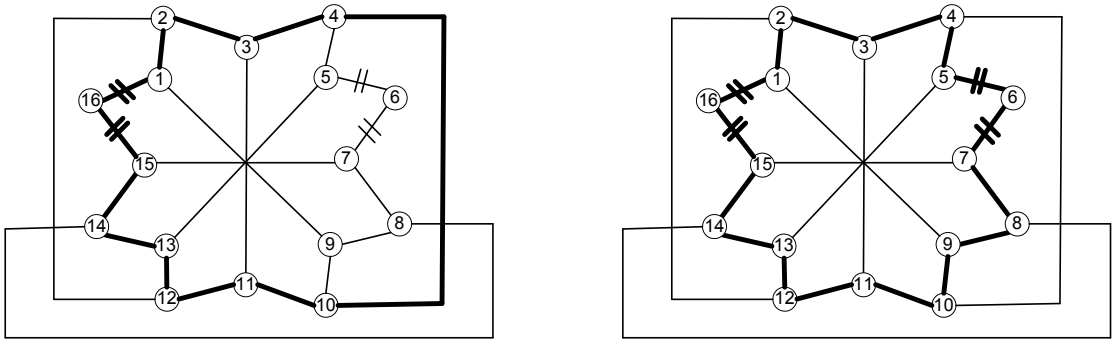
(a) Original graph *(b) initial cycle*

Figure 3.22: Graph with forced edges

With 2 as starting vertex and vertices 3 and 1 as its neighbors, in order, the heuristic finds an initial cycle 2-3-11-12-13-14-15-16-1-2 as shown in Figure 3.22(b). The forced edges are indicated with crossed lines across them. On starting from vertex 1, the heuristic follows the path of forced edges through vertices 16 and 15 and the distance from vertex 1 to vertex 15 is considered to be one or it can be assumed that one hop is necessary to reach from vertex 1 to vertex 15.

With vertex 3 and 11 as the adjacent open vertices, cycle expands to 2-3-4-10-11-12-13-14-15-16-1-2 as shown in Figure 3.23(a). Further expansion with vertices 4 and 10, leads us to a successful detection of Hamiltonian cycle 2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-1-2 as shown in Figure 3.23(b). Here, again, on reaching the vertex 5 the path from 5-6-7 is forced and the distance is considered to be one.

For a graph G, heuristic scans G and classifies the edges as forced and not forced. If there are forced edges then the forced edges have to be propagated through the graph so that appropriate path selections happen while finding shortest paths in G.

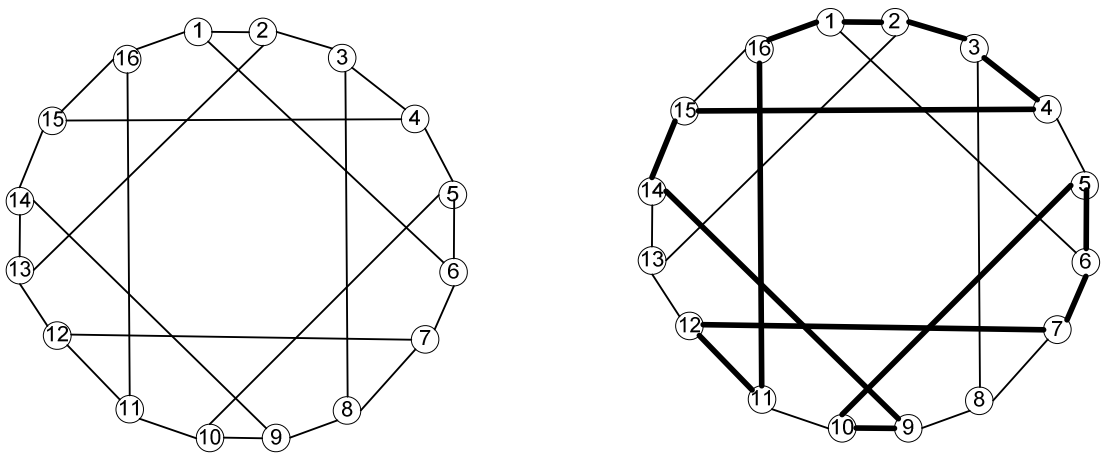


(a) *With expanded cycle* *(b) Successful attempt*

Figure 3.23: Successful detection of Hamiltonian cycle on graph with forced edges

Consider the graph in Figure 3.24(a). Initially the graph G is scanned for forced edges. It tries to find a Hamiltonian cycle without the benefit of having forced edges identified. Figure 3.24(b) shows an attempt made by the heuristic on the base graph G with 6 as starting vertex.

The heuristic makes a single attempt in finding the cycle on the base graph with an arbitrary starting vertex. If it fails, edge pruning function gets called and an arbitrary edge gets removed. Assuming that the edge from vertex 7 to 12 gets removed the graph with forced edges looks like as shown in Figure 3.25. The edges *13-12, 12-11, 7-8* and *6-7* are forced.



(a) Input Graph *(b) Unsuccessful attempt*

Figure 3.24: Failure of heuristic on a graph with start vertex as 6

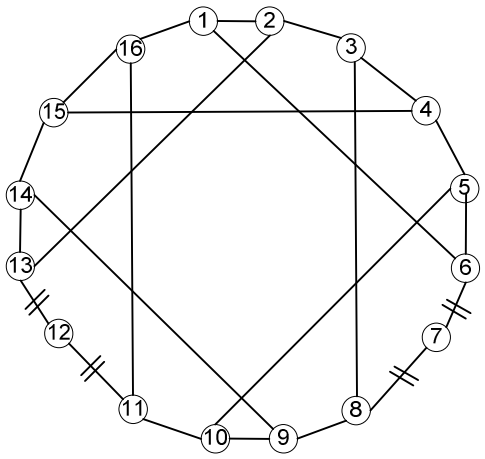


Figure 3.25: Graph with marked forced edges.

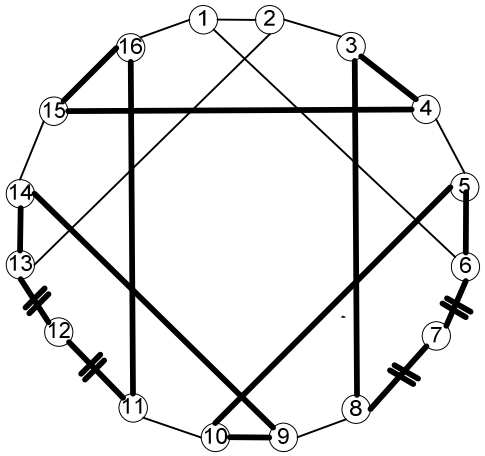
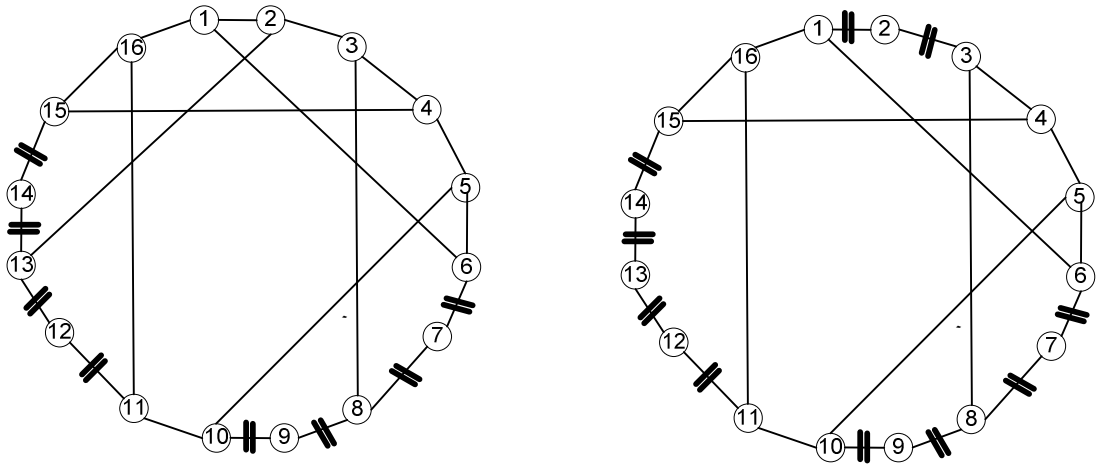


Figure 3.26: Heuristic's attempt with start vertex as 4

Now consider that the heuristic starts with vertex 4 and vertices 5 and 15 are its selected neighbors. Heuristic's attempt on the graph can be seen in Figure 3.26. Again the heuristic fails. Hence further pruning happens and let's consider now the edge 14 to 9 gets removed.

Graph with marked forced edges looks like in Figure 3.27(a). Now propagate forced edges function has to be called which makes the graph look like in Figure 3.27(b). As there are two forced edges for the vertex 13 in Figure 3.27(a) the edge from vertex 13 to vertex 2 can be safely pruned. This makes couple of more edges (edge from vertex 2 to 3 and edge from vertex 2 to 1) to become forced.



(a) Edge 14 to 9 removed *(b) After propagating forced edges*

Figure 3.27: Demonstration of propagation of forced edges

Having done this, heuristic has a better chance of finding the Hamiltonian cycle. The resultant graph after propagating forced edges is shown in Figure 3.27(b). The heuristic's attempt to find the cycle for the graph in Figure 3.27(b) is shown in Figure 3.28. Hence propagation of forced edges becomes very critical in large instances of graphs.

In addition to the "edge pruning" function, the "propagate forced edges" function removes the additional edges and thus making the graph better for the heuristic to detect Hamiltonian cycles. After each edge pruning a bi-connectivity check has to be made, since a graph must be bi-connected to be Hamiltonian.

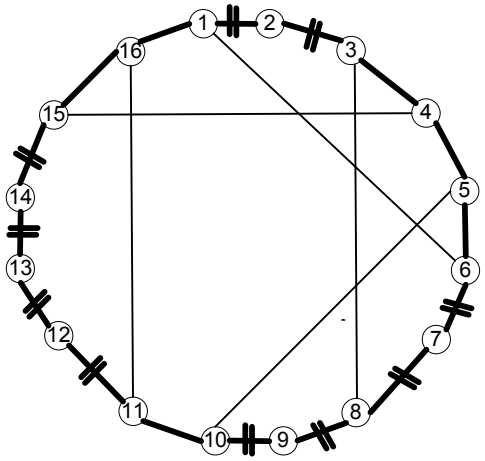


Figure 3.28: Detected Hamiltonian cycle with start vertex as 5 and 6 and 4 as neighbors

The pseudo code for the functions "mark forced edges" and "propagate forced edges" can be seen in Figure 3.29-3.30

Input: Input Graph G
Output: ForcedEdgeList F

MarkForcedEdges(G)
1 for each x ε G and degree[x] =2 do
2 Add both the edges to the ForcedEdgeList F
3 PropagateForcedEdges(G,F) // propagates the edges and update F
4 return ForcedEdgeList F

Figure 3.29: Mark forced edges function pseudo code

```
Input: Input Graph G, ForcedEdgeList F
Output: UpdatedForcedEdgeList

PropagateForcedEdges(G,F)
1  while no forced deletions do
2     for each x ε G and have two forced edges
3         check for extra edges and delete //forced edge deletion
4.        if (bi-connectivity(G)= success) then
5.              mark new forced edges and add to F
6         else
7               break and return "edge deletion not possible"
8    return F
```

Figure 3.30: Propagate forced edges function pseudo code

Encountered Issues

The forced edges and propagation of the forced edges produced very good results among various instances of symmetric graphs. When we moved to large instances of symmetric graphs we faced with the issue of dense components. By dense component, we mean a maximal sub graph with high density. Typically a large graph can have many dense components. A pictorial representation of dense component is shown in the Figure 3.31.

There are three dense components (between vertices 1 and 2, between vertices 28 and 15 and between vertices 172 and 25) in the graph. There can be any number of vertices in each component (if symmetric, all the components will have same number of vertices). As you can see in the first component, between vertex 1 and 2, there are hardly three outgoing edges. Though the heuristic is now equipped with edge pruning and forced edge propagation techniques it could not detect Hamiltonian cycles in graphs which had multiple dense components.

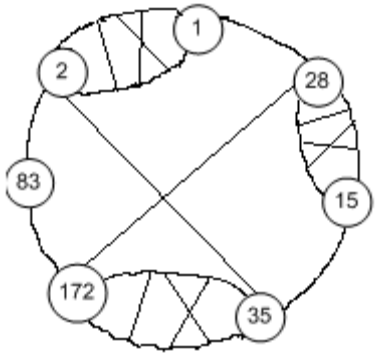


Figure 3.31: Dense components sample

Proposed Solutions

The main challenge lies in analyzing the graph structure, if the heuristic fails to give the result. When we moved to large instances, our heuristic failed in many graphs which led to

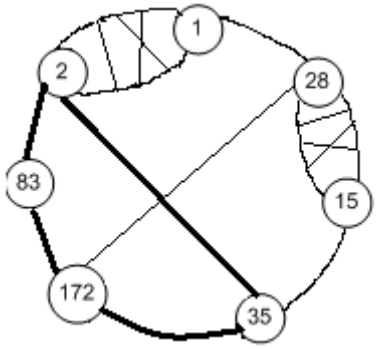


Figure 3.32: Dense components sample with partial cycle

our discovery of dense component structures in the graph. We started analyzing the partial cycles which the heuristic managed to find. To visualize it better let's again consider the Figure 3.31.

The heuristic returned a cycle "2-83-172-….- 35-2" as in Figure 3.32 for the graph with structure similar to that of Figure 3.31. The cycle had less number of vertices in comparison with total number of vertices of the graph. When we analyzed, we could recognize the existence of symmetric components (we call them as dense components as introduced in the previous sections). Here in our sample in Figure 3.32 heuristic was able to break only one such component. Graph with many dense components is equivalent to many small symmetric graphs bi-connected among them. As a rule of thumb, the heuristic has to break all symmetric components to detect the Hamiltonian cycle.

3.4.Farthest Edge Removal

We came up with the *farthest edge removal* technique to solve the issue concerning with dense components. The edge pruning code takes care of breaking symmetry in graphs. In order to break many dense components in one graph, edge pruning code was tweaked to remove edges which were far from each other. The farthest edge removal technique appears to break almost all the dense components. The rate of finding Hamiltonian cycle is directly proportional to the number of the dense components, the edge pruning code breaks.

The heuristic selects an arbitrary edge to be pruned and makes an attempt to find the Hamiltonian cycle. Upon failure, it selects arbitrarily multiple edges and calculates the distance from the edges that were pruned earlier. The edge, which is farthest from the previously pruned edges, is selected and pruned. This gives us a higher probability that the edge pruned is outside the dense component and this heuristic improvement helps to break many such dense components. Often, once the symmetry gets broken the heuristic will be able to add all the vertices to the cycle. At each step of edge pruning as before the heuristic checks for the bi-connectivity in the graph.

The pseudo code for the farthest edge removal logic is given in the Figure 3.33 and the modified version of the "HamiltonianSearch" function is given in Figure 3.34. Only the lines which have been modified are presented here. Refer to Figure 3.18 for previous full version of this function and replace the line with the lines in Figure 3.34.

Input: Input Graph G, List of previous pruned edges L
Output: Modified Graph G′

FarthestEdgeRemoval(G, PrunedEdgeList L)
1 while G is bi-connected
2 select multiple edges $e_1,e_2 \ldots e_n$ in G
3. find the distance from L to $e_1,e_2 \ldots e_n$ in G
4. distance between two edges vw and xy is the minimum among d(v,x), d(w,x), d(v,y) and d(w,y)
5. select farthest edge e_i
6. G′ = G - e_i
7. if (bi-connectivity(G′)=success)
8 break and return G′
9 else continue the loop

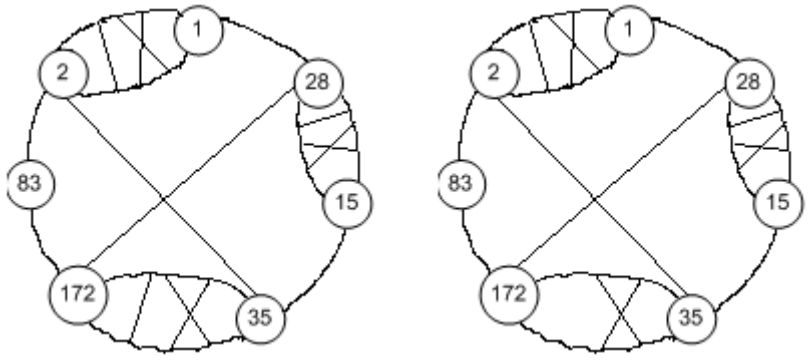
Figure 3.33: Farthest edge removal function pseudo code.

Consider the graph in Figure 3.31 (reproduced as Figure 3.35(a) with more edges inside each dense component). Initially the pruned edge list is empty. When the heuristic removes the first edge, no exclusive check is made. The edge thus pruned, is added to the pruned edge list and now the list contains an edge. Figure 3.35(b) shows the first edge that got pruned in the dense component between vertices 172 and 35.

```
// Refer to Figure 3.18 for the full version. Replace the lines number 12-15 in Figure 3.18
//with the lines below.

12  if (farthestEdgeRemoval(G) == success)
13        then continue with the new edge pruned graph
14  else
15        restore the original graph and start with a new attempt
```

Figure 3.34: Hamiltonian Search function pseudo code with call to farthest edge removal



(a) Original graph *(b) After removing an edge*

Figure 3.35: Farthest edge removal - initial edge pruning

The heuristic makes an attempt to detect the Hamiltonian cycle. On failure, edge pruning functions arbitrarily selects multiple edges in the graph. Figure 3.36 shows the edges selected.

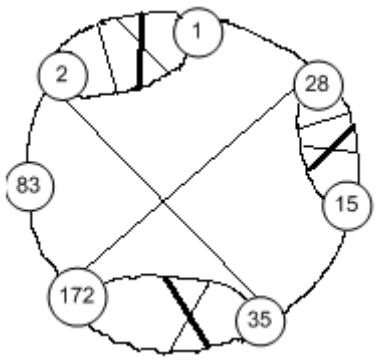


Figure 3.36: Farthest edge removal - edge pruning selections

Here we have three edges selected. The first edge is present in the dense component with vertices 172 and 35. The next one is present in the component with vertices 28 and 15 and the last one is present in the component with vertices 1 and 2. The main idea behind choosing multiple edges is to find the farthest edge and thereby increase the probability of breaking the dense component. The edge pruning list contains an edge in the component with vertices 172 and 35. The distance from the selected edges to the edges in the edge pruning list, are computed.

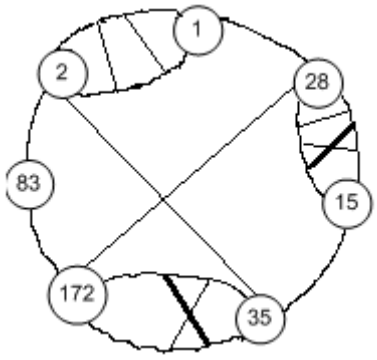


Figure 3.37: Farthest edge removal - couple of edges pruned

Let the farthest edge among the selections is the one in dense component with vertices 2 and

1. Now the edge gets removed and added to the edge pruning list. The edge pruning list now has two edges in it. Figure 3.37 shows the graph with selected edge removed.

This edge removal technique now guarantees of having broken symmetry in couple of dense components. This process continues till the heuristic finds a Hamiltonian cycle. This enhancement of farthest edge removal led to good success among large instances of symmetric graphs.

3.5. Versions of the Heuristic

As described in the previous sections, there are three major versions of our heuristic. The first version of our heuristic, called *basic heuristic*, is devoid of edge pruning. The second version of our heuristic, termed *edge pruning heuristic*, is with edge pruning and forced edge propagation rules. The third version, called *farthest edge removal heuristic*, is with farthest edge removal logic. We created two additional versions of our heuristic. The fourth version, of our heuristic is same as the second edge pruning version except for the fact that the heuristic restarts when the bi-connectivity test fails. Heuristic runs the bi-connectivity test whenever an edge is pruned and forced edges are propagated. The second version upon failure of the bi-connectivity test restores the edge back to the graph and prunes a different edge. This process continues till no further edge could be pruned and then the heuristic restarts with a fresh copy. In the fourth version, we start pruning the edges as we do in second version. Once the bi-connectivity test fails, the heuristic doesn't try a different edge. It rather restarts with a fresh copy. The rationale behind this is the fact that as edges are chosen at random for pruning there could be a chance that the heuristic failed to prune the correct edge. Hence when the bi-connectivity fails, heuristic restarts with a fresh copy. Similar to this fourth version, the fifth version of our heuristic is with farthest edge removal code and restart on bi-connectivity failure. As a peer to the farthest edge removal heuristic,

the fifth version differs from the third version from the fact that it restarts in bi-connectivity failure.

3.6.Time Complexity

On having discussed various techniques, it becomes very critical to discuss the time complexity of the heuristic. This heuristic requires the input graph in the memory for efficient processing of edge pruning and forced edge propagation enhancements.

Breadth first search (BFS) is the most critical operation in our heuristic. BFS takes O(m) where m is number of edges. Basic algorithm calls the BFS at most n times in the worst case when each call to BFS expands the cycle by a vertex. This basic algorithm with calls to BFS at O(mn) dominates all the other functions like forced edge selections, farthest edge removal and bi-connectivity tests (since each of these is O(m) and done only one per run of the basic algorithm). Hence per trial we have an O(mn) algorithm, but given that the input graphs are sparse, this is $O(n^2)$ for Cayley and knight tour graphs and $O(n^2 \log n)$ for the random and geometric graphs. The number of trials that we make depends on number of edges we prune and re-run the basic algorithm. An arbitrary choice of removing at most n/4 edges worked well for our heuristic.

Hence the time complexity $O(n^3)$ for Cayley and $O(n^3 \log n)$ for random and geometric graphs. Though the worst case time complexity is $O(n^3)$ and $O(n^3 \log n)$, our experiments suggest that it is close to $O(n^2)$.

3.7.Conclusion

In this chapter we proposed a base algorithm for detecting Hamiltonian cycle. This chapter discussed the issues around the base algorithm in detail and the necessity of edge pruning to break symmetry in symmetric graphs. Later this chapter introduced the notion of forced edges and propagation of forced edges to gain success in various instances of symmetric graphs. Then we dealt with a potential problem of dense components in larger instances of symmetric graphs. Finally we introduced the notion of farthest edge removal and the intelligent edge pruning (though edges were selected arbitrarily) technique to break the dense components. This step by step evolution of the base algorithm made our life easier as we progressed from smaller instances of symmetric graphs to larger instances.

4. Classes of Graphs

This chapter deals with the various classes of graphs which were considered for our experiments to evaluate the performance of our heuristic. The chapter is divided into 4 major sections covering cubic Cayley graphs, random graphs, knight tour and geometric graphs. Each section begins with a brief overview regarding the class of the graph, method of generation and its important properties regarding Hamiltonicity.

4.1. Cubic Cayley graphs

Cubic graphs are the graphs with all of its vertices having a degree of exactly 3. They are also called trivalent graphs or 3-regular graphs. Although it is one of the simplest classes of graphs, many graph theoretic problems remain NP-hard for the special case of cubic graphs. Cubic graphs were first studied in late 1800's and interested readers can refer to work by Greenlaw and Petreschi (Greenlaw & Petreschi, 1995) to know about the history and important properties that hold on cubic graphs. Refer to work by Robinson and Wormald (Robinson & Wormald, 1992) who have proved that almost all (as n tends to ∞) cubic graphs are Hamiltonian.

Symmetric graphs are those graphs which are both vertex transitive and edge transitive. Refer to Section 2.1 for basic definitions regarding transitivity. A cubic symmetric graph is often referred to as a symmetric cube. Symmetric cubes were first studied by Foster in 1932 (Foster, 1932). Since all cubic graphs must have even number of vertices, cubic symmetric graphs also have the same property.

Cayley graphs are named after Arthur Cayley, who discovered them. He was also the first person to formally define the notion of groups in abstract algebra. Cayley graphs are often used as a model for developing interconnection networks. They were motivated by a theorem of Arthur Cayley. Every group G is isomorphic to a subgroup of the symmetric group on G. Readers can refer to the text (Malik, Mordeson, & Sen, 1997) for the fundamentals in

abstract algebra.

Definition

A symmetric group S_n of degree n is the group of all permutations on n symbols. Hence S_n is a permutation group on $n!$, elements.

Let A be a finite group, and let S be a non-empty, finite subset of A such that S doesn't contain the identity element. Assume S to be symmetric, i.e., $S = S^{-1}$.Then Cayley graph the A(G, S) is the graph with the vertex set V ε G and edge set E = {{x,y} : x,y in G, there exist s in S with y = xs}. Here the set S is called the generating set.

Let's see an example of how Cayley graphs are generated. Cayley graphs are generated by a set of generator permutations. By mutually permuting the elements (permutations are done until no new elements can be formed) we get a closed set S. Suppose S, a subset of S_4, consists of generator permutations g1=1243; g2 = 1432; g3=1324. As there are 4 elements in each set, the maximum possible distinct elements can be 24 (4!).

Generator permutations are applied to get a closed set S. For example, possible permutations of 1243 are as follows.

g1 applied to 1243 yields 1234

g2 applied to 1243 yields 1342

g3 applied to 1243 yields 1423

Similarly by applying all permutations for all distinct elements we get 1234, 1243, 1432, 1324, 1342, 1423. These elements form the set S. As there are 6 elements in this set the number of vertices in this graph will be 6. To draw the edges, consider the first element 1234. It's possible permutations were 1243, 1432 and 1324. Hence we draw three edges to these vertices. Similarly the whole graph is built.

Figure 4.1 shows the Cayley graph for this example.

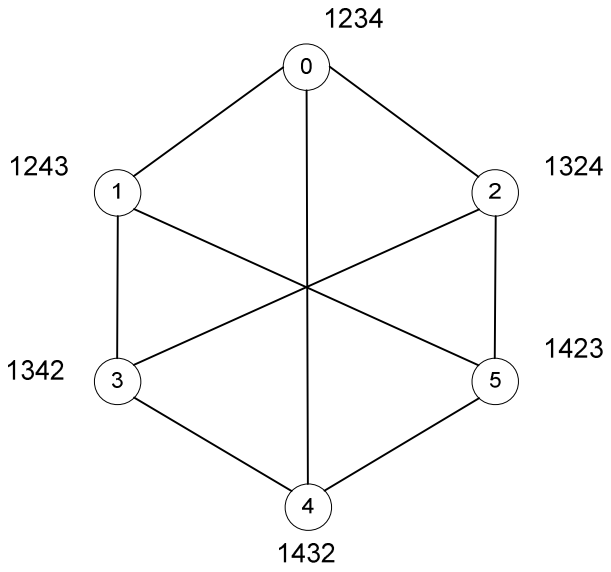


Figure 4.1: Cayley graph

Properties of Cayley graphs and important results

Readers can refer to the paper by Cooperman and Finkelstein (Cooperman & Finkelstein, 1992) for interesting properties of Cayley graphs and various methods for generating them efficiently.

All Cayley graphs are vertex transitive but the inverse doesn't hold. The Peterson graph, shown in Figure 4.2, is a 3- regular graph with 10 nodes and 15 edges. This graph serves as the smallest counter example to prove the fact that all vertex transitive graphs are not Cayley graphs. Refer to paper by McKay and Praeger (McKay & Praeger, 1994) to know more about the graphs which are vertex transitive but not Cayley.

Lovasz (Lovász, 1970), in 1970, conjectured that every finite vertex transitive graph has a Hamiltonian cycle. This conjecture is still open, i.e., no formal proof exists. Later in 1996,

Jixiang and Qiongxiang (Jixiang & Qiongxiang, 1996) have proved that almost all Cayley graphs are Hamiltonian. That is, as n, the order of the graphs, approaches infinity, the ratio of the number of Harniltonian Cayley graphs to the total number of Cayley graphs approaches 1.

A transposition is a function which swaps any two elements of a set. In other words, it is a bijective function. Elements of the symmetric group S_n can be generated by a minimal generating set of n-1 transpositions, called a basis. Kompel'maher and Liskovec (Kompel'maher & Liskovec, 1975) proved the fact that Cayley graph (S_n:B) is Hamiltonian for any basis B consisting of transpositions.

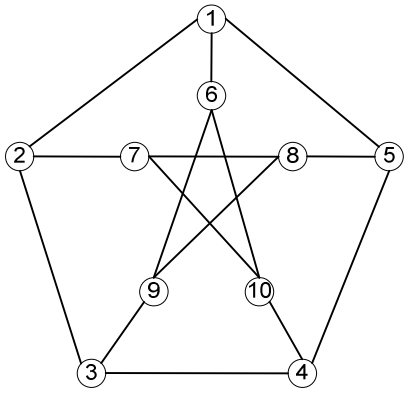


Figure 4.2: Petersen Graph

Witte (Witte & Keating, 1985) proved the conjecture, "Every connected Cayley graph (apart from k_2) has a Hamiltonian Cycle", for *p*- groups. A finite group is a p-group if and only if its order (the number of its elements) is a power of p. Further Pak and Radocic (Pak & Radoicic, 2004) proved that every group G has a generating set of size $\leq \log_2(|G|)$ for which the corresponding Cayley graph is Hamiltonian. Krivelevich and Sudakov (Krivelevich & Sudakov, 2003) showed that taking a random set of $\log^5(|G|)$ elements of G as generators, almost surely yields a Hamiltonian graph.

Ruskey and Savage (Ruskey & Savage, 1993) further enhanced this result, discovering Hamiltonian cycles in Cayley graphs by extending transposition matchings to a cycle. Let X be a generating set of transpositions for the S_n where n>4. For any xεX, M_x extends to a Hamiltonian cycle in the graph (S_n:X)

Generation of Cayley graphs for our experiments

Effler and Ruskey (Effler & Ruskey, 2000) worked in early 2000 on Cayley graphs. They generated the graphs with 3 involutions. An involution, σ, is an element of S_n such that $\sigma^2 = 1$ (element of order 2). They proved that most of the graphs with generating set S_7 were Hamiltonian.

Later Shields and Savage (Shields, 2004) continued this work and proved the fact that all the graphs in the generating set S_7 were Hamiltonian. Further Shields enhanced the idea to generate graphs with one-involution and one non-involution. Most of the cubic Cayley graphs that were used for our experiments were from Shield's work.

Chapter 5 gives details about our experiments and results on this class of graphs.

4.2.Random Graphs

Random graphs were first introduced and defined by Erdos and Renyi (Erdős & Rényi, 1959) in 1959. According to them a random graph is a graph which can be generated by some random process. With n vertices in hand adding edges at random (based on some probability distribution) produces different sets of random graphs. To confine ourselves into a specific random model, we will look into the most common model.

Definition

Let's call the model G(n,p), where p, 0<p<1 stands for probability of occurrence of an edge. In this model each edge in the graph are chosen independently with a probability p. Expected number of edges will be C(n,2)p. Each specific graph with *n* vertices and *m* edges has a probability $P(G) = p^m . (1-p)^{C(n,2)-m}$. All throughout this discussion on random graphs, we always refer to G(n,p) model and we assume that the expected number of edges is *m*.

Properties of random graphs and important results

Interesting questions in this class of graphs are as follows. Are all random graphs Hamiltonian? The answer to this question is a resounding no. Since all the edges and elements are associated with some probability there is always a chance of graph being disconnected (existence of isolated vertices) or not being bi-connected (existence of cut vertices). Our next question is: When can we say that a random graph is Hamiltonian? To formally state this question, what is the value for *n*, the number of vertices, and what is the expected number of edges *m* required, to ensure that almost all random graphs are Hamiltonian? There were various conjectures to answer this question. In 1974, Pósa (Pósa, 1976) put forth that if a graph has *n* vertices and if it has at least cn log n edges then the graph is Hamiltonian. Here c is some sufficiently large constant. Korshunov in 1976 (Korsunov, 1976) came up with a proof stating that the graph should have at least ½ n log n edges to be Hamiltonian. Many extensions were made on this result. Readers can refer to the text by Bollobas (Bollabas B. , 2004) for interesting results and properties of random graphs.

The most important fact about random graphs was stated and proved by Komlos and Szermedi (J.Komlos & Szemeredi, 1983) and later Korshunov (Korshunov, 1985) also proved the same condition. The theorem goes like this.

Let ω(n) be some function tending to ∞. Let the number of edges be represent by a function M(n). If M(n) = (n/2) (log n+ log log n + ω(n)), then almost all G(n,p) are Hamiltonian. Reader can refer to the paper by Bollabas (Bollabas B. , Almost all regular graphs are

Hamiltonian, 1983) for a detailed proof of the above theorem.

Generation of random graphs for our experiments

Vandegriend (Vandegriend, 1998) in his master's thesis worked on various algorithms starting from naïve backtracking to the various variants of Pósa's path rotation algorithm. He generated different classes of graphs for his experiments up to 1600 vertices. We modified his code so that it can generate larger graphs which can be used as inputs to our heuristic. It was ensured that the G(n,p) graphs had at least as many edges as suggested by the theorem, so that the graph had a high probability of being Hamiltonian.

All the random graphs for our experiments were from Vandegriend's code in Skiena's repository (Skiena, 1998).

4.3.Knight tour graphs

A knight's tour is the path taken by a knight in the game of chess. The knight, starting at an arbitrary square, must move on an empty chess board so that it visits all the squares exactly once and returns back to the starting square. The analogy here is that, each square on a chessboard can be imagined as a vertex in the graph and visiting all squares once and returning back to the starting square is same as visiting all vertices exactly once and returning back to the starting vertex. Moving from a square to other square is same as that of travelling through an edge from one vertex to another vertex. This is a Hamiltonian circuit problem of our interest.

In a typical chessboard (8 by 8) there are 64 squares. A knight from a particular position can take any of the possible 8 moves. In the Figure 4.3, k corresponds to the current position of the night and moves are numbered from 1 till 8. However at some positions (towards the

board boundaries) knight may not have all the possible 8 moves.

The motivation behind considering knight's tour for our experiments is that we wanted to try our heuristic against various classes of graphs were the Hamiltonian cycles were known and measure the performance of our heuristic.

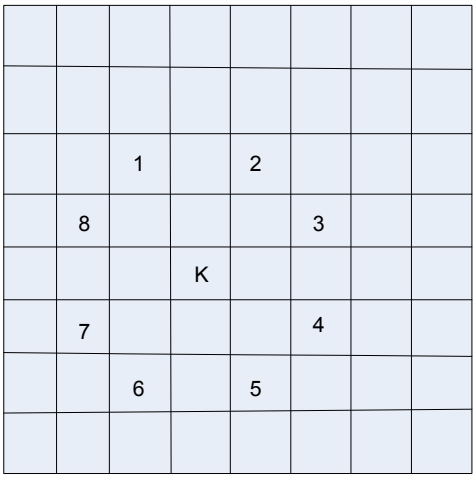


Figure 4.3: Possible moves for a knight in a chessboard

Definition

The knight's tour problem can be translated formally into a specific instance as defined below.

Consider each square on a chessboard to be a vertex. Then in an *n* by *m* chessboard there can be *mn* vertices. An edge from vertex *x* to vertex *y* can be added in the graph if a knight can move from the square *x* to the square *y* following all the rules of chess. Given all these, our Hamiltonian circuit problem is nothing but to find a circuit in the chess board of given size. Figure 4.4 shows a sample knight tour (Hamiltonian circuit) on an 8 by 8 standard chess board.

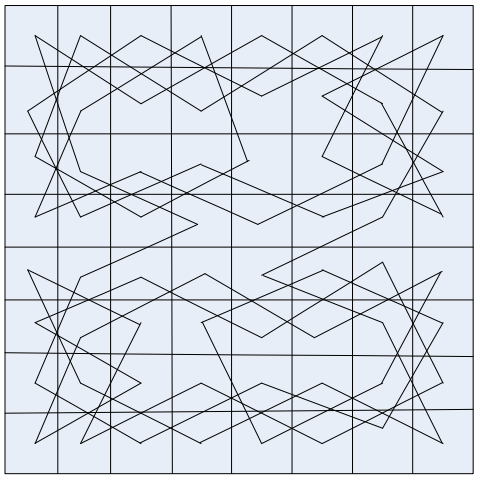


Figure 4.4: Hamiltonian Circuit in an 8 by 8 standard chess board

Properties of knight-tour graphs and important results

The most general question here is: When can we conclude that the given board cannot have a Hamiltonian circuit? If the *n* by *m* chess board contains an odd number of squares then it cannot contain a circuit. A 4 by m board cannot have a circuit for any value of m. A knight tour must typically alternate between the white and black squares of the chess board. Schwenk (Schwenk, 1991) has found all the possible combinations of *m* and *n* for which the Hamiltonian circuit cannot exist and proved that for all other values there does exist a Hamiltonian circuit.

Readers can refer to the work Vandegriend (Vandegriend, 1998) for a detailed proof of many of these properties. In the paper (Raptopoulos & Spirakis, 2005) by Raptopoulos and Spirakis, they talk about the various classes of random graphs and efficient greedy methods to detect Hamiltonian cycles in knight-tour graphs.

Generation of knight-tour graphs for our experiments

Again Vandegriend's code from Skienna's repository (Skiena, 1998) was used in generating knight tour graphs. Graphs of various board sizes (more than 1000 vertices) were generated and we ran experiments against those graphs.

4.4.Geometric graphs

Geometric graphs are graphs based on points in some geometric objects with edges connecting points that are nearby.

Definition

A geometric graph G (n, r) is constructed by choosing n points uniformly at random in a unit square $[0, 1]^2$ and adding edges to connect any two points which are at a distance of at most r from each other. In other words, G (n, r) is a graph that has a vertex set V and an edge connecting each pair of vertices u and v at distance d (u, v) $\leq$ r; where d denotes a distance metric.

Readers can refer to (Diaz, Mitsche, & Perez, 2007) for a detailed definition and description of geometric graphs. Geometric graphs depend on the kind of distance metric that is being used to generate them. There are two important norms to measure distances.

Under L_p norm,

the distance $||x-y||$ between two points $x = (x_1, y_1)$ and $y = (x_2, y_2)$ is $(|x_1-x_2|^p + |y_1-y_2|^p)^{1/p}$.

In L_∞ norm, the distance between x and y is max $\{|x_1-x_2|, |y_1-y_2|\}$.

In order for the geometric graph G(n,r) to be connected, the distance r has to be at least $\sqrt{\log (n) / 4 (n)}$ (Appel & Russo, 2002). Diaz, Mitsche and Perez (Diaz, Mitsche, & Perez, 2007) proved that when $r = \omega (\sqrt[4]{(\log n/\pi n)})$, G(n,r) becomes Hamiltonian.

For a long time, geometric graphs have been used as models for large autonomous networks such as sensor networks. In recent times, they have received attention in ad-hoc wireless networks. Refer to Penrose (Penrose, Diaz, Petit, & Serna, 2001) for a detailed history and to know interesting facts about this class of graphs.

Generation of geometric graphs for our experiments

Geometric graphs were generated with a distance parameter r based on the definitions and theorems in the previous sections. Vandegriend's code was used to generate the graphs of various sizes for our experiments.

4.5. Conclusion

In this chapter we discussed various classes of graphs and their method of generation. Most importantly we reviewed various important theorems regarding the Hamiltonicity in those classes of graphs. In the next chapter we will document all the results of our heuristic on these classes of graphs.

5. Experimental Results

This chapter deals with all the experiments that we have done and summarizes all the results. We begin by talking about the various configuration settings in our heuristic and machine configuration on which we ran the experiments. Chapter 3 dealt with various variations of our heuristic namely the base heuristic; edge pruning heuristic and farthest edge pruning heuristic. This chapter presents the results for various classes of graphs against different versions of our heuristic. This chapter also gives a comparison chart of our heuristic against the various existing popular heuristics.

5.1.Heuristic Settings and Machine Configuration

As described in the Section 3.5, there are five versions of our heuristic. Before getting into the experimental results we would like to make the terminology and heuristic settings clear for the readers.

Attempt: When the heuristic restarts with a fresh copy of the graph we call that as an attempt.

Trial: Initially the heuristic tries to execute the basic algorithm and on failure, depending on the version of the heuristic, it might either execute normal edge pruning or farthest edge removal or nothing. After pruning an edge it tries to execute the basic algorithm again. Trial is nothing but the heuristic's execution of the basic algorithm. In one attempt there may be many trials. If no edge pruning takes place each attempt has only one trial.

***Average Search Length per Node*:** There may be many BFS searches in a trial. Each time the heuristic tries to build the initial cycle or expand the cycle a breadth first search is performed. We record the total length of the searches and number of such searches. Average

search length is total length of searches / number of searches. Average search length per node is the average search length / number of nodes.

Time per Node (in seconds): Time per Node is the statistic recorded to see total time spent on each vertex. Time per Node = total execution time/ number of vertices.

Length of Cycle (in percentage): As the name implies it is the length of cycle formed by the heuristic as a percentage of maximum possible, i.e., a Hamiltonian cycle of length n. This statistic gives a clear idea of about the performance of individual versions of the heuristic.

Heuristic Options:

As our heuristic may run indefinitely until it finds a Hamiltonian circuit, we set various conditions for termination. It should be noted that the code wasn't tuned for giving optimal performance. Number of attempts and time are the important parameters set before running the heuristic. Many of our experiments, on various classes of graphs, were given a run time of 15 minutes to find the cycle. After the specified time limit the heuristic quits. We realized that this 15 minute time limit was never enough for the heuristic when dealing with larger instances of problem (>2500 vertices) on normal machines. Hence we ran them for 16 hours. Refer to the next paragraph for machine configurations. As there are many random choices to be made, the seed can be set as a parameter to the heuristic. If the seed is not provided, the algorithm takes the current time as the seed and generates random choices based on it.

Machine Configuration:

The smaller instances of the problems (<2000 vertices) were run on an Enterprise Red Hat Linux platform with two Intel(R) Pentium(R) 4 CPU 3.00GHz processors and a cache memory of 1024KB. The larger instances of the problem (>2000 vertices) were run in batch mode on machines of the NC State HPC (High Performance Computing) cluster. These have two Intel(R) Xeon(TM) CPU 2.80GHz processors with a cache size of 512 KB. Note that the 'edge pruning' and 'farthest edge removal' versions were run on HPC machine whereas the 'basic heuristic' version wasn't.

5.2.Cayley Graphs

A substantial part of the research was devoted to cubic graphs, specifically Cayley graphs. The motivation behind this is that cubic graphs are the most difficult ones to solve. Initially, we tried to run our base heuristic on small instances (<150 vertices) of cubic graphs. The base heuristic was successful against the small instances but it failed to find cycles in larger graphs.

All cubic Cayley graphs generated from three involutions were used for our experiments. The Table 5.1 gives the results of our base heuristic on various instances of graphs. Smaller instances were given a time out of 15 minutes. Larger instances (indicated by * against the vertices) were run on the HPC machine.

As you can see from Table 5.1, basic version doesn't find cycles even for the instances of graphs with vertices less than 500. The average length of the cycle which the base heuristic managed to find was around 85-90 %. The edge pruning version of our heuristic was more successful in terms of statistics. It was highly successful on instances <2500 vertices. On the other hand edge pruning with restart on bi-connectivity failure didn't perform as expected. More or less the results its results were similar to basic heuristic. Hence we didn't run that heuristic against the larger instances. Farthest edge removal heuristic performance was almost equal to farthest edge removal with restart heuristic.

Table 5.1: Performance of our heuristic on Cayley graphs

Graph No	Vertices	Length of the cycle detected in %				
		Basic Heuristic	Edge pruning	Farthest edge removal	Edge Pruning with restart	Farthest edge removal with restart
3007	336	100.0	100.0	100.0	100.0	100.0
119	360	93.3	100.0	100.0	88.6	100.0
636	360	88.6	100.0	100.0	93.3	100.0
1073	576	89.9	100.0	100.0	89.9	100.0
1088	576	100.0	100.0	100.0	100.0	100.0
155	720	90.0	100.0	100.0	90.0	100.0
2517	720	86.1	100.0	100.0	86.1	100.0
225	1152	89.2	100.0	100.0	95.0	100.0
2949	1152	96.2	92.4	94.1	88.2	94.1
2109	1344	92.1	95.7	91.5	92.2	91.5
814	1344	85.4	100.0	100.0	85.7	100.0
217	1440	87.9	100.0	99.0	86.6	99.0
338	2520	85.0	100.0	87.1	85.4	87.1
833*	2520	92.2	91.4	91.9		91.7
2222*	5040	87.5	87.5	87.2		87.2
2269*	5040	85.8	87.3	85.5		85.4
621*	20160	79.0	78.6	78.7		78.5
928*	20160	84.2	83.9	84.2		85.9

* indicates the runs made on HPC Machines

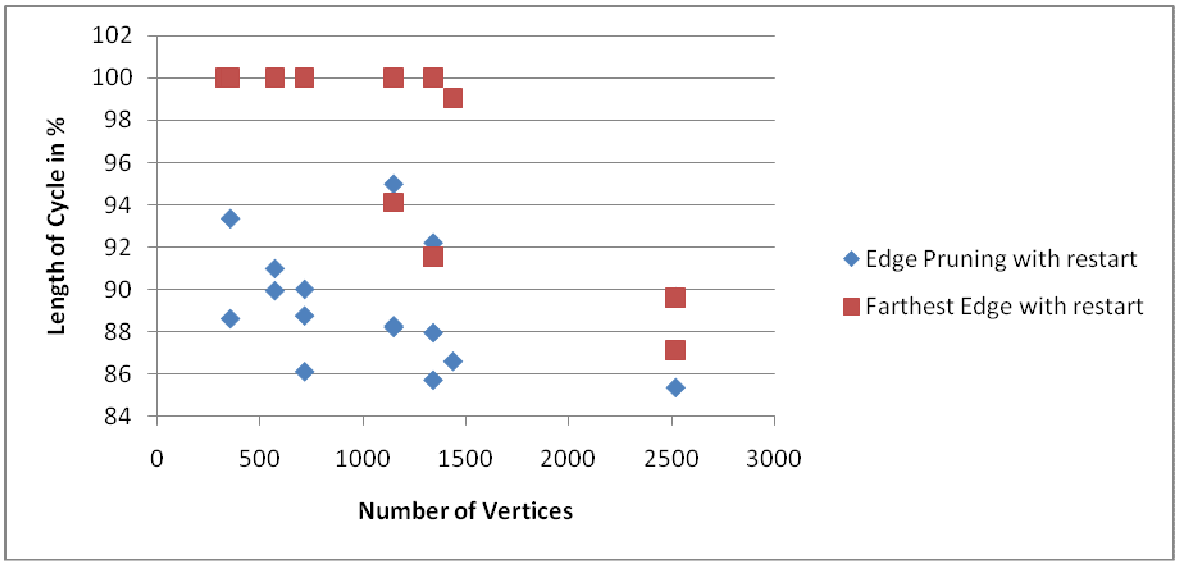


Figure 5.1: Edge Pruning vs Farthest Edge (both with bi-connectivity restarts)

Figure 5.1 shows the comparison between the edge pruning heuristic with restart against farthest edge pruning heuristic with restart. Though there isn't much difference between edge pruing and farthest edge heuristic, there is a huge difference here in Figure 5.1.

We wanted to evaluate edge pruning and farthest edge with restart on the basis of number of attempts they took to find the cycle. As their performance in terms of detecting cycles is almost the same, the comparison on number of attempts would be more interesting. Figure 5.2 shows the comparison on basis of number of attempts. We can clearly see that edge pruning heuristic performs much better than the farthest edge with restarts.

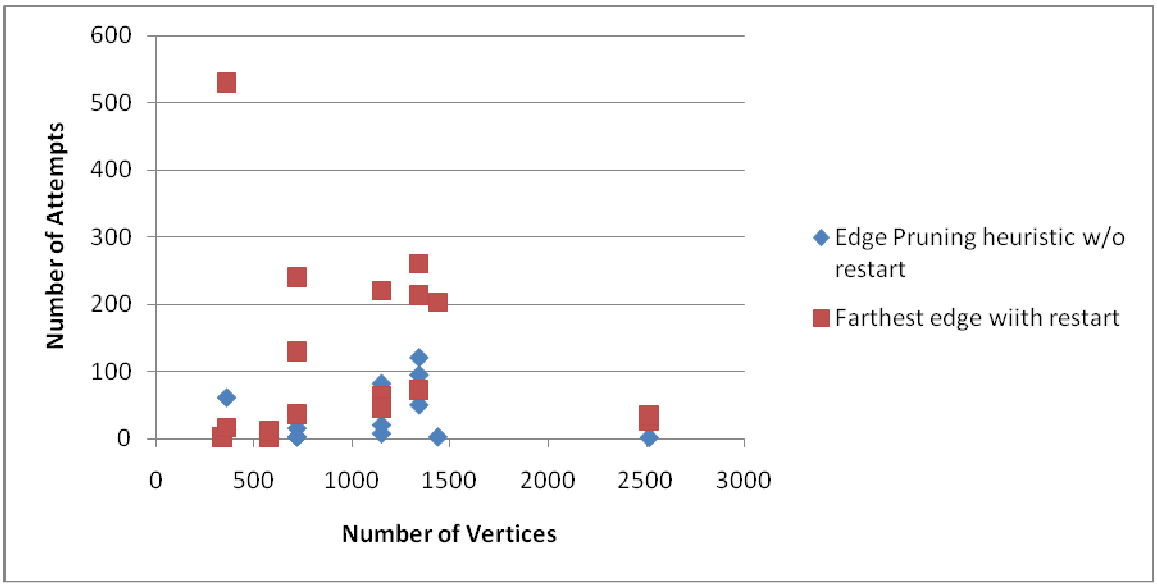


Figure 5.2: Edge pruning w/o restart vs Farthest edge with bi-connectivity restart

As our heuristic is highly randomized, none of the results follow a particular pattern which you would normally expect from a deterministic algorithm. The results depend much on the random edge pruning choices that are made initially. Forced edge propagation eliminates many bad choices and makes it more convenient for the heuristic. But if the case of bad initial choices, the heuristic would waste much of its time in further pruning the edges and attempting to find cycles in the graph. Refer to the chapter 6, future work section we have discussed much towards the edge pruning choices made.

5.3.Random Graphs

As described in Chapter 4, we generated random graphs with the help of Vandegriend's code. The expected number of edges was ensured to be greater than the threshold given by the function $(n/2)\{ \log n + \log \log n + \omega(n) \}$ where $\omega(n)$ is some function tending to infinity.

Table 5.2: Performance of our heuristic on random graphs

Graph No	Vertices	Length of the cycle detected in %			
		Basic Heuristic	Edge pruning	Farthest edge removal	Farthest edge with restart
R_200_1.3_1	200	99.0	100.0	100.0	98.0
R_200_1.3_3	200	98.0	100.0	100.0	98.5
R_300_1.3_1	300	97.3	100.0	100.0	96.7
R_300_1.3_3	300	97.3	100.0	100.0	96.7
R_500_1.3_1	500	94.8	100.0	100.0	94.4
R_500_1.3_2	500	95.4	99.6	93.8	94.2
R_600_1.3_1	600	95.7	100.0	94.0	95.2
R_600_1.3_2	600	95.0	100.0	93.5	93.8
R_800_1.3_1	800	93.6	99.9	92.9	93.3
R_800_1.3_3	800	94.3	95.9	97.1	94.3
R_10.86_2500_1*	2500	89.9	89.4	89.5	89.4
R_10.86_2500_5*	2500	90.0	89.5	89.5	89.9
R_11.72_5000_1*	5000	88.6	88.4	88.5	88.5
R_11.72_5000_5*	5000	88.8	88.5	88.7	88.6
R_12.57_10000_1*	10000	88.2	88.2	88.0	88.2
R_12.57_10000_6*	10000	88.2	88.2	88.2	88.1

* indicates the runs made on HPC Machines

This makes sure that the graph ($G(n,p)$) is Hamiltonian. Having setup the instances for the test, we did run the experiments against all versions of our heuristic.

Table 5.2 shows the performance of our heuristic on random graphs. As you can see, the heuristics were not able to find cycles in many cases. Edge pruning heuristic which performed better for the Cayley graphs again performs better than the other heuristics here in the class of random graphs. The farthest edge heuristic with bi-connectivity restarts wasn't able to find cycles in the cases of smaller instances of the graph though it performed fairly well in the case of Cayley graphs.

Figure 5.3 shows the behavior of edge pruning heuristic when the instances become larger.

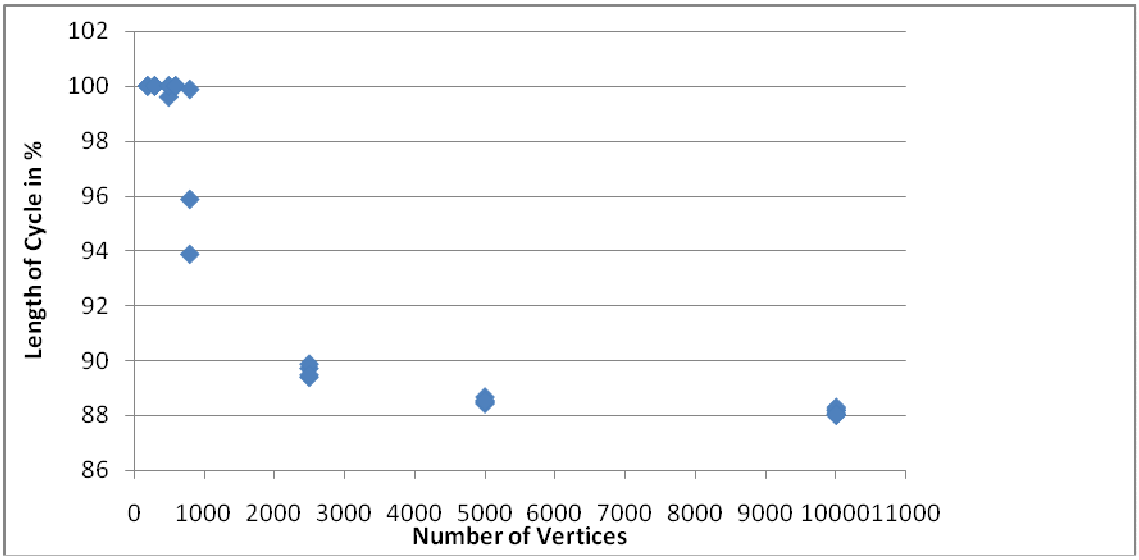


Figure 5.3: Edge pruning heuristic performance

As you see in Figure 5.3, the performance of the heuristic drops drastically above 1000 vertices. Farthest edge removal heuristic's performance is very similar to the performance of the edge pruning heuristic.

5.4.Knight Tour Graphs

We wanted to try running our heuristic against knight-tour graphs as well. The Table 5.3 shows the performance statistics of our heuristic against the knight tour graphs.

Table 5.3: Performance of our heuristic on knight tour graphs

Graph Name	Vertices	Length of the cycle detected in %			
		Basic Heuristic	Edge pruning	Farthest edge removal	Farthest edge with restart
KT_20_20_400_2	400	96.5	99.0	96.5	96.5
KT_20_20_400_3	400	96.5	100.0	96.0	96.5
KT_20_30_600_1	600	96.0	99.7	96.0	95.7
KT_20_30_600_3	600	96.0	95.7	95.0	95.7
KT_20_40_800_1	800	95.8	95.5	95.8	95.3
KT_20_40_800_3	800	95.8	95.8	95.5	95.5
KT_30_30_900_2	900	96.0	95.1	95.6	95.8
KT_30_30_900_3	900	96.0	95.6	95.6	95.8
KT_40_40_1600_2	1600	96.0	95.3	95.4	95.9
KT_40_40_1600_3	1600	95.9	95.5	95.5	95.9
KT_50_100_5000_2*	5000	96.2	95.9	95.9	95.8
KT_50_100_5000_3*	5000	96.0	95.8	95.8	95.9
KT_100_100_10000_1*	10000	96.1	96.0	96.0	96.0
KT_100_100_10000_5*	10000	96.1	96.0	96.0	96.0

* indicates the runs made on HPC Machines

The two conditions that we have to make sure while creating the instances is that the (1) board shouldn't have odd number of squares (2) Assuming the board is represented as *m* by *n*, we made sure that m ≠ 4 and n ≠ 4 (Refer to the theorems in chapter 4 regarding knight tour graphs). The knight tour graph names in the table follow the convention KT_*m*_*n*_#ofVertices_Index.

On examining the results, it is clear that the heuristic has detected 95 % of the cycle before it failed in all the cases in knight tour graphs. Careful analysis (a step to be taken as a part of future work) regarding this behavior of the heuristic against this class of graph would certainly improve the results. The performance of all the versions of the heuristics was almost the same.

5.5.Geometric Graphs

The heuristics were much successful against the geometric graphs. In fact all the versions of the heuristics were able to find cycles in geometric graphs in milliseconds. In most of the cases, it was able to find cycles on the first trial.

The results clearly show that all versions of the heuristic perform superiorly against this class of graphs. As a part of future work geometric graph instances of large sizes can be tested against all the versions of the heuristic.

5.6.Interesting Results

The Tables 5.4-5.7 show the performance of the edge pruning heuristic on various classes of graphs. *"Number of Searches"* in the table indicates the number of breadth first searches made. *"Total SearchLength"* indicates the total length of cycles (BFS cycles) that were made by the heuristic. *"AverageSearchLength"* is *TotalSearchLength/Number of Searches.* Number of trials relies much on the order of edges removed. The average length of the cycle increases as number of vertices increase.

Table 5.4: Experimental Results on Cayley graphs

Graph Name	Vertices	Average Search Length	Number of Trials
3007	336	5.3	1
119	360	30.2	659
636	360	25.2	3512
1073	576	42.7	83
1088	576	20.8	3
155	720	40.3	202
158	720	41.5	1570
225	1152	53.2	1137
2949	1152	35.4	16960
807	1344	55	9648
814	1344	47	18049
217	1440	63.9	410
338	2520	67.1	368
833*	2520	52	340350
2230*	5040	65.5	146540
2269*	5040	109.2	150673
621*	20160	494.2	4310
928*	20160	137.2	2941

* indicates the runs made on HPC Machines

Table 5.5: Experimental Results on Random graphs

Graph Name	Vertices	Average Search Length	Number of Trials
R_200_1.3_1	200	15.2	3928
R_200_1.3_3	200	14.4	447
R_300_1.3_1	300	17	2179
R_300_1.3_3	300	18	7368
R_500_1.3_1	500	23.8	1475
R_500_1.3_3	500	22.2	28064
R_600_1.3_1	600	24.5	23465
R_600_1.3_2	600	26.4	1852
R_800_1.3_1	800	31	25065
R_800_1.3_3	800	28.7	26243
R_10.86_2500_1*	2500	21.7	229343
R_10.86_2500_5*	2500	22.2	383391
R_11.72_5000_1*	5000	28.8	88838
R_11.72_5000_5*	5000	27.6	87582
R_12.57_10000_1*	10000	40.8	8674
R_12.57_10000_3*	10000	40	11408

* indicates the runs made on HPC Machines

Table 5.6: Experimental Results on Knight-tour graphs

Graph Name	Vertices	Average Search Length	Number of Trials
KT_20_20_400_1	400	27.5	117528
KT_20_20_400_3	400	20.4	27800
KT_20_30_600_1	600	25.7	54790
KT_20_30_600_3	600	25.7	55629
KT_25_25_625_1	625	26.4	52622
KT_25_25_625_2	625	26.3	51724
KT_20_40_800_2	800	31.2	31187
KT_20_40_800_3	800	30.9	31533
KT_30_30_900_1	900	33	24253
KT_30_30_900_3	900	47.4	24493
KT_25_40_1000_2*	1000	33.7	1246424
KT_25_40_1000_4*	1000	33.8	1213706
KT_40_40_1600_2	1600	50.3	5558
KT_40_40_1600_3	1600	51	5862
KT_50_50_2500_1*	2500	70.9	181482
KT_50_50_2500_5*	2500	70.6	178650
KT_50_100_5000_2*	5000	115.5	28781
KT_50_100_5000_4*	5000	116.8	28098
KT_100_100_10000_1*	10000	218.8	1761
KT_100_100_10000_5*	10000	223.6	2048

* indicates the runs made on HPC Machines

Table 5.7: Experimental Results on Geometric graphs

Graph Name	Vertices	Average Search Length	Number of Trials
G_200_0.03_1	200	13.5	1
G_200_0.03_2	200	11.2	1
G_300_0.02_1	300	9.9	1
G_300_0.03_2	300	17.2	1
G_400_0.01_2	400	17.8	2
G_400_0.01_3	400	15.1	9
G_500_0.01_2	500	25.3	1
G_500_0.01_3	500	10.3	1
G_800_0.01_2	800	49.3	1
G_800_0.01_3	800	20.4	1
G_0.0188_1000_3*	1000	17.5	1
G_0.0188_1000_4*	1000	35	1
G_0.0072_2520_3*	2500	25.8	1
G_0.0072_2520_4*	2500	75.5	1
G_0.00615_5000_1*	5000	161.4	1
G_0.00615_5000_3*	5000	32.8	1
G_0.00353_10000_2*	10000	160.7	1
G_0.00353_10000_3*	10000	83	1

* indicates the runs made on HPC Machines

The chart in the Figure 5.4 shows the comparison between *average search length* versus *number of vertices* for all the classes of the graphs. There is a linear increase in terms of average search length.

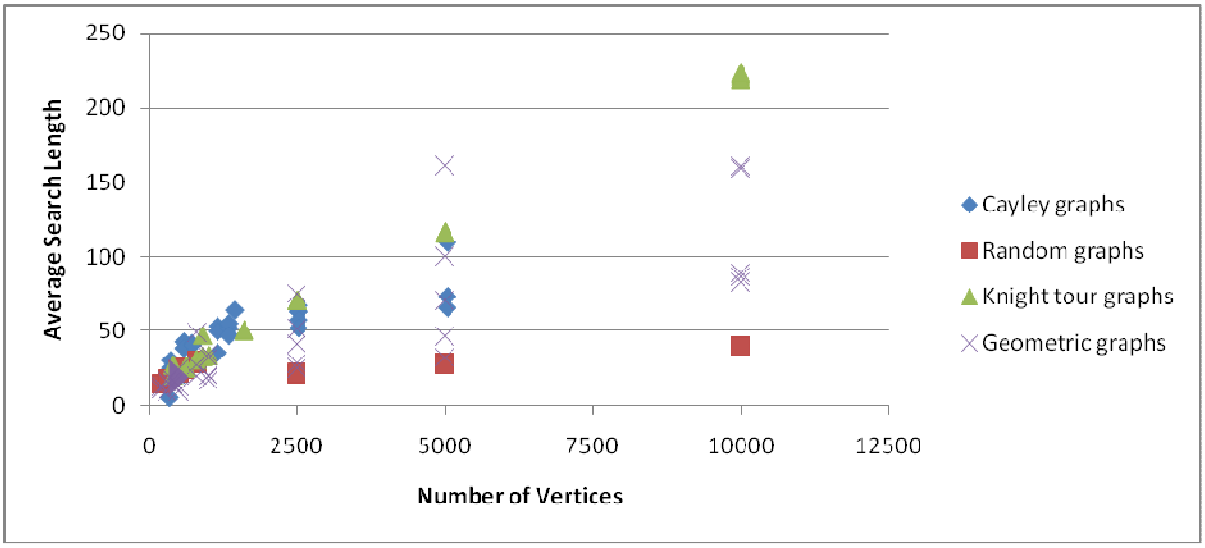


Figure 5.4: Average Search Length vs Number of Vertices

Expected execution time per trial should be proportional to the product of average search length, number of vertices and average degree. As average search length, indicates the number of nodes in the cycle, the total number of node accesses should be product of average search length and average degree of the graph. The Table 5.8 shows the results for Cayley graphs.

Table 5.8: Experimental Results Runtime/Trial

Class	Vertices	Avg Search Length	Avg Degree	Expected Run time/trial	Actual Run time/trial (seconds)
Random	200	15.2	9.1	27512	< 0.05
Random	200	14.5	9.1	26245	< 0.05
Random	200	14.4	9.1	26064	< 0.05
Geometric	200	13.5	16.1	43389	< 0.05
Geometric	200	11.2	15.5	34720	< 0.05
Geometric	200	13.1	15.7	41186.4	< 0.05
Random	300	17	9.7	49317	< 0.05
Random	300	17	9.7	49317	< 0.05
Random	300	18	9.7	52218	< 0.05
Geometric	300	9.9	16.6	49361.4	< 0.05
Geometric	300	16.6	16.4	81871.2	< 0.05
Geometric	300	17.2	16.5	85243.2	0.1
Cayley	336	5.3	3.0	5342.4	< 0.05
Cayley	360	30.2	3.0	32616	< 0.05
Cayley	360	25.2	3.0	27216	< 0.05
Knight tour	400	27.5	6.1	66880	< 0.05
Knight tour	400	27.4	6.1	66636.8	< 0.05
Knight tour	400	20.4	6.1	49612.8	< 0.05
Geometric	400	17.8	11.5	81595.2	< 0.05
Geometric	400	15.1	11.7	70909.6	< 0.05
Random	500	23.8	10.5	124355	< 0.05
Random	500	23.3	10.5	121742.5	< 0.05
Random	500	22.2	10.5	115995	< 0.05
Geometric	500	13.3	14.4	95693.5	< 0.05
Geometric	500	25.3	14.3	181148	0.1
Geometric	500	10.3	14.4	74005.5	< 0.05

Table 5.8: Continued

Class	Vertices	Avg Search Length	Avg Degree	Expected Run time/trial	Actual Run time/trial (seconds)
Cayley	576	42.7	3.0	73785.6	< 0.05
Cayley	576	20.8	3.0	35942.4	< 0.05
Cayley	576	38.4	3.0	66355.2	< 0.05
Random	600	24.5	10.7	157584	< 0.05
Random	600	26.4	10.7	169804.8	< 0.05
Random	600	25.9	10.7	166588.8	< 0.05
Knight tour	600	25.7	6.4	98688	< 0.05
Knight tour	600	25.7	6.4	98688	< 0.05
Knight tour	600	25.7	6.4	98688	< 0.05
Knight tour	625	26.4	6.5	107250	< 0.05
Knight tour	625	26.3	6.5	106843.75	< 0.05
Cayley	720	40.3	3.0	87048	< 0.05
Cayley	720	41.5	3.0	89640	< 0.05
Cayley	720	40.6	3.0	87696	< 0.05
Random	800	31	11.2	276520	< 0.05
Random	800	30.8	11.2	274736	< 0.05
Random	800	28.7	11.2	256004	< 0.05
Knight tour	800	30.9	6.5	161668.8	< 0.05
Knight tour	800	31.2	6.5	163238.4	< 0.05
Knight tour	800	30.9	6.5	161668.8	< 0.05
Geometric	800	27.7	22.7	503032	0.2
Geometric	800	49.3	22.5	888188.8	0.2
Geometric	800	20.4	23.1	376176	0.2
Knight tour	900	33	6.7	198990	< 0.05
Knight tour	900	33.4	6.7	201402	< 0.05
Knight tour	900	47.4	6.7	285822	< 0.05

Table 5.8: Continued

Class	Vertices	Avg Search Length	Avg Degree	Expected Run time/trial	Actual Run time/trial (seconds)
Knight tour	1000	33.7	6.7	225790	< 0.05
Knight tour	1000	33.8	6.7	226460	< 0.05
Geometric	1000	32.7	52.0	1701708	0.3
Geometric	1000	20.2	52.6	1062116	0.3
Geometric	1000	17.5	51.5	900725	0.2
Geometric	1000	35	52.9	1850100	0.2
Geometric	1000	20.5	51.9	1063745	0.3
Cayley	1152	53.2	3.0	183859.2	< 0.05
Cayley	1152	35.4	3.0	122342.4	0.1
Cayley	1152	49.6	3.0	171417.6	< 0.05
Cayley	1344	51.7	3.0	208454.4	< 0.05
Cayley	1344	55	3.0	221760	< 0.05
Cayley	1344	47	3.0	189504	< 0.05
Cayley	1440	63.9	3.0	276048	< 0.05
Knight tour	1600	50.9	7.0	571708.8	0.2
Knight tour	1600	50.3	7.0	564969.6	0.2
Knight tour	1600	51	7.0	572832	0.2
Random	2500	21.7	10.9	589155	0.3
Random	2500	21.2	10.9	575580	0.1
Random	2500	21.5	10.9	583725	0.1
Random	2500	22.2	10.9	602730	0.2
Knight tour	2500	70.9	7.2	1277972.5	0.3
Knight tour	2500	70.8	7.2	1276170	0.3
Knight tour	2500	70.7	7.2	1274367.5	0.3
Knight tour	2500	70.6	7.2	1272565	0.3
Geometric	2500	41.8	53.1	5546860	2.3
Geometric	2500	51.8	52.7	6827240	1.6
Geometric	2500	25.8	53.4	3441720	1.7
Geometric	2500	75.5	53.1	10020737.5	1.7
Geometric	2500	29.1	53.4	3881212.5	1.6

Table 5.8: Continued

Class	Vertices	Avg Search Length	Avg Degree	Expected Run time/trial	Actual Run time/trial (seconds)
Cayley	2520	63.1	3.0	477036	0.1
Cayley	2520	67.1	3.0	507276	0.1
Cayley	2520	63	3.0	476280	0.2
Cayley	2520	57.3	3.0	433188	0.2
Cayley	2520	52	3.0	393120	0.2
Random	5000	28.8	11.7	1687680	0.7
Random	5000	28.6	11.7	1675960	0.7
Random	5000	28.8	11.7	1687680	0.6
Random	5000	28.4	11.7	1664240	1.0
Random	5000	27.6	11.7	1617360	0.7
Knight tour	5000	116.5	7.4	4310500	2.0
Knight tour	5000	115.5	7.4	4273500	2.0
Knight tour	5000	115.7	7.4	4280900	2.0
Knight tour	5000	116.8	7.4	4321600	2.0
Knight tour	5000	116.6	7.4	4314200	2.0
Geometric	5000	161.4	89.8	72468600	23
Geometric	5000	100.3	90.6	45440915	15.7
Geometric	5000	32.8	90.7	14866600	16.3
Geometric	5000	70.8	90.7	32097180	15.1
Geometric	5000	46.9	90.7	21278530	14.9
Cayley	5040	73.1	3.0	1105272	0.4
Cayley	5040	65.5	3.0	990360	0.4
Cayley	5040	109.2	3.0	1651104	0.4
Random	10000	40.6	12.6	5103420	3.3
Random	10000	40.8	12.6	5128560	10.0
Random	10000	40.6	12.6	5103420	5.0
Random	10000	40	12.6	5028000	5.0
Random	10000	40.7	12.6	5115990	5.0
Random	10000	40.3	12.6	5065710	5.0

Table 5.8: Continued

Class	Vertices	Avg Search Length	Avg Degree	Expected Run time/trial	Actual Run time/trial (seconds)
Knight tour	10000	218.8	7.6	16628800	< 0.05
Knight tour	10000	221.2	7.6	16811200	< 0.05
Knight tour	10000	222.6	7.6	16917600	< 0.05
Knight tour	10000	219.2	7.6	16659200	< 0.05
Knight tour	10000	223.6	7.6	16993600	< 0.05
Geometric	10000	86.8	105.8	91834400	68
Geometric	10000	160.7	105.7	169779550	69.7
Geometric	10000	83	105.6	87631400	74.6
Geometric	10000	159.1	105.4	167659580	73.9
Geometric	10000	89.3	105.5	94247220	81.5
Cayley	20160	494.2	3.0	29889216	13.4
Cayley	20160	137.2	3.0	8297856	19.7

5.7. Comparison with other heuristics

The biggest demerit of our heuristic is that it runs indefinitely even in the case of graphs which don't have Hamiltonian cycles. Edge pruning procedure doesn't detect the possible existence of cycles before pruning the edges. Hence our heuristic performs considerably poor when compared to programs of Shields, McKay and Hertel (Hertel, 2004) in the case of Cayley graphs.

The code hasn't been fine tuned to beat the run times of the other popular heuristics. We wanted to check the heuristic's effectiveness rather than run times. We ran experiments on Cayley graphs and compared against the results that were available in the Ruskey's website (Effler & Ruskey, 2000). As our heuristic was not able to find cycles in instances of Cayley

graphs (>2500) we could not compete against the popular heuristics (that proved to solve larger instances of Cayley graphs easily).

As our heuristic performed better in the case of geometric graphs we wanted to compare the performance against a well known heuristic. The Table 5.9 shows the comparison of our heuristic performance against the Hertel's SCHA algorithm (Refer to chapter 2 literature review section for more details regarding this algorithm) in terms of time taken. Version 5 of Hertel's SCHA algorithm was used as a basis for comparison.

The experiments were run with a maximum time limit of 10 minutes on a Intel ® CPU T2250 system with a processor speed for 1.73 GHz and 2 GB RAM. The time taken by our heuristic was less than the SCHA algorithm as the size of the instances increased. In fact for lager instances (graph size=10000) the SCHA algorithm didn't find the cycles in the specified time limit whereas our heuristic found cycles in all the instances.

Table 5.9: Comparison for our heuristic vs SCHA algorithm

Graph Name	Vertices	Time taken by our heuristic (in seconds)	Time taken by SCHA algorithm (in seconds)
G_200_0.03_1	200	0	0.2
G_200_0.03_2	200	0	0.7
G_200_0.03_3	200	0	0.0
G_300_0.02_1	300	0.1	0.5
G_300_0.02_3	300	0.2	1.3
G_300_0.03_2	300	0.1	0.3
G_400_0.01_2	400	0.2	356.9
G_400_0.01_3	400	1.2	35.7
G_500_0.01_1	500	0.2	39.4
G_500_0.01_2	500	0.2	245.4
G_500_0.01_3	500	0.2	126.9
G_800_0.01_1	800	0.5	11.3
G_800_0.01_2	800	0.5	6.8
G_800_0.01_3	800	0.5	83.5
G_0.0188_1000_1	1000	1.2	0.3
G_0.0188_1000_2	1000	1	4.2
G_0.0188_1000_3	1000	1.2	0.5
G_0.0188_1000_4	1000	1.1	1.7
G_0.0188_1000_5	1000	1.1	0.8
G_0.0072_2520_1	2520	5.9	43.3
G_0.0072_2520_2	2520	6.9	74.7
G_0.0072_2520_3	2520	7.8	337.7
G_0.0072_2520_4	2520	7.5	33.0
G_0.0072_2520_5	2520	5.2	570.9
G_0.00615_5000_1	5000	35.5	211.1
G_0.00615_5000_2	5000	45.2	Didn't complete (98.9%)
G_0.00615_5000_3	5000	40.2	Didn't complete (98.7%)
G_0.00615_5000_4	5000	50.1	152.8
G_0.00615_5000_5	5000	44.4	Didn't complete (98.9%)

Table 5.10: Continued

Graph Name	Vertices	Time taken by our heuristic (in seconds)	Time taken by SCHA algorithm (in seconds)
G_0.00353_10000_1	10000	230.2	Didn't complete (96.8%)
G_0.00353_10000_2	10000	249.6	Didn't complete (94.1%)
G_0.00353_10000_3	10000	216.9	Didn't complete (94.5%)
G_0.00353_10000_4	10000	225.2	Didn't complete (94.9%)
G_0.00353_10000_5	10000	275	Didn't complete (92.8%)

The Vandegriend's code (variations of Pósa's algorithm) was much slower and it was not completing for even the small instances of geometric graphs that we had. Hence we didn't compare against those implementations.

5.8.Conclusion

In this chapter, we presented the results of our heuristic (all the versions) against various classes of graphs (Cayley, random, knight tour and geometric). Other than the geometric graph instances, the results against other classes of graphs weren't encouraging. The edge pruning heuristic was slightly better than the other heuristics in case of Cayley and random graphs. In the case of knight tour graphs, in all most all the cases the heuristic (all the versions) found about 95 % of the cycle which is an encouraging factor. Finally we presented the comparison results of our heuristic against the Hertel's SCHA algorithm on geometric graphs. With other classes of graphs, our heuristic performed poorly when compared with the popular heuristics.

6. Conclusions and Future Work

In this thesis, we presented a novel heuristic for the Hamiltonian circuit problem. We analyzed and explored the method in depth. Novel ideas such as forced edge propagation and farthest edge removal were applied to the basic heuristic. As expected, randomization of our heuristic gave better results than a methodical deterministic approach. In chapter 5, we presented experimental results for our heuristic's performance on different classes of graphs (Cayley, random, knight tour and geometric graphs). We now conclude with various suggestions that can be taken up as future research.

First and foremost, as pointed out in chapter 5, our heuristic runs indefinitely on graphs until it finds a Hamiltonian circuit. Analysis can be done here and the heuristic can be modified such a way that it tries to recognize non Hamiltonian graphs rather quickly. The theorems introduced in Chapter 2 and the surveys (Gould, Advances on the Hamiltonian problem-a survey, 2003) (Gould, Updating the Hamiltonian problem—a survey, 1991) give all the necessary and sufficient conditions for a graph to be Hamiltonian. It can be made sure that the heuristic checks for these conditions.

The next improvement can be done in the edge pruning procedure. The heuristic initially selects a random edge to be pruned (if all the vertices have greater than degree 2) and after propagating forced edges the heuristic still makes random choices. Instead of making mere random choices, the graph structures can be analyzed (as in Hertel's algorithm) and appropriate edge pruning procedure changes could be made. Farthest edge removal technique is not applicable to all graphs because most graphs don't have multiple dense components. Analysis can be done to detect dense components on the fly and use farthest edge removal wherever appropriate rather than using it for the entire procedure.

There are many improvements that can be considered to improve runtime. We store the graph as an adjacency list in the memory and process the graph for the Hamiltonian cycle. Rather than following this approach, edges can be generated on the fly and memory could be saved and the processing could be faster. In this way we are not restricting the heuristic to run for a relatively small number of vertices. Most of the graphs that are of theoretical interest, i.e., are conjectured but not known to be Hamiltonian, have tens or even hundreds of thousands of vertices. Code tuning should also be considered in future to achieve much better performance.

In knight tour graphs, from our experimental results, it is clear that the heuristic managed to include 95% of the vertices in the cycle, almost in all the cases. Our heuristic relies much on adjacent open nodes in the cycle to perform cycle expansion. In some cases it is possible that the adjacent nodes may not be open and the vertices (that are not part of the cycle) could be left out for the same reason. We suspect that same problem could exist in larger instances of cubic Cayley graphs as well. Hence a hybrid procedure like path extension can be used along with our heuristic to achieve better results. The main advantage of our heuristic is that it can generate a cycle which includes almost 80% of vertices on the first trial for most of the graphs. A simple path rotation is also an option so that the open nodes come together and the cycle expansion procedure can efficiently make use of this.

Our heuristic is not equipped with backtracking. Once the heuristic ends up with a graph which is no longer bi-connected (after edge pruning) then it restarts with a fresh copy of the graph to start a new attempt. Once the heuristic prunes an edge it never puts it back. In this way, the heuristic loses much of its information as it could have pruned an important edge that is needed for the Hamiltonian cycle. Some sort of backtracking could be done to restore the edges that were determined to be wrong decisions and try with another set of edges. Restoring an edge back to the graph makes the propagation of forced edge component more complex. Hence an in-depth analysis has to be done before taking this approach as it could affect the run time of the heuristic to a large extent.

Bibliography

Angluin, D., & Valiant, L. G. (1977). Fast probabilistic algorithms for hamiltonian circuits and matchings. *Proceedings of the ninth annual ACM symposium on Theory of computing* (pp. 30-41). New York: ACM.

Appel, M., & Russo, R. P. (2002). The connectivity of a graph on uniform points on [0, 1]d. *Statist. Probab. Lett. , 60*, 351-357.

Bollabas, B. (1983). Almost all regular graphs are Hamiltonian. *Europ. J. Combinatorics , 4*, 97-106.

Bollabas, B. (2004). *Random Graphs* (Second ed.). Cambridge University Press.

Bollabas, B., Frieze, A. M., & Fenner, T. I. (1987). An algorithm for finding Hamilton paths and cycles in random graphs. *Combinatorica. 7*, pp. 327-341. New York: Springer-Verlag Inc.

Bondy, J., & Murty, U. (2008). *Graph Theory.* Springer.

Broder, A. Z., Frieze, A. M., & Shamir, E. (1991). Finding hidden Hamiltonian cycles. *Proceedings of the twenty-third annual ACM symposium on Theory of computing* (pp. 182-189). New York: ACM.

Brunacci, F. A. (1988). DB2 AND DB2A - 2 USEFUL TOOLS FOR CONSTRUCTING HAMILTONIAN CIRCUITS. *European Journal of Operational Research , 34* (2), 231-236.

Cooperman, G., & Finkelstein, L. (1992). New Methods for Using Cayley Graphs in Interconnection Networks. *Discrete Applied Mathematics 37/38* , 224-232.

Danielson, G. H. (1968). On finding simple paths and circuits in a graph. *IEEE Trans. Circuit Theory , 15*, 294-295.

DeLavernede. (1839). *Memories de l'iicademie Royale du Gard* , 151-179.

Diaz, J., Mitsche, D., & Perez, X. (2007). Sharp threshold for Hamiltonicity of random geometric graphs. *SIAM J. Discrete Math. , 21* (1), 57-65.

Dirac, G. A. (1952). Some theorems on abstract graphs. *Proc. London Math. Soc. 2*, pp. 69-81. Cambridge: LMS Publications.

Effler, S., & Ruskey, F. (2000). *Enumeration, isomorphism and Hamiltonicity of Cayley graphs: 2-generated and cubic*. Retrieved Aug-Dec 2008, from http://www.theory.cs.uvic.ca/~cos/cayley1/

Erdős, P., & Rényi, A. (1959, Dec 6). On Random Graphs. *Publ. Math.* , 290-297.

Euler, L. (1759). Solution d'une Question Curieuse qui ne Paroit Soumise a Aucune Analyse. *Mémoires de l'Academie Royale des Sciences et Belles Lettres , 15*, 310-337.

Foster, R. M. (1932). Geometrical Circuits of Electrical Networks. *Trans. Amer. Inst. Elec. Engin. , 51*, 309-317.

Frieze, A. (1987). Finding Hamilton cycles in sparse random graphs. *Journal of Combinatorial Theory Series A , 44* (3), 230-250.

Gardner, M. (1957, May). Mathematical Games: About the Remarkable Similarity between the Icosian Game and the Towers of Hanoi. *Sci. Amer.* , 150-156.

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W.H. Freeman.

Gould, R. J. (2003). Advances on the Hamiltonian problem-a survey. *Graphs Combin , 19* (1), 7-52.

Gould, R. J. (1991). Updating the Hamiltonian problem—a survey. *Journal of Graph Theory , 15* (2), 121-157.

Greenlaw, R., & Petreschi, R. (1995). Cubic Graphs. *ACM Computing Surveys , 27* (4), 471-495.

Hakimi, S. (1966). Recent progress and new problems in applied graph theory. *IEEE Region Six Conference Record* , 635-643.

Hertel, A. (2004). Hamiltonian Cycles in Sparse Graphs. University of Toronto, Toronto: Master's Thesis.

J.Komlos, & Szemeredi, E. (1983). Limit distribution for the existence of Hamiltonian cycles in a random graph. *Discrete Math. , 43* (1), 55-63.

Jixiang, M., & Qiongxiang, H. (1996). Almost all Cayley graphs are hamiltonian . *Acta Mathematica Sinica , 12* (2), 151-155.

Kamae, T. (1967). A systematic method of finding all directed circuits and enumerating all directed paths. *IEEE Trans. Circuit Theory , 14*, 166-171.

Karp, R. M. (1972). Reducibility Among Combinatorial Problems. In R. E. Miller, & J. W. Thatcher, *Complexity of Computer Computations* (pp. 85-103). New York: Plenum.

Kawarabayashi, K. (2001). A Survey on Hamiltonian Cycles. *Interdisciplinary Information Science , 7*, 25-39.

Kompel'maher, V. L., & Liskovec, V. A. (1975). Successive generation of permutations by means of a transposition basis. *Kibernetika* (3), 17-21.

Korshunov, A. D. (1985). A new version of the solution of a problem of Erdos and R´enyi on Hamiltonian cycles in undirected graphs. *Random graphs '83* , 171-180.

Korsunov, A. D. (1976). Solution of a problem of P. Erdos and A. R´enyi on Hamiltonian cycles in undirected graphs. *Soviet Math Dokl. , 17* (3), 760-764.

Krivelevich, M., & Sudakov, B. (2003). Sparse pseudo-random graphs are Hamiltonian. *Journal of Graph theory , 42*, 17-33.

Kroft, D. (1967). All paths through a maze. *Proc. IEEE 55* , 88-90.

Legendre. (1830). *Theorie des Nombres* (2 ed., Vol. 2). Paris, France.

Lovász, L. (1970). Problem 11. *Combinatorial structures and their applications* .

Malik, D., Mordeson, J. M., & Sen, M. (1997). *Fundamentals of Abstract Algebra* (McGraw-Hill International Edition ed.). McGraw-Hill.

McKay, B. D., & Praeger, C. E. (1994). Vertex-transitive graphs which are not Cayley graphs. *Journal of the Australian Mathematical Society (Series A) , 56*, 53-63.

Pósa, L. (1976). Hamiltonian circuits in random graphs. *Discrete Math. , 14* (4), 359-364.

Pak, I., & Radoicic, R. (2004, August). Hamiltonian paths in Cayley graphs.

Parthasarathy, K. (1964). Enumeration of all paths in digraphs. *Psychometrika , 29*, 152-165.

Penrose, M., Diaz, J., Petit, J., & Serna, M. (2001). Approximating Layout Problems on Geometric Graphs. *Journal of Algorithms , 39*, 78-116.

Pohl, I. (1967). A method for finding Hamilton paths and Knight's tours. *Communications of the ACM , 10* (7), 446-449.

Pratt. (1825). *Studies ofchess* (6 ed.). London: Samuel Bagster.

Rao, V. V., & Murti, V. G. (1969). Enumeration of all circuits in a graph. *Proc. IEEE , 57*, 700-701.

Raptopoulos, C., & Spirakis, P. (2005). Simple and Efficient Greedy Algorithms for Hamilton Cycles in Random Intersection Graphs. In *Lecture Notes in Computer Science (Algorithms and Computation)* (Vol. 3827/2005, pp. 493-504). Springer Berlin / Heidelberg.

Roberts, S. M., & Flores, B. (1966). Systematic generation of Hamiltonian circuits. *Comm. ACM , 9*, 690-694.

Robinson, R. W., & Wormald, N. C. (1992). Almost all cubic graphs are hamiltonian. *Random Structures and Algorithms , 5* (2), 363-374.

Rubin, F. (1974). A Search Procedure for Hamilton Paths and Circuits. *Journal of the ACM (JACM) , 21* (4), 576-580.

Ruskey, F., & Savage, C. (1993). Hamilton cycles that extend transposition matchings in Cayley graphs of Sn. *SIAM J. Discrete Math. , 61* (1), 152-166.

Schwenk, A. (1991). Which rectangular chessboards have a knight's tour? *Math. Magazine , 64*, 325-332.

Shields, I. (2004). Hamilton Cycle Heuristics in Hard Graphs. Ph.D. Dissertation.

Skiena, S. (1998, June 15). *The Hamiltonian Page*. Retrieved Jan-Dec 2008, from http://www.ing.unlp.edu.ar/cetad/mos/Hamilton.html

Thomason, A. (1989). A simple linear expected time algorithm for finding a Hamilton path. *Discrete Mathematics , 75* (1-3), 373-379.

Vandegriend, B. (1998). Finding Hamiltonian cycles: Algorithms, Graphs and Performance. University of Alberta, Edmonton, Alberta: Master's Thesis.

Vandermonde. (1774). Remarques sur les Problèmes de Situation. *L'Histoire de l'Académie des Sciences avec les Mémoires, Année 1771* , 566-574.

Warnsdorff, H. C. (1823). *Des Rösselsprungs einfachste und allgemeinste Lösung.* Schmalkalden: Varnhagenschen Buchhandlung.

West, D. B. (2001). *Introduction to Graph Theory.* Englewood Cliffs, NJ: Prentice-Hall.

Witte, D., & Keating, K. (1985). On Hamilton cycles in Cayley graphs in groups with cyclic commutator subgroup. *Ann. Discrete Math , 27*, 89-102.

Yau, S. S. (1967). Generation of all Hamiltonian circuits, paths, and centers of a graph, and related problems. *IEEE Trans. Circuit Theory , 14*, 79-81.