A Fast Algorithm For Finding Hamilton Cycles

by

Andrew Chalaturnyk

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Masters of Science
in
Computer Science

Winnipeg, Manitoba, Canada
Summer 2008

Abstract

This thesis is concerned with an algorithmic study of the Hamilton cycle problem. Determining if a graph is Hamiltonian is well known to be an NP-Complete problem, so a single most efficient algorithm is not known. Improvements to the understanding of any single NP-Complete problem may also be of interest to other NP-Complete problems.

However, it is an important problem and creating an algorithm which is efficient for many families of graphs is desirable.

The majority of the work described within starts with an exhaustive backtracking search, called the Multi-Path method and explores two main areas:

- Creating an algorithm based upon the method which is both efficient in time and memory when implemented in code.
- Developing a pruning condition based upon separating sets that may be found during the execution of the method in order to maximize the amount of pruning possible.

Both of these areas represent a significant evolution of previous work done with the method. The resulting algorithm is extremely fast and requires only $O(n + \varepsilon)$ space in memory, where n is the number of vertices and ε is the number of edges.

Additionally a class of graphs based on the Meredith graph is described. These graphs have a property which significantly affects the performance of the multi-path method. The insights that follow from this property lead to a reduction technique that further improves the algorithm in a significant way when determining if graphs are non-Hamiltonian. Further directions for study along the lines pursued by these insights is discussed.

Testing of the algorithm is performed on a family of graphs known as knight's move graphs, which are known to be difficult for algorithms dealing with Hamilton cycles.

Contents

Chapter 1

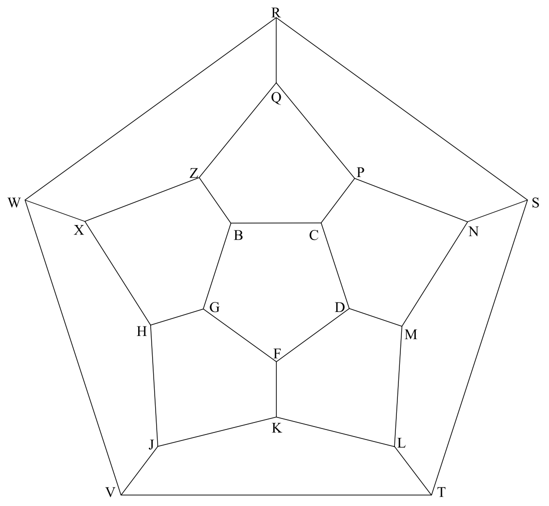
Introduction

This thesis describes work that improves and extends upon an algorithm for finding *Hamilton cycles* in graphs.

Named for *Sir William Rowan Hamilton*, the popularity of the problem of finding Hamilton cycles has one of its roots in the *Icosian Game* invented by Hamilton. The object of the game is to find a continuous route along the edges of a dodecahedron that visit all of its corners exactly once and ends at the starting corner.

A graph can be visualized as a collection of points, or *vertices*, connected by lines, or *edges*. A *cycle* is route along edges visiting vertices until ending at the originating vertex. A Hamilton cycle is a cycle that visits every vertex in a graph. The difficulty of finding Hamilton cycles increases with the number of vertices in a graph. Not all graphs contain a Hamilton cycle, and those that do are referred to as Hamiltonian graphs. Determining if a graph is Hamiltonian can take an extremely long time.

In the broad field of Computer Science the problem of determining if a graph is Hamiltonian is known to be in the class of *NP-Complete* problems.



Simply put, the time it takes to solve NP-Complete problems can rise exponentially with the problem size.

Another NP-Complete problem very closely related is the *Traveling Salesman Problem.*

The nature of the Hamilton Cycle problem is such that no single most efficient algorithm is known [Van98]. In [Van98], *Vandegriend* provides a survey of different Hamiltonian algorithms and the problems encountered that can cause extreme slowdowns during algorithm execution.

Any improvements that can be made to speed up solutions to both determining if a graph is Hamiltonian and finding Hamiltonian cycles are of interest. Also, because of the complexity of the problem, improvements may reveal more insight into the qualities of graphs that make them Hamiltonian. Due to the nature of NP-Complete problems, all problems in the class NP may benefit from improvements or insights found.

The multi-path method introduced by Rubin [Rub74] and *Christofides* [Chr75] is an exhaustive search for all possible Hamilton cycles that can be found within a graph. The work by *Kocay* [Koc92], implements the method with an algorithm that overlays additional pruning capabilities into the search. Under certain conditions the improvements in [Koc92] can allow for significant pruning of the search tree generated by the multi-path algorithm.

This thesis extends and improves on the work in [Koc92] with two major improvements and introduces a method for dealing with certain special graphs that can lead to exponential increases in the performance of the algorithm.

As with the work from [Koc92], the work described here is for finding Hamilton cycles in undirected simple graphs.

The *first* of the improvements are to the design and implementation of the algorithm in [Koc92]. These improvements result in reduced runtime and decreased memory use and are significant code optimizations. The memory reduction is by an order of magnitude in terms of the number of vertices in the graph. The runtime improvement is linear with respect to the complexity and size of the graph.

The *second* of the improvements is to the scope and frequency of the pruning enhancement from [Koc92].

These improvements are described in depth in chapters 6 through 9.

The *new method* focuses on a recursive graph *reduction* technique that can be applied repeatedly to graphs that contain subgraphs with a certain interesting property. Chapter 10 describes this reduction technique and shows how it can be used to sometimes quickly determine if the original graph is not Hamiltonian.

Chapter 2 describes and introduces common Graph theory concepts that are used throughout this work and chapters **??** through **??** introduce and describe the multi-path method in detail.

Lastly in chapter 11, these improvements are analyzed and compared against the original implementation from [Koc92].

Chapter 2

Graph Theory Concepts

2.1 Basic Terminology

A **graph** G is composed of a set $V(G)$ of **vertices** and a set $E(G)$ of **edges**. It is assumed that $n = |V(G)|$ and $\varepsilon = |E(G)|$. Every edge $e \in E(G)$ is composed of a set $e = \{u, v\}$ where $u, v \in V(G)$. In **undirected** graphs, edges are unordered pairs of vertices. In **directed** graphs, edges or **arcs** are ordered pairs of vertices. **Simple** graphs do not allow edges to repeat and the vertices in an edge must be distinct. Only simple graphs are considered here.

Edges from $E(G)$ in which v appears are said to be **incident** to v. The two vertices incident to an edge are said to be **adjacent**.

Given the graph G and two vertices u and v from $V(G)$, $u \rightarrow v$ means that u and v are adjacent. $u \not\rightarrow v$ indicates that u and v are not adjacent.

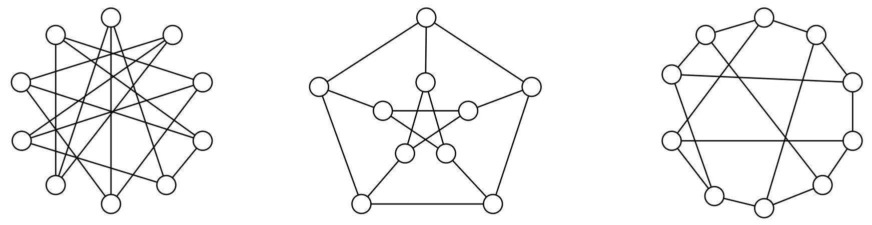


Figure 2.1: Three drawings of the Petersen graph.

2.2 Graphical Representation

As an abstraction, a graph G is represented mathematically as composed of two sets $V(G)$ and $E(G)$. Visually this can be represented as a collection of points, $V(G)$, connected by lines, $E(G)$. The co-ordinates of the points or the shape of the lines do not matter. The same graph G can therefore be represented visually by an infinite number of drawings. Depending on the drawing, different attributes of a graph may become more apparent to the observer. For instance the bottom right drawing in figure 2.1 suggests that there may be a Hamilton cycle in the Petersen graph. While the Petersen graph is not hamiltonian, by using the same graphical orientation of fitting adjacent vertices around a circle, it is a useful representation to help in finding Hamilton cycles in relatively small graphs using simple observation.

2.3 Paths and Cycles

A **path** in G is defined to be an ordered sequence of *distinct* vertices such that every vertex in the sequence is adjacent to the vertex that immediately precedes it and adjacent to the vertex that immediately follows it. This can be written as $u_1 \to u_2 \to ... \to u_k$, where u_1 through u_k are all distinct vertices from $V(G)$. A

cycle in the G is defined as a closed path in G that terminates on the starting vertex. The number of edges in a cycle is equal to the number of vertices visited.

2.4 Connectivity

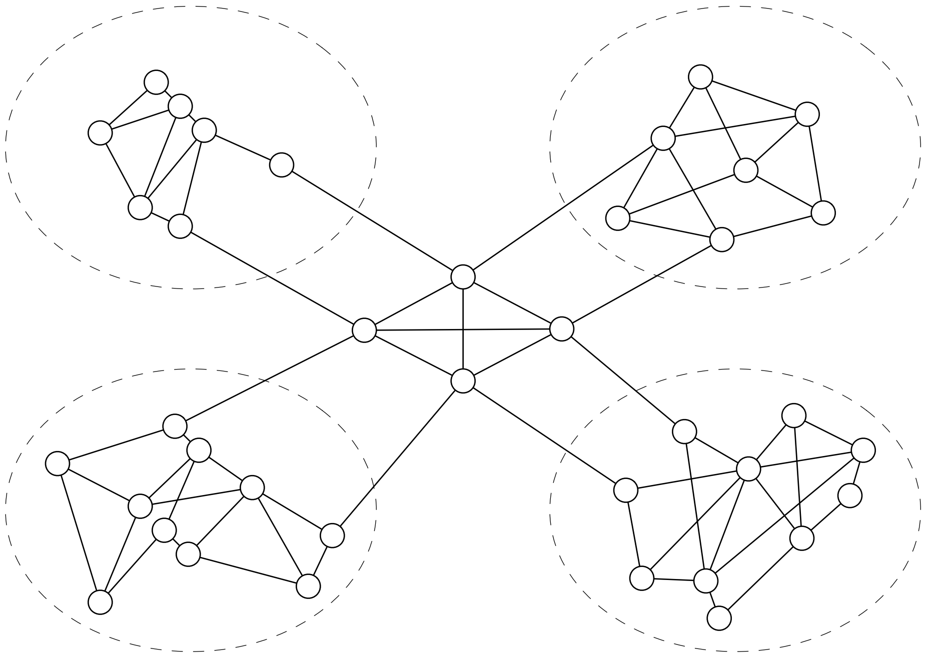


Figure 2.2: Example of a graph with a separating set.

If a path exists between two distinct vertices, $v, u \in V(G)$, they are called **connected**. A graph G is connected if every pair of distinct vertices from $V(G)$ is connected. A **component** in G is a maximal subgraph of G that is connected. A graph with more than one component is called **disconnected**.

In a connected graph G, a **separating set** is a set of vertices from G that when removed, disconnects G into two or more components. A separating set, K, with

$m = |K|$, is called minimal when there is no other separating set in G smaller than m.

A **cutpoint** in G is a single vertex from G that forms a separating set in G.

2.5 Bipartitions

A **bipartition** exists in a graph G, if $V(G)$ can be divided into two distinct sets, A and B, where $A \cap B = \emptyset$ and $A \cup B = V(G)$, and no two vertices from the same set are adjacent. A graph with a bipartition is referred to as **bipartite**.

2.6 Isomorphism

An **isomorphism** is a one-to-one mapping between vertices of two distinct graphs that preserves the connections, or edges, between vertices. If an isomorphism exists, the two graphs are said to be **isomorphic**. The three drawings of the Petersen graph in figure 2.1 are isomorphic to each other.

Chapter 3

The Multi-Path Method

This chapter describes the multi-path method for finding Hamilton cycles. While it was originally proposed by Rubin in [Rub74], the description of the multi-path method within this chapter is a synthesis of the descriptions from [Koc92, Chr75] as well as the work described by this thesis.

To facilitate understanding of both the multi-path method and the enhancements to it some preliminary ground must be covered. Section 3.1 describes a truly exhaustive method of finding Hamilton cycles. Sections 3.2 and 3.3 describe two improvements that can be used to reduce the amount of search required and section 3.4 combines the results of the improvements to describe the multi-path method. Finally in section 3.5, a few refinements that appear in [Koc92] are described and added to the method.

Assume that all graphs G to be considered are *non-trivial* with respect to the Hamilton Cycle problem, implying that all vertices in any graph will be at least of degree three.

The algorithms described in this chapter are partial algorithms used as a tool to convey general information about the multi-path method. Procedures that are not fully defined can be implicitly understood given their names and context of use. Each undefined procedure is assumed to be implementable by some algorithm with a polynomial overhead in time and space.

3.1 An Exhaustive Search

By definition all vertices from $V(G)$ must be part of any Hamilton cycle. The edges that make up any Hamilton cycle are what must be varied.

One method to determine if a graph is Hamiltonian is to generate all combinations of n edges from $E(G)$ until a Hamilton cycle is found or until no combinations are left. In the worst case, the number of edge combinations to try is $\binom{\varepsilon}{n}$.

The order in which edges are chosen with this method is not restricted by the order in which they would be traversed in a Hamilton cycle. When testing if the chosen edges form a Hamilton cycle the correct traversal order will implicitly be found.

Let S be the subgraph of G describing a partial Hamilton cycle formed by choosing m edges, $m < n$, to be on the Hamilton cycle. S forms a subgraph in G, with $E(S)$ and $V(S)$ its edges and vertices. Obviously $m = |E(S)|$.

When $m = n$ we have a candidate subgraph that will form a cycle *iff* G is Hamiltonian.

Algorithm 3.1.1 briefly describes a recursive procedure for generating and testing all possible combinations of edges up to the first Hamilton cycle found.

The algorithm works by recursively fixing an edge within S and attempting all possible edge combinations for the remaining $n - m$ positions to be filled with the current edges available.

IsCycle verifies if $E(S)$ describes a Hamilton cycle and *ChooseEdge* returns some unspecified edge from the remaining edges at the current depth in the search space.

The recursive nature of the procedure allows for the search space of the algorithm to be modeled as a tree. In this case the depth is bounded by n and the number of leaves is bounded by $\binom{\varepsilon}{n}$. Each node in the search tree represents this fixed position in S and the branches represent the current edge in that position.

```
Algorithm 3.1.1: HCEdges( $n$, $\mathcal{E}$, $\mathcal{S}$ )

 comment: { All parameters passed by value.
          { Initially $n = |V(G)|$, $\mathcal{E} = E(G)$ and $\mathcal{S}$ is empty.
          { $result =$ true $\iff$ $G$ is Hamiltonian

 if $|E(\mathcal{S})| = n$ then return (IsCycle($\mathcal{S}$))

 while $\mathcal{E} \neq \emptyset$
   do { $e \leftarrow$ ChooseEdge($\mathcal{E}$)
      { $\mathcal{E} \leftarrow \mathcal{E} - e$
      { if HCEdges( $n$, $\mathcal{E}$, $\mathcal{S} + e$ ) then return ( true )

 return ( false )
```

This type of algorithm is referred to as a *backtracking* algorithm. The ability to return to a specific spot while searching and ensuring that all possible permutations of decisions are attempted from that point characterize these kind of algorithms.

3.2 Incident Edges about v

Algorithm 3.1.1 blindly generates edge combinations. If instead the focus is now placed on vertices, a significant fact emerges. All possible edge combinations about $v \in V(G)$ use one of the $deg(v)$ edges incident to v. There are up to $\binom{deg(v)}{2}$ ways of selecting pairs of incident edges in any possible cycle about v.

Algorithm 3.2.1 is a modification of algorithm 3.1.1 to instead use vertices chosen by *ChooseVertex*, in the selection of edges. These vertices are referred to as *anchor points*. Instead of branching out with all remaining edges at the current spot in the search space, only edges incident to the anchor vertex v are chosen for branches.

Similar to *ChooseEdge* from the previous algorithm, *ChooseVertexEdge* returns some unspecified edge incident to v from the remaining edges at the current depth in the search space .

```
Algorithm 3.2.1: HCVertexEdges( $n$, $\mathcal{V}$, $\mathcal{E}$, $\mathcal{S}$ )

  comment: { All parameters passed by value.
           { Initially $n = |V(G)|$, $\mathcal{V} = V(G)$, $\mathcal{E} = E(G)$ and $\mathcal{S}$ is empty.
           { $result =$ true $\iff$ $G$ is Hamiltonian

  if $|E(\mathcal{S})| = n$ then return ( IsCycle($\mathcal{S}$) )

  $v \leftarrow$ ChooseVertex($\mathcal{V}$)
  if $v \in V(\mathcal{S})$ then $\mathcal{V} \leftarrow \mathcal{V} - v$

  while $deg(\mathcal{E}, v) \neq 0$
     do { $e_b \leftarrow$ ChooseVertexEdge( $v$, $\mathcal{E}$ )
        { $\mathcal{E} \leftarrow \mathcal{E} - e_b$
        { if HCVertexEdges( $n$, $\mathcal{V}$, $\mathcal{E}$, $\mathcal{S} + e_b$ ) then return ( true )

  return ( false )
```

As seen in the algorithm a given vertex may be chosen twice when descending towards a leaf in the search space. This allows for the $\binom{deg(v)}{2}$ possible edge pairs about v to be explored iteratively and dramatically reduces the breadth of the search tree by limiting the total number of leaves while not removing any possible Hamilton cycle that could be found.

The order and manner in which *ChooseVertex* picks vertices in $\mathcal{V}$ will be discussed later in section 6.2.

3.3 Limiting Edges

Edge choices from $E(G)$ can be limited based on the edges already chosen. For any v in a cycle, only two edges incident to v may be used. So $deg(v) = 2$ with respect to the subgraph the cycle defines in G.

Definition (segment)**.** *A segment is a path in G. Segments do not overlap or share endpoints with other segments.*

Let S_i be a subgraph in G defined by some segment and let $S = S_1 \cup \ldots \cup S_k$ be the k segments formed from $E(S)$. $V_{ex}(S)$ denotes the external vertices, or segment endpoints, and $V_{in}(S)$ denotes the internal degree 2 vertices with respect to S.

When adding another edge to S the choice of $e_{m+1} \in E(G)$ is now limited to the edges of $E(G - V_{in}(S))$, as no more edges incident to the internal vertices in S can be chosen and still satisfy the degree 2 restriction.

Inductively it can be seen when m starts at 0 and approaches n, only two edges incident to a given $v \in V(G)$ can ever be in $E(S)$.

The main idea of the multi-path method is to construct a Hamilton cycle by piecing together the disjoint segments formed by the edges in S.

3.4 Multi-Path Method

The multi-path method uses the two ideas from sections 3.2 and 3.3 to dramatically reduce the search space of algorithms 3.1.1 and 3.2.1.

The selection of a new edge will eventually cause a chain reaction as each new $v \in V_{in}(S)$ may induce conditions that require unchosen additions to S.

Let the vertex and edge sets for the derived graph M be:

$$V(M) = V(G) - V_{in}(S) \tag{3.1}$$

$$E(M) = \mathcal{E} \cap E(G - V_{in}(S)) \tag{3.2}$$

Where $\mathcal{E}$ is the set of edges remaining in G to be selected for $E(S)$. More on $\mathcal{E}$ will be described later in this section. For now, assume that $\mathcal{E}$ has been inherited from the parent node in the search space.

M represents the remaining choices available to S at the point in the search space it is defined. Deriving M may result in one or more vertices in $V(M)$ that no longer have any choice in what edges are placed into $E(S)$, leading to the following definitions:

Definition (forced vertex)**.** *A forced vertex is a vertex $v \in V(M)$ where $deg(v) = 2$ when $v \notin V(S)$ or $deg(v) = 1$ when $v \in V_{ex}(S)$.*

Definition (forced edge)**.** *An forced edge is an edge $e \in E(M)$ where $v \in e$ and v is a forced vertex.*

Forced vertices must be added to $V_{in}(S)$. These additions are accomplished by adding forced edges one at a time to $E(S)$.

The chain reaction occurs when the two vertices incident to a forced edge both belong to $V_{ex}(S)$. In this situation either a cycle is found or two segments are forced to merge. The two endpoints representing the junction point of two segments now belong to $V_{in}(S)$. Any edges not in $E(S)$ that are incident to the junction vertices will be removed in M and may result in one or more additional forced vertices.

Definition (consistent)**.** *M is said to be **consistent** if for all $v \in M$, $deg(v) > 1$ when $v \in V_{ex}(S)$ and $deg(v) > 2$ when $v \notin V_{ex}(S)$.*

The chain reaction continues until M stabilizes and is found to be consistent or until a *stop condition* occurs. A consistent M implies a *stable* search space.

A stop condition can occur in three situations:

Stop Condition (1A)**.** $\exists v \in V_{ex}(S)$ *and* $deg(M, v) = 0$

Stop Condition (1B)**.** $\exists v \notin V_{ex}(S)$ *and* $deg(M, v) = 1$

Stop Condition (2)**.** *For some segment $S_i \in S$ there exists a forced edge, $e \in E(M)$, such that the addition of e to S_i would form a cycle. In this case if $m = n$ a Hamilton cycle has been found. If $m < n$ no Hamilton cycle can exist in the current S.*

After each change to S a stop condition may occur in M. Algorithm 3.4.1 describes this process of forcing edges into S and testing for a stop condition.

```
Algorithm 3.4.1: ForceEdges( $G$, $\mathcal{V}$, $\mathcal{E}$, $\mathcal{S}$ )

  comment: All parameters passed by value.
           Returns the set { $\mathcal{V}$, $\mathcal{E}$, $\mathcal{S}$, stop } where stop is a
           boolean flag indicating a stop condition.

  $E_M \leftarrow E(G - V_{in}(\mathcal{S})) \cap \mathcal{E}$
  $V_M \leftarrow V(G) - V_{in}(\mathcal{S})$

  while not IsConsistent($V_{ex}(\mathcal{S}), E_M$)
    do $\mathcal{S} \leftarrow \mathcal{S}$ + GetForcedEdge($V_{ex}(\mathcal{S}), E_M$)
       $E_M \leftarrow E(G - V_{in}(\mathcal{S})) \cap \mathcal{E}$
       $V_M \leftarrow V(G) - V_{in}(\mathcal{S})$
       if IsStopCondition($\mathcal{S}, E_M$) then return ( { $\emptyset$, $\emptyset$, $\emptyset$, true } )

  $\mathcal{V} \leftarrow \mathcal{V} \cap V_M$
  $\mathcal{E} \leftarrow E_M$

  return ( { $\mathcal{V}$, $\mathcal{E}$, $\mathcal{S}$, false } )
```

Now that the process of forcing edges into S has been described, the rest of the method follows the same design as algorithm 3.2.1.

Using the first improvement from section 3.2, an *anchor point* v is chosen from $V(M)$ and a single edge, e_b, incident to v is selected from $E(M)$ and added to $E(S)$. This edge is referred to as a *branching edge*, and is a reflection of one of the two locations in the search space where choice is allowed in the algorithm. The other being the selection of anchor points. The overall shape and size of the search space is controlled by these choices.

After removing e_b, M may no longer be consistent and have edges to be forced into S, ultimately reducing the number of anchor points and branching edges to choose from. Algorithm 3.4.2 describes the multi-path method using the ideas given so far.

As previously mentioned and used in the previous algorithms, $\mathcal{E}$ represents the edges remaining in G that can be chosen for S. Edges are removed from $\mathcal{E}$ in two ways. The first is by the $G - V_{in}(S)$ reduction. The second is by removal of branching edges, e_b, that have had all possible combinations of $E(S)$ attempted for the current location in the search space. Once a branching edge has been used, no more cycles can be found that contain that edge when continuing to search at the current depth in the search tree.

```
Algorithm 3.4.2: HCMultiPath( $G$, $\mathcal{V}$, $\mathcal{E}$, $\mathcal{S}$ )

comment: { All parameters passed by value.
         { Initially $\mathcal{V} = V(G)$, $\mathcal{E} = E(G)$ and $\mathcal{S}$ is empty.
         { $result =$ true $\iff$ $G$ is Hamiltonian

{ $\mathcal{V}$, $\mathcal{E}$, $\mathcal{S}$, $stop$ } ← ForceEdges( $G$, $\mathcal{V}$, $\mathcal{E}$, $\mathcal{S}$ )
if $stop$ then return ($|V(G)| = |E(\mathcal{S})|$)

$v$ ← ChooseVertex($\mathcal{V}$)

while $deg(\mathcal{E}, v) \neq 0$
  do { $e_b$ ← ChooseVertexEdge( $v$, $\mathcal{E}$ )
     { $\mathcal{E} \leftarrow \mathcal{E} - e_b$
     { if HCMultiPath( $G$, $\mathcal{V}$, $\mathcal{E}$, $\mathcal{S} + e_b$ ) then return ( true )

return ( false )
```

As can be seen from the use of the stop conditions, no *IsCycle* test is required with the multi-path method. If the algorithm ever allows $E(S)$ to contain n edges, a Hamilton cycle will be found. Taken together, both of the improvements reduce the total breadth and average depth of the search space. *This is one of the major benefits* *of the multi-math method.*

3.5 Simplifying the Search

A couple of minor improvements to the multi-path method are now described that focus on simplifying movement through the search space. Both of these improvements are used by [Koc92]. Section 3.5.1 introduces a reduction that can be applied to G in order to direct the path through the search space more efficiently. Section 3.5.2 removes redundant recursive calls to the multi-path method at a given node in the search space.

Algorithms 3.5.1 and 3.5.2 reflect these improvements, and are the basis for the multi-path algorithm developed and presented later on in this work.

3.5.1 Reducing G

At any given node in the search space there is the original graph G, the set S of segments derived from the m edges selected so far and the derived graph M as described by equations 3.1 and 3.2.

A reduced graph $G_{S\mathcal{E}}$ defined by M and S may be defined for any point in the search space.

$$G_{S\mathcal{E}} = M + E_v(S) \tag{3.3}$$

$E_v(S)$ is a set of *virtual* edges created to connect the segment endpoints in M. Each virtual edge represents a single segment from S at the current point in the search space. Edges from M may be referred to as *real* edges.

Due to equation 3.3 some of the conditions that define forced vertices, consistency and stop conditions can be simplified:

Definition (forced vertex)**.** *A vertex $v \in V(G_{S\mathcal{E}})$ where $deg(v) = 2$.*

Definition (consistent)**.** *$G_{S\mathcal{E}}$ is said to be consistent if for all $v \in V(G_{S\mathcal{E}})$, $deg(v) > 2$.*

Stop Condition (1)**.** *$\exists v \in V(G_{S\mathcal{E}})$ such that $deg(v) = 1$.*

In algorithms 3.5.1 and 3.5.2, the variable $\mathcal{G}$ represents the reduced graph $G_{S\mathcal{E}}$.

Ensuring a Simple Graph

A consequence of introducing the virtual edges from $E_v(S)$ in the forming of $G_{S\mathcal{E}}$ is the potential for a virtual and a real edge to share the same endpoints. In this situation $G_{S\mathcal{E}}$ would not represent a simple graph. With respect to G, the segment that the virtual edge represents and the real edge would form a cycle. As long as this cycle does not contain all vertices, it is safe to remove the real edge.

In fact its removal would reduce the size of the tree formed below the current point in the search space.

In algorithm 3.5.1 it is to be assumed that *ReduceGraph* would ensure that a simple graph is returned, and any virtual edge takes precedence over a real edge.

3.5.2 Removal of Branching Edges

As seen in algorithm 3.4.2, once a branching edge, e_b, has been attempted, it is removed for the remaining branches at the current level in the search space. Since all edge combinations that may exist at that point in the search space which contain that branching edge will have been attempted, it must be removed. This removal may cause an inconsistent $G_{\mathcal{SE}}$.

In this case the multi-path method as currently defined, will fail at the first call to ForceEdges for each recursive call to the multi-path method on the remaining branches. It would instead be better to detect if $G_{\mathcal{SE}}$ is inconsistent once the recursive call to the multi-path method returns for that branching edge. In algorithm 3.5.2, the addition of a call to ForceEdges from within the while loop reflects this improvement.

```
Algorithm 3.5.1: FORCEEDGES( $\mathcal{G}$, $\mathcal{S}$ )

  comment: { All parameters passed by value.
           { Returns the set { $\mathcal{G}$, $\mathcal{S}$, stop } where stop is a
           { boolean flag indicating a stop condition.

  $\mathcal{G} \leftarrow$ REDUCEGRAPH( $\mathcal{G}$, $\mathcal{S}$ )
  if ISSTOPCONDITION( $\mathcal{G}$, $\mathcal{S}$ ) then return ( { $\emptyset$, $\emptyset$,  true } )

  while not ISCONSISTENT($\mathcal{G}$)
    do { $\mathcal{S} \leftarrow \mathcal{S}$ + GETFORCEDEDGE($\mathcal{G}$)
       { $\mathcal{G} \leftarrow$ REDUCEGRAPH( $\mathcal{G}$, $\mathcal{S}$ )
       { if ISSTOPCONDITION( $\mathcal{G}$, $\mathcal{S}$ ) then return ( { $\emptyset$, $\emptyset$,  true } )

  return ( { $\mathcal{G}$, $\mathcal{S}$,  false } )
```

```
Algorithm 3.5.2: MultiPath( $n$, $\mathcal{G}$, $\mathcal{S}$ )

 comment: { All parameters passed by value.
          { Initially $\mathcal{G} = G$, and $\mathcal{S}$ is empty.
          { $\mathit{result} =$ true $\iff G$ is Hamiltonian

 $\{\ \mathcal{G},\ \mathcal{S},\ \mathit{stop}\ \} \leftarrow$ ForceEdges( $\mathcal{G}$, $\mathcal{S}$ )
 if $\mathit{stop}$ then return ( $n = |E(\mathcal{S})|$ )

 $v \leftarrow$ ChooseVertex($V(\mathcal{G})$)

 while $\mathit{deg}(v) \neq 0$
        { $e_b \leftarrow$ ChooseVertexEdge( $v$, $E(\mathcal{G})$ )
        { $\mathcal{G} \leftarrow \mathcal{G} - e_b$
   do   { if MultiPath( $n$, $\mathcal{G}$, $\mathcal{S} + e_b$ ) then return ( true )
        { $\{\ \mathcal{G},\ \mathcal{S},\ \mathit{stop}\ \} \leftarrow$ ForceEdges( $\mathcal{G}$, $\mathcal{S}$ )
        { if $\mathit{stop}$ then return ( $n = |E(\mathcal{S})|$ )

 return ( false )
```

Chapter 4

The Search Space

To provide context on where the improvements described in this thesis are focused, this chapter illustrates the search space of the multi-path method.

The search trees used to describe the search space are assumed to be traversed in a depth first, left to right manner and are a reflection of the route generated by algorithm 3.5.2.

A hypothetical example of a search tree is presented in figure 4.1. The larger circular nodes represent the anchor points chosen and the smaller solid points represent vertices removed from $\mathcal{G}$ and placed into $V_{in}(\mathcal{S})$. The small diamond shaped leaves of the search tree represent the occurrence of a stopping condition.

Anchor points that exist close to the extremities of the search tree are referred to as *leaf nodes*. A leaf node is an anchor point in the tree that only contains leaves as descendants.

The edges joining points in the search tree do *not* correspond to edges added to $E(\mathcal{S})$. The dashed edges between points in the tree do however indicate that a branching edge was chosen at that point in the search space.

The rightmost solid edge descending out of an anchor point in the search tree represents an inconsistent state occurring due to the removal of a branching edge from $\mathcal{G}$.

4.1 Example: Petersen Graph

The search tree for the non-hamiltonian Petersen graph is shown in this example. This graph is used as an example because it terminates after only finding six stopping conditions.

Figure 4.2 represents the search tree for the Petersen graph. The labeled vertices on the drawing of the Petersen graph correspond to the labels on the search tree. In the case of the Petersen graph, each of the six leaves terminate with stop condition 1 being encountered.

Table 4.1 describes the state of $E(\mathcal{S})$ at each of the six leaves in the search tree moving from left to right. Note how the branching edges correspond to the anchor points from the tree, and how $E(\mathcal{S})$ behaves like a stack when moving between the branching edges.

In figure 4.3 the state of $\mathcal{G}$ is represented for each stable state after the branching edge has been added to $E(\mathcal{S})$ for each branching edge down to the leftmost leaf. Note stop condition 1 is detected for vertex 4 after branching edge **{2,3}** has been added. Also note that virtual edge {8,9}, requires the removal of the corresponding real edge, leading to the addition of vertex 7 to $V_{in}(\mathcal{S})$.

As a comparison to the first method presented for finding Hamilton cycles, this

Table 4.1: $E(\mathcal{S})$ at the leaves of search tree for the Petersen graph. Branching edges are bold.

Edge #	1	2	3	4	5	6
1	**{1,3}**	**{1,3}**	**{1,3}**	**{1,3}**	{1,7}	{1,7}
2	**{1,7}**	**{1,7}**	{1,10}	{1,10}	{1,10}	{1,10}
3	{9,10}	{9,10}	{7,8}	{7,8}	{2,3}	{2,3}
4	{6,10}	{6,10}	{4,7}	{4,7}	{3,5}	{3,5}
5	**{2,3}**	{3,5}	**{2,3}**	{3,5}	**{2,4}**	{2,9}
6	{5,6}	{2,4}	{5,6}	{2,4}	{8,9}	{4,7}
7	{5,8}	{2,9}	{5,8}	{2,9}	{9,10}	{4,6}
8	{7,8}	{5,6}	{2,4}	{9,10}	{4,7}	{5,6}

example of the multi-path method only generates 6 leaves, using only 5 points where a branching is directly controlled by the algorithm. The first method would have generated $\binom{15}{10}$, or 3003 leaves, and many more branchings would have to be controlled directly by the algorithm.

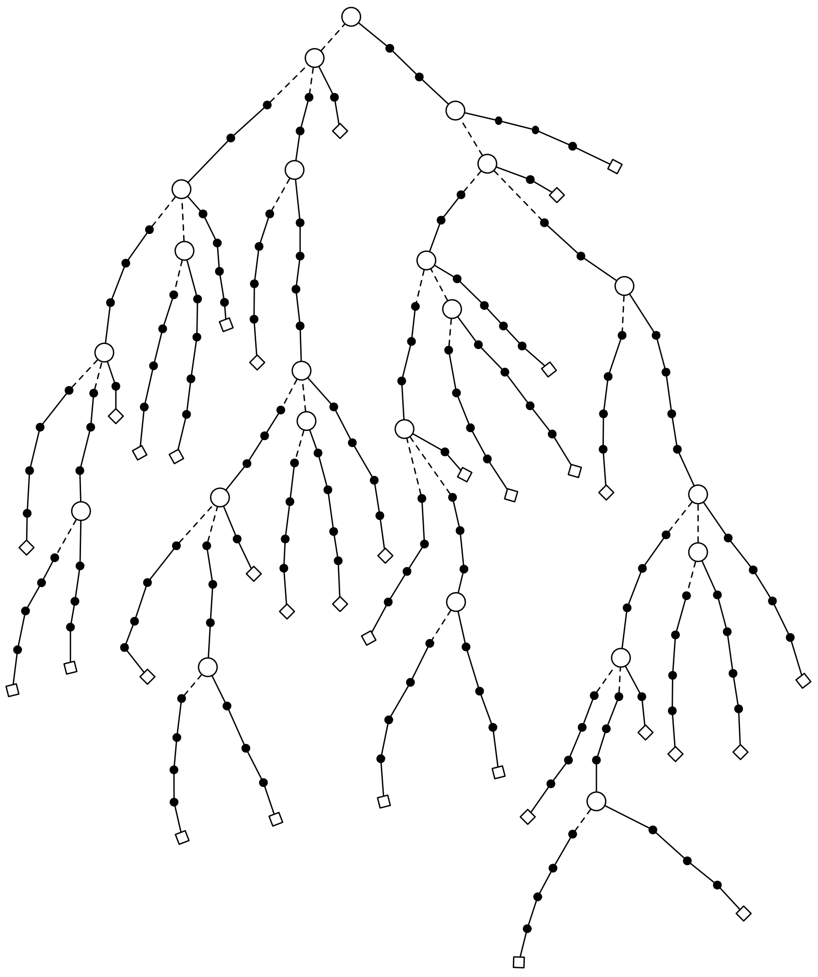


Figure 4.1: Hypothetical search tree generated during the search for Hamilton cycles for some unknown graph using the Multi-Path method. Typically the tree generated would be extremely large and would not be practical to display.

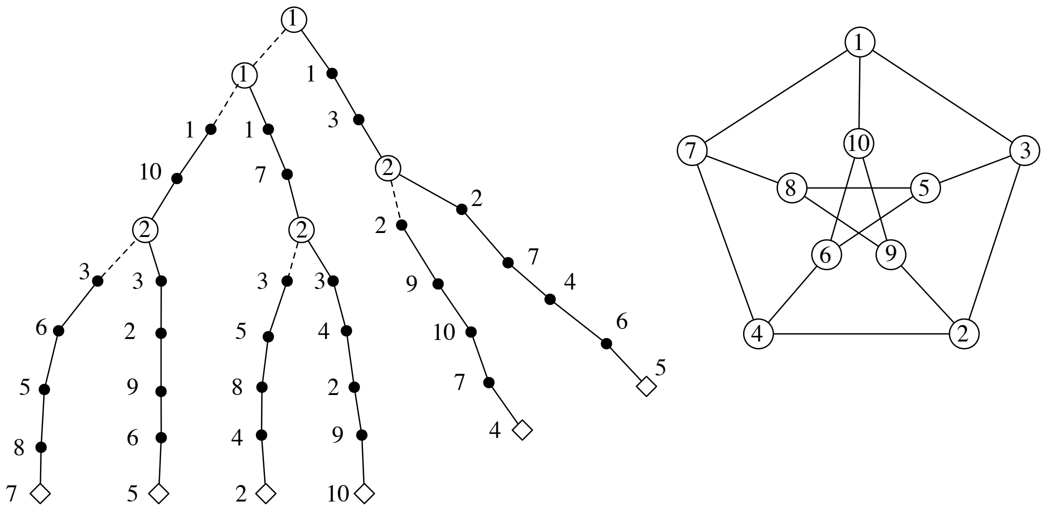


Figure 4.2: Full search tree of the multi-path method for the Petersen graph.

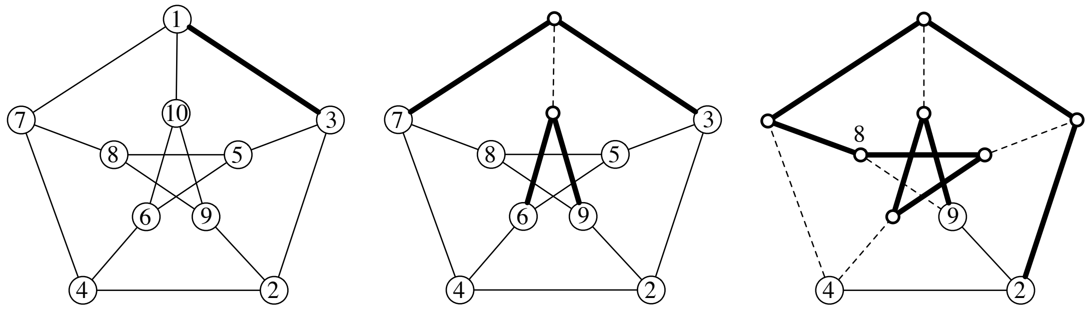


Figure 4.3: State of $\mathcal{G}$ after each branching edge down to the leftmost leaf.

Chapter 5

Considerations for Algorithm Design

Before progressing further with the design of the multi-path algorithm, some considerations with respect to data structures and implementation details for [Koc92] must first be mentioned.

The implementation for [Koc92] is similar to what has already been introduced for the multi-path method in algorithms 3.5.2 and 3.5.1. It is a recursive procedure that can be configured to either stop at the first Hamilton cycle found, or to visit each possible Hamilton cycle by examining the entire search space.

If a Hamilton cycle is treated as a stopping condition for a branch instead of a stopping condition for the algorithm, this visiting operation is possible in the multi-path method.

5.1 Data Structures

If each vertex is labeled with a positive integer value, it can be used both as an identifier and as an index in lookup tables, arrays and other data structures. The special identifier of '0' is reserved, and can have specific meaning depending on use.

Typically this use will result in the 0^{th} entry from any table/array to be used either as a temporary store or go unused. Entries in a table/array structure that use the 0 value in an entry where a vertex identifier is expected typically mean the indexed value does not belong to a set tracked by the structure.

5.1.1 Adjacency Matrix and Lists

Two ways to store the edges of a graph are by using either a matrix, refered to as an *adjacency matrix*, or a linked list construct, called an *adjacency list*.

An adjacency matrix is an n x n data structure that uses the index of two vertices to determine the existence of an edge. An adjacency matrix however has two main drawbacks: it is slow to determine all of the edges incident to a given vertex and it takes $O(n^2)$ of memory to store.

A linked list has the drawback of not being able to quickly test if two vertices are adjacent. The adjacency list for a given vertex must be scanned until the other vertex is encountered.

The benefit of an adjacency list is when the traversal of all edges of a given vertex is all that is required. Additionally with adjacency lists, the entire graph is stored in $O(\varepsilon)$ space.

In [Koc92] both of these structures are used, however as will be shown in later chapters, there is need only for the adjacency lists.

5.1.2 Degree Values and Virtual Edges

Storing the current degree value in a lookup table/array is extremely useful, as it avoids the overhead of recalculating them. Depending on the data structure used to represent edges, the time overhead of recalculating is at least $O(deg(v))$. A lookup table reduces this to $O(1)$. The tradeoff here is that more memory is required to store the degree value of each vertex.

One very useful aspect of maintaining the degree array, is that it becomes trivial to test if a vertex is still within the current $\mathcal{G}$, as it must have a non-zero value in the array.

With respect to the multi-path method, the extra virtual edge that may exist for any vertex can be stored using an array of vertices. At any point in the search space, the state of $E_v(S)$ can be reflected by an array with either a '0' value for vertices not in S or a vertex value for the other end point of a virtual edge.

As will be seen later, this way of representing the virtual edge is preferable to inserting a new edge into an adjacency list.

5.1.3 Segment and Cycle Storage

One result of the work done in this thesis, is that it is unnecessary to store the current segments and partial cycles during the running of the algorithm. Instead only directly maintaining the stack of edges in $E(S)$ is necessary, as both segment and cycles can be directly derived from the contents of $E(S)$.

However, in [Koc92] maintaining them is an integral part of the algorithm. To this effect a single array of vertices is used for segments and two arrays of vertices are used to track the current partial and potential Hamilton cycle.

For the segments, a merge-find like structure is maintained in the array. More on the merge-find structure can be found in [Wei95].

For the partial Hamilton cycle, the two arrays use the entries to point to the vertices to the left and to the right of a given vertex.

5.2 Reducing Time and Space Overhead

As noted in the header comments for each of the previous algorithms described, procedure arguments are passed by value.

The overhead required to pass variables by value increases the memory requirements by an order of magnitude. Additionally, passing by value implies that time must be spent copying data. The expected overhead of copying would be in the range of $O(n + \varepsilon)$ per branching edge in the search space.

To limit the time and space penalties of copying by value, [Koc92] uses an in-place reduction on G. This means that a reference to the attributes in G is available at all points during the running of the algorithm and the original state of G is not known until the algorithm finishes. Only the current reduction of G is known and is equivalent to that given by equation 3.3. This in-place reduction is referred to as G_r.

To accomplish this in-place reduction, there must be a way to reverse changes to G_r at each point in the search space, so as to ensure that the state of the data for each function call is equivalent to that of the pass by value model. The method used to restore G_r for the work described in this thesis can be found in the next chapter and differs from the way it is done in [Koc92].

One of these differences is in the extent of removing the pass by value data. While limiting the expense of retaining a complete copy of the graph state per node, in [Koc92] there is still a memory overhead of $O(n^2)$ consisting of degree information and segment information. This is the remaining data that is still passed by value during branching operations, with $O(n)$ cost in time and space for each node. Without the in-place reduction, memory overhead would be $O(n^3)$ in space.

This remaining overhead is eliminated by the work described in this thesis and the entire state of the algorithm only requires $O(n + \varepsilon)$ in space to function.

5.3 Detecting Stopping Conditions

The detection of a stopping condition has very little overhead. The algorithm forces edges into $E(S)$ one at a time, and a stopping condition can be detected while dealing with each edge. Assume the current forced edge is e.

The second stopping condition is easy to detect by simply checking segment membership of each of the two vertices incident to e. If both vertices belong to the same segment, a cycle is forced and a stop flag is raised. This can be done with an time overhead of just $O(1)$.

The first stopping condition has slightly more overhead than the second. A forced vertex never has any more edges to remove from G_r, but the other vertex v incident to e may already be an endpoint for another segment. In this case, v must be added to $V_{in}(S)$ and all remaining edges incident to v must be removed from G_r. The other vertices incident to each of these removed edges must have their degree values reduced by one. If the degree value for any of these vertices is reduced to one, a stopping condition has occurred. The overhead here is just $O(deg(v))$, per non forced vertex pushed into $V_{in}(S)$.

5.4 Tracking Forced Vertices

Instead of scanning each vertex in G_r for forced vertices each time the *ForceEdges* method is called, it is possible to record the current forced edges in a list for quick referencing later.

As mentioned with detecting stopping conditions, changes to S all occur one edge at a time. When removing the edges incident to each new vertex in $V_{in}(S)$, the changes to the degree values for the other vertex can also be used to detect when a vertex has become forced. Each of these vertices can be added to a list for later processing. The list construct used in [Koc92] is a queue. In the implementation for the multi-path algorithm for this thesis a stack is used instead. A stack was chosen because by marking the beginning of the stack with a 0 value and not using it, no length variable is needed. A pointer to the current position is all that is required when a stack is used in this way.

As the order in which forced edges is unimportant with respect to the final consistent G_r, any abstract data type for managing list addition and removal can be used here.

The order in which forced vertices are removed from the list becomes important when trying to detect stopping conditions sooner and more sophisticated structures could be used to enable this, however this is beyond the scope of the work done here and further work could yield some interesting results.

Chapter 6

Designing the Multi-Path Algorithm

Algorithms 3.5.1 and 3.5.2 used to describe the multi-path method are enough to provide a broad recipe on how to go about creating a complete multi-path algorithm. This chapter is focused on fully developing the key aspects of the algorithm with respect to efficiently managing G_r and S, as well as the selection of anchor points.

The implementation of the multi-path algorithm resulting from the work described in this thesis was developed incrementally, as a series of refinements and data structure changes, each of which brought an increase in efficiency and speed. By studying the effects of each of the changes on various graphs, the techniques were continually improved, and new approaches suggested themselves. This chapter and the next represents the finalization of the work in this thesis on the stand alone portion of the multi-path algorithm.

Recall that G_r was introduced in the previous chapter as a representation of $\mathcal{G}$

from algorithm 3.5.1, with the difference being that it is the continuously changing instance of the original graph G that is *not* passed by value during the execution of the multi-path algorithm.

As mentioned in the last chapter, passing by value increases both time and space overhead to the algorithm that would be best left avoided if possible. In order to design an algorithm that completely removes the pass by value requirements of movement between consistent states of G_r, the first section of this chapter is concerned with the mechanics of changes to S and G_r during movement up and down the search space of the multi-path algorithm. This also includes an improved algorithm for creating and extending the segments in S over the one used in [Koc92].

The final section of this chapter deals with the order in which anchor points are chosen. This can have a profound effect on the shape and size of the search space to be navigated by the algorithm.

6.1 Efficient Reduction and Recovery

The reduction and recovery of G_r between branching edges in the search space consumes the majority of the time spent running the algorithm. The improvements described in this section attempt to reduce the internal changes that are required when transforming a graph from one state to another, while additionally removing the remaining pass by value requirements from [Koc92].

By changing how edges are added to S it is possible to reverse the changes to G_r as well as restore the data structures storing segment and degree information, subsequently reducing the memory footprint required by this portion of the algorithm by a factor of n. This results in a memory footprint of $O(n+\varepsilon)$ for the entire algorithm.

Additionally the total time overhead of managing G_r is reduced to $O(\Delta S)$, where ΔS is the number of edges added to $E(S)$ between branching edges. This is instead of the $O(n + \Delta S)$ from the previous version.

Section 6.1.1 describes a different way of creating and extending segments in S that improves on what has been used previously. The improvement here over [Koc92] is difficult to quantify as it is heavily dependent on the structure of the graphs searched, however at worst it will be equivalent in steps required.

Section **??** describe the way removed edges from G_r are managed, and section **??** describes how the state of G_r is restored. The process of restoring G_r used here is the reason for the $O(\Delta S)$ claim.

6.1.1 Extending Segments

Up to this point changes to segments in S occur one edge at a time. In this way there are three possible results when adding a forced or branching edge to $E(S)$; either a segment is created or extended, or two segments will be merged into a single segment. Each time $E(S)$, $E_v(S)$ and G_r must be updated to reflect the new reduction. The process is repeated until no more forced edges remain in G_r.

Extending Segments from Forced Vertices via Paths

The new approach changes focus from single edges to paths that can be extended from forced vertices in G_r. *Extending segments using a branching edge and its corresponding anchor point is a special case of this approach and is described at the end of*

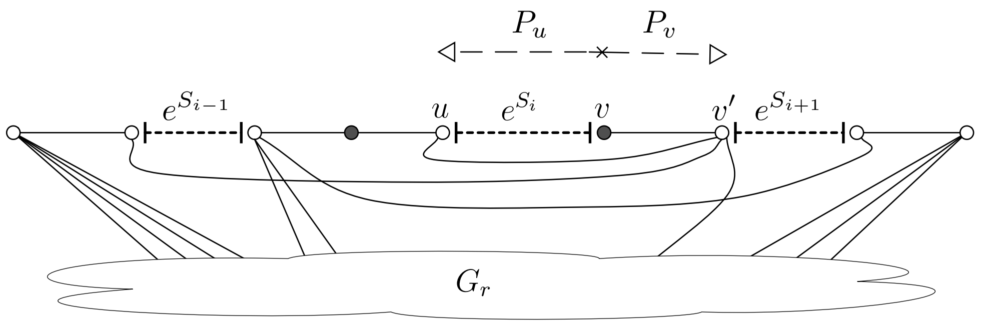


Figure 6.1: Segments S_{i-1}, S_i, and S_{i+1} before extension of S_i^* via forced vertex v. Forced vertices are represented by the solid symbols. The virtual edge $\{u,v\}$ representing S_i is labeled with a e^{S_i} and the other virtual edges are labeled likewise.

this section.

Let v be the next forced vertex to be processed. If v is the endpoint of some segment, $S_i \in S$, virtual edge $\{u,v\}$ represents this segment.

Let P_v be the first path to be extended into G_r that originates at v. The first edge in P_v is the real edge, $\{v,v'\}$. This edge is added to $E(S)$.

Let P_u be the second path to be extended into G_r that originates at v, but proceeds in the opposite direction. If v is the endpoint of a segment, let the first edge in P_u be the virtual edge $\{u,v\}$. If v is not the endpoint of a segment, let the first edge in P_u be the real edge $\{u,v\}$, and *ensure it is added to* $E(S)$. Initially let $u' = u$.

During the growth of P_v and P_u, the shared endpoint v remains fixed, while v' and u' are updated to indicate the new endpoints. Figure 6.1 illustrates this initial configuration on a cross section of an example G_r.

The goal is to form a new segment, S_i^*, represented by the following:

$$S_i^* = P_u + P_v \tag{6.1}$$

Path Growth

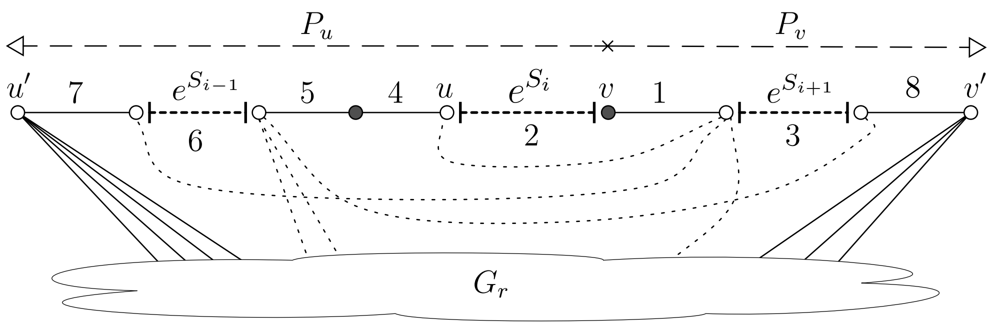


Figure 6.2: Example of extension of paths P_u and P_v from forced vertex v. The numbers represent the order in which the associated edges have been added to the paths during the extension. For real edges, the relative ordering represents the order in which they were added to $E(S)$.

Definition (locally consistent)**.** *A segment $S_k \in S$ is locally consistent if for all $w \in V_{ex}(S_k)$, $deg(G_r, w) > 2$.*

Initially focus is centered on the growth of P_v. The value of v' and the edges that make up P_v change under the following two conditions:

1. v' collides with another segment. The corresponding segment is absorbed by the growing path and v' changes to the other endpoint of that segment.

2. v' is a forced vertex. Let e be the other edge incident to v'. e must be added to the path *as well as* $E(S)$. v' changes to the other endpoint of e.

Once P_v can longer be extended, focus switches to the growth of P_u with the same two conditions for growth applied to endpoint u'.

If an edge from $E_v(S)$ is to be added to a path due to condition 1, the segment it represents is now absorbed by the new segment, S_i^*. This is instead of the merging metaphor. At the endpoint where a collision has occurred, all non-path real edges in $E(G_r)$ which are incident to that endpoint must be removed. This is illustrated in figure 6.2, just after edges 1 and 5 are processed. *Section 6.1.2 will describe how these edges are to be removed from G_r.*

The removal of non-segment edges may cause the first stopping condition to occur or induce new forced vertices.

If one of the current u' or v' become a forced vertex due to a segment collision, the growth of the other path must resume after growth stops for the current path. This can repeatedly occur at multiple points during the growth of both paths causing the focus of growth to alternate between P_u and P_v repeatedly.

Figure 6.2 illustrates this when edges 1 and 5 have been added and the subsequent non-segment edges have been removed. After edge 1 is added, u' becomes forced, allowing P_u to grow. During the growth of P_u, once edge 7 has been added, focus must switch back to P_v in order to add edge 8.

Path growth stops once G_r becomes locally consistent with respect to u' and v'. *At this point it is important to ensure that the new segment represented by a virtual edge $\{u',v'\}$ would not turn G_r into a multi-graph.*

In order to ensure that no real edge $\{u',v'\}$ exists in G_r, the adjacency list of one of u' or v' must be scanned for the other potential endpoint. The adjacency list for the vertex with the lower degree value is chosen. If an edge is found, it must be completely removed as per section 6.1.2. The removal of that edge may cause a locally inconsistent G_r at one or both u' and v' and the paths must continue to grow as before.

Once a locally consistent G_r is reached and is ensured to be a simple graph, the new segment S_i^* can now be defined and a virtual edge can be created in $E_v(S)$. Figure 6.3 illustrates the new virtual edge that represents S_i^*.

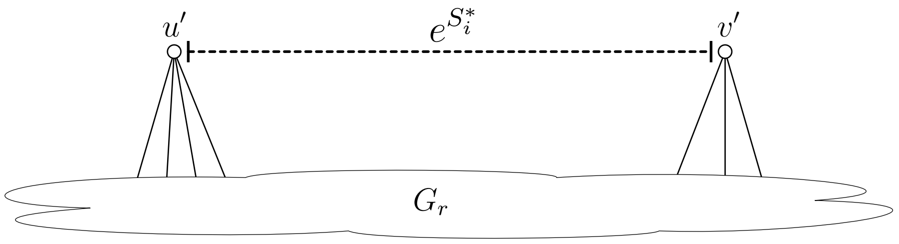


Figure 6.3: S_i^* after extension of S_i. Segments S_{i-1} and S_{i+1} have been absorbed during the extension of segment S_i from the previous figure 6.2.

The process of extending segments is repeated for the next forced vertex remaining in G_r until no more forced vertices remain and a consistent G_r is reached, or until a stopping condition is encountered.

Stopping Conditions

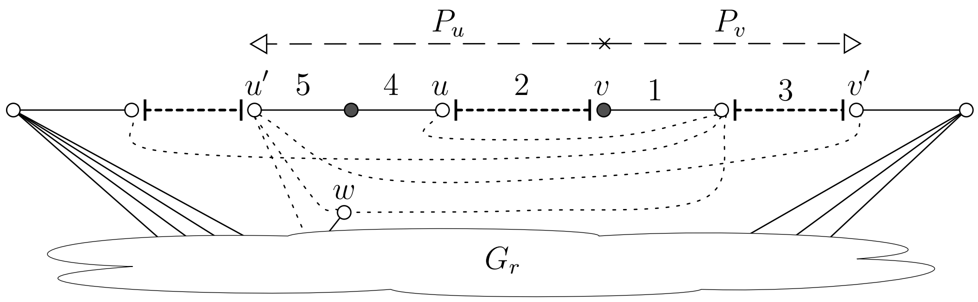


Figure 6.4: Example of stopping condition 1. The removal of non-path edges when extending a segment from v eventually causes vertex w to be reduced to a degree of 1.

As briefly mentioned earlier and as illustrated in figure 6.4, the first stopping condition is encountered while removing edges.

The detection of the second and most important stopping condition is simplified quite nicely here. If during a path extension the two paths are forced to collide, or $u' = v'$, a cycle will have been found. Figure 6.5 illustrates this condition when edge 9 is traversed.

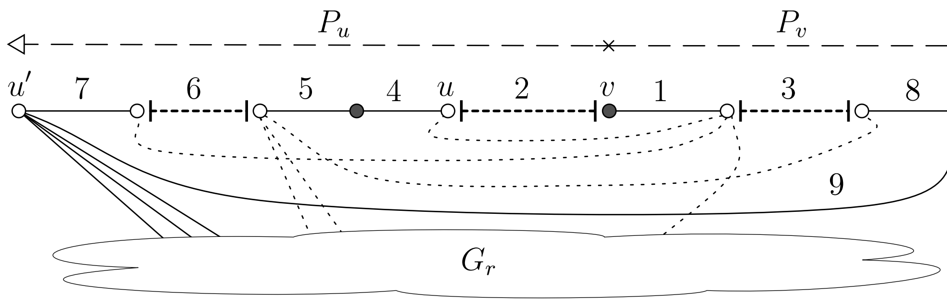


Figure 6.5: Example of stopping condition 2. Attempting to extend P_v with edge 9 leads to $u' = v'$.

Extending Segments from Anchor Points

When an anchor point, v, is selected during the descent down the search space, a branching edge is chosen incident to v. In this and the previous work, the branching edge is the first one encountered in the adjacency list for v. Let the branching edge, e_b, be the edge $\{v,v'\}$.

In order to use the method of extending segments, there are two cases in regards to v that must be considered.

The first case is that no segment is currently incident to v. In this case P_u will initially be empty and $u' = v$. P_v and v' are initialized using e_b and e_b is then added to $E(S)$. The growth of P_v, if required, can now start as described for forced vertices. If during the growth of P_v, v becomes forced, then as previously described, the focus of growth will alternate to P_u once P_v stops growing.

The second case is when v is already a segment endpoint. In this case, all of the real edges but e_b that are incident to v are removed from G_r. This removal converts v into a vertex of degree 2 and it can be treated the same as a forced vertex. P_v and P_u can now be initialized as such, with v being the first vertex to be considered so as to ensure that e_b is in the correct position within $E(S)$.

Improvements over Previous Work

Three key aspects are improved on by this approach to creating and extending segments over what was done previously.

The first is that forced vertices can be processed when they are encountered by the path endpoints rather then through the previous approach of dealing with a single forced vertex at a time. This requires a constant change in focus which creates many redundant segments, that otherwise would be avoided using the new method. The redundant segment creation is the second aspect that is improved upon, as less redundant segments will be created.

The redundant segments cannot be completely avoided with this method because the extension of one segment may still cause a chain reaction that impacts another segment created during the same inconsistent state of G_r.

The third aspect is the avoidance of using an adjacency matrix for detecting if the new virtual edge $\{u',v'\}$ would create a multi-graph. This is what allows for the

$O(n + \varepsilon)$ total space overhead for the algorithm.

The time saving of not managing an adjacency matrix to reflect the current state of G_r though-out the entire search space seems to offset the expense of scanning the adjacency list. More analysis could be done to prove this point.

6.1.2 Managing Removed Edges

There are two ways edges can be removed from G_r during the execution of the algorithm.

The first is a partial removal that takes advantage of the fact that one endpoint of the edge will have been removed from G_r and placed in $V_{in}(S)$. This is what occurs when non-segment edges are removed due to a segment collision.

The second is a complete removal of an edge which occurs when both endpoints of the edge remain in G_r. This is what occurs with the endpoints of branching edges and with the endpoints of real edges that have been superseded by virtual edges.

Partially Removing Edges from G_r

Let w be the endpoint that is the target of a segment collision. The vertex w must be added to $V_{in}(S)$ and each of the non-segment real edges incident to w must be removed. Since G_r is where the active searching occurs during the multi-path method, the removed edges only have to appear missing from G_r. From the perspective of $V_{in}(S)$, each of the real edges incident to w can remain attached.

To accomplish this partial removal of an edge, G is treated as if it were a directed graph from the perspective of the vertices in $V_{in}(S)$ and as an undirected graph from the perspective of G_r. When using adjacency lists to represent the edges in a graph, this ability to switch perspectives between directed and undirected graphs comes naturally, as each vertex in the graph must have its own adjacency list.

Recall that G_r is the continuously modified instance of G for the current location in the search space. When the undirected G is initially defined in memory by the data structures representing it, each edge is actually represented by two arcs. This is one arc pointing out of the adjacency list for each of the two vertices incident to it.

For each non-segment real edge, e_w, incident to w, only the other endpoint of e_w needs to have its corresponding arc removed from its adjacency list. The only time a segment edge must be partially removed is when the other vertex which is incident to that the edge is to be the new endpoint of the fully extended segment. This is because the other vertex incident to the segment edge must remain in G_r.

The elegance of this approach of removing only one arc is that half of the work of removing, and subsequently restoring, an edge is required and *the adjacency list of w contains a complete record of what edges were incident to it at the time it was removed from G_r*. This is extremely useful for the recovery of G_r during ascent up the search space. More on this is discussed in section 6.1.3.

Figure 6.6 is an illustration of the results of using this method of partially removing edges from G_r, for some segment, $S_i \in S$. Note how the edges in $E(S)$ that are not incident to a segment endpoint do not even have to be removed from G_r using this approach.

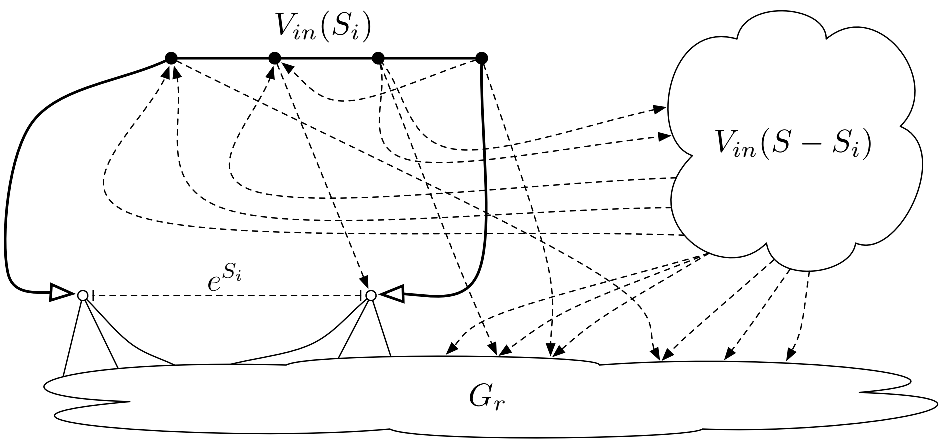


Figure 6.6: Example of the partial removal of edges from G_r. Focus is provided for some segment, $S_i \in S$. The thicker lines represent edges in $E(S_i)$. Arcs shown in the diagram represent the remaining arcs for the detached vertices of G contained by $V_{in}(S)$.

Completely Removing Edges from G_r

Unfortunately only detaching the inactive portion of the graph does not work with branching edges and real edges that have been superseded by virtual edges. A way to manage completely removed edges must be determined.

In [Koc92] all edges to be removed are completely removed from G and stored in a list of removed edges. There is a list of removed edges associated with each consistent state of G_r up the search tree.

During ascent up the search space, once the state returns to the branching edge of an anchor point, the removed edges contained by the list for that consistent state are restored and the branching edge is removed and stored in the list of edges previous to the current one. The latest branching edge will be restored once the current anchor point has been exhausted and the state returns to the consistent state one level above the current one.

In essence this is a stack of lists, that are removed and added in the same manner that branching edges in $E(S)$ are. In [Koc92] this stack of lists is implicitly maintained by the recursive nature of the algorithm.

This same method of managing removed edges is used in the same manner here for the fully removed edges. However the stack of lists must be directly managed by the algorithm, as will be made more apparent by the next chapter.

6.1.3 Unwinding Segments and Restoring State

Now that extending segments and managing the removed edges from G_r have been discussed, a way to restore G_r to a previous state must be introduced. In order to restore G_r without relying on any pass by value data, the adjacency lists, virtual edge array and degree array must have some way to be systematically restored.

Restoring the Adjacency Lists and the Degree Array for G_r

Let G_r^P represent the consistent state of G_r that is to be restored and is some point above the current one in the search space. This corresponds to the closest branching edge when treating $E(S)$ like a stack.

By restoring the edges to G_r in the reverse order in which they were removed from G_r, and by restoring any partially removed edges associated with the incident vertices of each edge, G_r can be incrementally restored towards G_r^P. More on how the edges can be processed in this reversed order will be provided in the next chapter.

Let $e = \{u,v\}$ be the latest segment edge added to $E(S)$. Without loss of generality, assume that u was removed from G_r and added to $V_{in}(S)$. All of the arcs but the arc with target v, listed in the adjacency list for u point to vertices still in G_r that have had the corresponding edge for that arc partially removed.

Generally at the time of restoring e, only one of u or v will have partially removed edges to be restored. The exception to this is for branching edges that join two segments together.

Restoring the degree array to the correct state is trivial. The initial degree value of any removed vertex must be 2. This is due to the vertex being part of the potential Hamilton cycle in $V_{in}(S)$.

For each arc a from the adjacency lists of u or v that represents a partially removed edge, the corresponding opposite pointing arc is a'. The target vertex of a must have a' restored to its adjacency list. During this restoration the target vertices of both a and a' must have their degree entries incremented by one each.

Additionally if a vertex from e represents a segment endpoint, the endpoint still in G_r must have an arc restored to its adjacency list with no corresponding change to the degree entries. Knowledge about e with respect whether or not it is contains a segment endpoint is explained in the next chapter.

Each e from S is processed in this way up to the first branching edge encountered in $E(S)$.

Once this branching edge is reached, the list of completely removed edges associated with that level in the search space can be restored to the adjacency lists along with the degree values of $V(G_r)$ for the endpoints of those edges. Since the edges here are represented by two arcs that are to be restored, the target vertex of each arc is used to increment the corresponding entry in the degree array by 1 each.

Restoring the Virtual Edges Array

The virtual edge array is slightly trickier. Let EV be the virtual edges array for G_r, where the entries in the array correctly represent $E_v(S)$ for all vertices *not* in $V_{in}(S)$ for the current location in the search space. The remaining entries in EV reflect the prior states of $E_v(S)$ in the levels above the current one in the search space.

By taking advantage of the fact that segments are now ‘unwound’ in the reverse order in which they are created, the older entries in EV that represent vertices in $V_{in}(S)$ can be used to restore EV to the correct previous state of $E_v(S)$ before the segments were created or extended.

Let w', z' be the endpoints of a new virtual edge representing a segment S_i^* that is about to created by the path extension algorithm from section 6.1.1. Let w be the vertex that is the origin of the the two paths that were grown to find the segment.

If w was not originally a segment endpoint then $EV[w']$ and $EV[z']$ prior to the new segments creation, were equal to zero. After the segment is finalized, $EV[w'] = z'$ and $EV[z'] = w'$. When restoring $E_v(S)$ to its previous state all that needs to be done is setting both $EV[w']$ and $EV[z']$ back to zero.

If w was a segment endpoint. Let this segment, S_i, be represented by virtual edge $\{w,z\}$.

If $w' \neq w$ and $z' \neq z$, both w and z will both have been added to $V_{in}(S)$. Additionally the entries for w and z in EV will not be changed, as two new entries are to be made for w' and z'. In this case when reversing changes to G_r, all that needs to be done is setting the entries $EV[w']$ and $EV[z']$ to zero. Endpoints w and z will already reflect the correct values in EV for segment S_i.

Without loss of generality, let $w' \neq w$ and $z' = z$. In this case only w will be added to $V_{in}(S)$ and the entry $EV[z]$ will be updated to point to w', leaving the entry $EV[w]$ its previous value of z. When reversing changes to G_r for segment S_i, this unchanged entry $EV[w]$ is used to restore the value of $EV[z]$ and $EV[w']$ will be set to zero. More explicitly $EV[EV[w]] = w$ and $EV[z'] = 0$ when restoring the old virtual edge.

Now that EV is explained in terms of restoring previous states of $E_v(S)$ a way to know when to use which form of restoring EV needs to be found.

Let $e \in E(S)$ be the segment edge about to be restored as per the order explained in section 7.2. If information about how that edge was removed from G_r is maintained, a decision can be made on what endpoint of e needs to have its value in EV restored and in what way. Section 7.3 will provide details on how and what information is stored for each $e \in E(S)$.

If it is found that a vertex $x \in e$ is a segment endpoint when $e \in E(S)$, the restoration of e to G_r will remove that status and x will no longer be on a segment. In this case $EV[x]$ is set to zero. Otherwise $EV[EV[x]] = x$ is used for all other cases, even when $EV[x]$ is zero, as the 0^{th} entry goes unused by G_r.

6.2 Vertex Selection

Selecting the next anchor point at each point in the search space where G_r is consistent has been modified from that used in [Koc92]. Previously after each consistent state resulting from the selection of a branching edge, a vertex with the largest degree in G_r was chosen as the next anchor point. This can be done with an $O(n)$ time scan of the vertices in G_r.

The new algorithm has been modified to only select new anchor points once the last anchor point has been added to $V_{in}(S)$. This is accomplished by branching edges off of the last anchor point until it is removed from G_r. With respect to the search tree this appears as the same anchor point appearing up to two times in a row during the descent towards a leaf. *In this way a static list of vertices can be provided to control the order of anchor point selection.*

As will be seen in the analysis done in chapter **??**, by instead providing a presorted list of vertices in descending order with respect to degree, smaller search trees are generated, subsequently reducing the time to exhaustively search the entire space. *By default the implementation uses this ordering.*

Other presorted lists are possible as well and can sometimes yield better results than what the default provides. For example in chapter **??**, analysis of the results for the knight's graph '$k7x4$' show a remarkable speedup when using a presorted list based on an *ascending* degree sequence.

Finding new heuristics for vertex selection based on this static list has a lot of room for new discoveries and suggestions for further study is mentioned in chapter **??**.

Chapter 7

Navigating the Search Space using a Turing Machine

As briefly mentioned in section 4.1 and emphasized in figure 7.1, the behavior of changes to $E(S)$ between branching edges is that of a stack. This is to be expected due to the recursive nature of the algorithm. A drawback of using recursion is that making the algorithm re-enterable is difficult.

It would be nice to remove the direct recursion and be able to leave and enter the algorithm at any point in the search space. This has applications from easier check-pointing of code that can run for long periods of time, with the full exploration of the search space one Hamilton cycle at a time, and with performance gains due to having no recursive calls of the algorithm.

This chapter describes a way of modeling the navigation of the search space using a Turing Machine to describe the movement up and down the search tree.

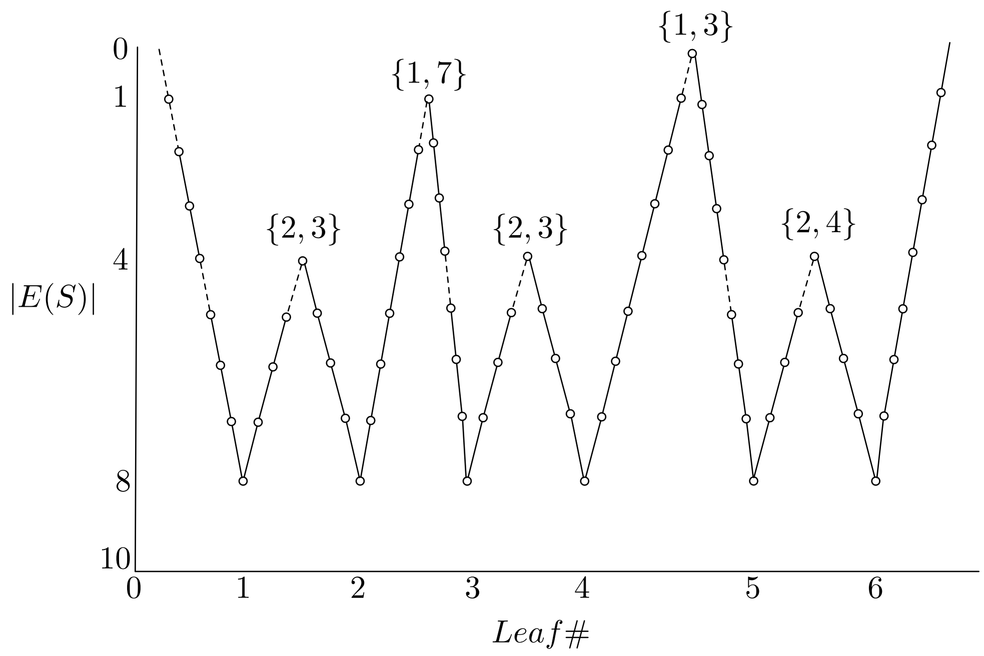


Figure 7.1: Changes to $E(S)$ from leaf to leaf based on the search tree for the Petersen graph. The state of G_r always returns to that of the latest branching edge added to $E(S)$.

7.1 State of the Tape

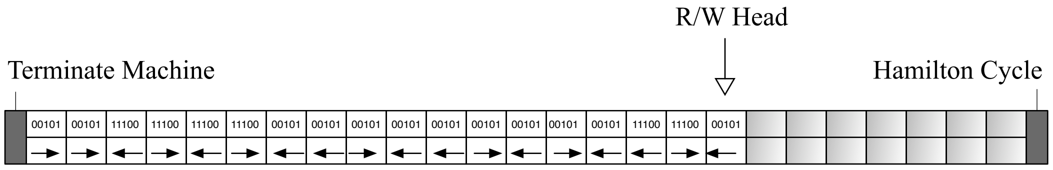


Figure 7.2: Tape for the Multi-Path Turing Machine.

A proper Turing Machine consists of a set of tapes and a read/write head for each tape. Each tape can have an infinite number of positions, called *cells*, to the left and right of each read/write head. The Turing machine described here uses one tape and one read/write head. The tape is only $n + 2$ cells in length.

There are two entries per cell on the tape, one of which is an arc representing an edge from $E(S)$ and the other a bit-field representing a finite number of states.

The Turing Machine used here, models the recursive nature of the algorithm by tracking the the current state of $E(S)$ within its cells and the current position in the search space by the location of the read/write head. The movement of the read/write head to the right tracks the descent into the search space while movement to the left tracks ascent. The bit-fields stored in the cells contain information on how to restore each cell's edge to the graph during ascent.

The two cells on the endpoints of the tape do not have edges set and only have a single bit permanently set in their bit-fields. A illustration of the tape for this Turing Machine can be found in figure 7.2.

On the leftmost cell of the tape, a *termination* bit is set. If the read/write head ever makes it to this cell and reads this termination bit, the entire search space will have been exhausted.

On the rightmost cell of the tape, a Hamilton cycle bit is set. If the read/write head reads this bit, a Hamilton Cycle will have been found.

The edges found in the cells between the endpoints are all of the edges in $E(S)$ that make up the Hamilton cycle.

If the read/write head is not at either endpoint, the current state of $E(S)$ is reflected by the edges found in the cells from the left most endpoint up to and including the current cell pointed to by the read/write head.

The starting point of the Turing machine is the left most cell on the tape and is the only time the machine can be in that position without terminating.

7.2 Edges from $E(S)$

The arcs stored in each cell that represent the edges from $E(S)$ come from the adjacency list of the path endpoint u' or v' of the growing segment in S at the time the edge was added to S. With the exception of the case when a branching edge is added to S as a stand alone segment, each of these arcs always point to a forced vertex added to $V_{in}(S)$ at the time the edge was added to $E(S)$. Additionally if a segment collision occurred when adding an edge the source vertex of the arc stored will be the point of collision. In this way when moving to the left, each vertex and any partially removed edges can be restored in the reverse order in which they were removed from G_r.

7.3 Status Flags represented by the Bit-Field

The following meanings are represented by the flags stored by the bit-field entries:

- Terminate Machine
- Hamilton Cycle
- Target of arc is a segment endpoint.
- Target of arc has edges to be restored (non-segment edges have been removed).
- Arc represents a branching edge, and the target of the arc is an anchor point.
- Arc represents a branching edge, and the source of the arc is an anchor point.
- The anchor point is on a segment.

More flags exist for the pruning operations that will be introduced and discussed in later chapters and are not yet required.

7.4 Moving Read/Write Head

7.4.1 Moving Right

Moving the read/write head to the right implies descent down the search space . The status and edge information is never read while moving to the right, only written. Control of the movement of the head to the right is controlled by the selection of branching edges off of anchor points, and the forced edges due to an inconsistent G_r.

Movement to the right stops when one of the two stopping conditions is encountered.

Typically movement to the right involves selecting an anchor point, choosing a branching edge off that point, forcing any edges from an inconsistent G_r into $E(S)$ and then if necessary choosing another branching edge off the same anchor point. This is repeated until a stopping condition occurs.

7.4.2 Moving Left

Moving the read/write head to the left implies ascent up the search tree and recovery of G_r to a previous state. In this case the read/write head only reads the tape one cell at a time, restoring the state of G_r based on the instructions in the status flags set in the bit field and the edge provided. Movement to the left stops if the Termination flag is encountered or the closest branching edge is encountered.

7.4.3 Switching Directions

If a branching edge is encountered during movement to the left, all remaining edges that have been removed and associated with that branching edge are restored and the branching edge is then removed. Movement can now begin to the right. At this point if G_r is no longer consistent after the removal of the branching edge, forced edges are placed onto the tape until the graph is consistent. *This process of changing directions from left to right is referred to as rotating the anchor point.*

Chapter 8

Implementation of the Multi-Path Algorithm

The ideas from the previous chapters can now be put together and expressed as a complete algorithm. To accomplish this, source code from the implementation of the algorithm is provided within this chapter and in the later appendices. The language used is the ‘c’ programming language.

The only code missing are the routines for loading graphs, and those for allocating, initializing and freeing data structures.

8.1 Data

All of the data types used are found in Appendix A.1. The HCStateRef type references a state structure storing the current state of the search space. This includes the attributes of G_r, the removed edges and the tape of the Turing Machine. Let s

be the current instance for some graph G.

8.1.1 Graph State

The current graph state is maintained via three variables in s. These are an array of adjacency lists $s \rightarrow adjList$, a degree array $s \rightarrow degree$ and a virtual edges array $s \rightarrow virtualEdge$. Each of these arrays are of length $n+1$, so that they can be indexed by the n vertices from $V(G)$.

An adjacency list is defined by type *VArc*. It is a doubly linked list of structures representing arcs in G. Each list, a, is null terminated in the forward direction *(via $a \rightarrow next$)* and circularly linked in the backwards direction *(via $a \rightarrow prev$)*. There is also a target vertex *(via $a \rightarrow target$)* and a cross reference to the opposite pointing arc *(via $a \rightarrow cross$)*.

When adding arcs to an adjacency list, the arc being added is always inserted to the top of the list.

8.1.2 Removed Edges

The removed edges from G_r are maintained by a stack of adjacency lists. Each null terminated list contains arcs from *different vertices*. Since removal of arcs from each list occurs all at once, there is no need to maintain the circular back reference, *'prev'*. When restoring each arc to its corresponding adjacency list, the cross reference, *'cross'*, is used to determine the correct source vertex.

A pointer to the correct adjacency list in the stack is maintained by the algorithm in the following way:

1. The pointer is moved upwards after restoring the completely removed edges it points to. The branching edge that was just exhausted for that point in the search space is then fully removed from G_r and placed into the list the pointer now references.

2. The pointer is moved downwards just before branching starts for some edge. This list the pointer now references will contain all arcs from any fully removed real edges from G_r up to the next anchor point in the search space.

8.1.3 Tape State

The structure storing the tape for the Turing machine is an array of type *HCTape*. The origin of the tape is stored in $s \rightarrow origin$ and the current position of the read/write head is stored in $s \rightarrow pos$. Each entry of the tape contains an arc of type *VArc* representing an edge in $E(S)$ and a bit-field containing the status information.

8.2 Code

The majority of the code for the implementation of the multi-path algorithm can be found in Appendix A. The source code that is listed here is sufficient to describe the overall use and structure of the program.

8.2.1 Running the Turing Machine

The source code in listing 8.1 contains the implementation of *runTuringMachine*. This top level procedure controls the overall movement of the read/write head while navigating the search space.

Before first starting the Turing Machine, the tape must be initialized or *primed*. This simply means that the entries in the tape must be set to that of the first decent down the search space up to the first leaf, or stopping condition. The actual machine always starts at a leaf in the search.

The reasoning for starting at a leaf is that if a Hamilton cycle is found, the machine will stop at a leaf with the read/write head ready to move to the left. If exploring the entire search space is required, it is then easy to just re-enter the machine.

The function *primeTape* initializes the tape to the first leaf in the search space. The function returns true only when *runTurningMachine* can be entered. A false result is because either the first leaf represents a Hamilton cycle or a simple non-hamiltonian decision is detected such as a vertex with degree lower than 2. The source code in listing 8.2 describes the use of *runTuringMachine* in finding and counting Hamilton cycles.

Listing 8.1: The main Turing Machine method.

```
/* Run Turing Machine until a Hamilton cycle is found or search space is
exhausted.  Ensure that tape is primed first before calling.  Returns false
when search space exhausted. */

bool
runTuringMachine( HCStateRef s )
{
  HCTape  *hx;        /* read/write head for tape */
  Vertex  *d2, x;

  VArc    **L  = s->adjList;
  Vertex  *e   = s->virtualEdge;
  UInt    *d   = s->degree;
  Vertex  *nv = s->vertexOrder;  /* nv[x] refers to next vertex in list */

  s->flags.isHamiltonCycle = false;

  /* tape head must be at a leaf in search space, move tape head left to the
     closest branching edge */

  hx = unwindSearchEdge(L, e, d, s->pos);

  while ( !(hx->status & HC_TERMINATE) ) {

    /* remove the exhausted branching edge and switch directions*/
    x  = rotateAnchorPoint(s, L, e, d, hx, &d2);

    /* removal of branching edge may have created an inconsistent state */
    x  = ensureConsistent(s, L, e, d, d2, x, nv);

    if (x) {
      /* keep extending branching edges from anchor points */
      while ( extendAnchor(s, L, e, d, x) ) {
        do x = nv[x]; while (!d[x]);
      }
    }

    /* stopping condition encountered (a leaf), stop movement to the right */
    if (s->flags.isHamiltonCycle) return true;
    hx  = unwindSearchEdge(L, e, d, s->pos);
  }

  s->pos = hx;
  return false;
} /* runTuringMachine */
```

Listing 8.2: Examples of using the Turing Machine to count Hamilton cycles.

```
/* Find the next hamilton cycle in the search space for the graph referenced
in s. Returns false when no Hamilton cycle found. */

bool
nextHamiltonCycle( HCStateRef s ) {

  return runTuringMachine(s);
} /* nextHamiltonCycle */


/* Find the first Hamilton cycle in the search space. Returns false when no
Hamilton cycle found. */

bool
firstHamiltonCycle( HCStateRef s )
{
  resetStateAndRestoreGraph(s);

  if ( primeTape(s) && !runTuringMachine(s) ) return false;

  s->flags.isHamiltonian = s->flags.isHamiltonCycle;
  return s->flags.isHamiltonCycle;
} /* firstHamiltonCycle */


/* Count all Hamilton cycles in search space for graph referenced in s. */

ULongLong
hamiltonCycles( HCStateRef s )
{
  ULongLong c = 0;

  if ( firstHamiltonCycle(s) )
    do {
      c++;
    } while ( nextHamiltonCycle(s) );

  return c;
} /* hamiltonCycles */
```

8.2.2 Extending Segments and Branching

Segment extension can occur in two points in *runTuringMachine*. Both methods *ensureConsistent* and *extendAnchor*, found in Appendix A.3, call *extendSegments*, found in Appendix A.2.2.

The complete implementation of the algorithm for segment extension discussed in section 6.1.1 is contained within *extendSegments*.

The first method in *runTuringMachine* that can call *extendSegments* is *ensureConsistent*. Just as its name would suggest, *ensureConsistent* ensures that the current state of G_r is consistent after the removal of a branching edge.

The second method appearing in *runTuringMachine* that calls *extendSegments* is *extendAnchor*. This procedure ensures that the anchor point, 'x', has been removed from G_r by extending up to two branching edges off of it.

8.2.3 Restoring State and Removing Branching Edges

The restoration of G_r when ascending the search space is controlled by *unwindSearchEdge*, found in Appendix A.4. This method starts at the location in the search space where a stopping condition has just occurred and *'unwinds'* the changes to G_r one edge/arc at a time from the tape using the procedure *unrollArc*, also found in Appendix A.4. Once the closest branching edge, or the end of the tape has reached, *unwindSearchEdge* returns that location to *runTuringMachine* for use by *rotateAnchorPoint* and *ensureConsistent*.

The procedure *rotateAnchorPoint*, found in Appendix A.3, removes this branching edge from G_r and if necessary populates the stack of forced degree 2 vertices '$d2$' to be later processed by *extendSegments* when ensuring that the graph state is consistent.

Chapter 9

Pruning Algorithm

In any backtracking algorithm, if conditions exist that allow for entire branches of the search tree to be avoided with out affecting the outcome of the search, these conditions are referred to as *pruning* conditions.

The pruning conditions used in [Koc92] center around a powerful lemma about Hamiltonian graphs. This lemma appears in Bondy and Murty [BM76].

Definition. *The number of connected components in the graph G is $\omega(G)$.*

Lemma 1. *Let K be a separating set of the graph G. If $\omega(G) - |K| > 0$ then G is not hamiltonian.*

Searching for the existence of separating sets with this property is difficult as it would involve enumerating over the power set for $V(G)$ in some way. However, if one could be found with this property for some $G_{\mathcal{SE}}$ while searching for a Hamilton cycle it may be possible to determine how far back in the search space the property holds and continue searching from that position onwards. The ability to prune back the search space given a separating set that satisfies lemma 1 is described in section 9.1.

Having a way to prune back the search space using separating sets is useless without first having a nice way to find them. Certain conditions exist that allow for relatively quick linear detection of separating sets. Both cut points and bipartitions can be detected in this way. Cut points automatically satisfy lemma 1, as each cut point taken individually results in two or more components. In a bipartite graph, if the two sets differ in length, the smaller of the sets can be chosen as a separating set that satisfies lemma 1.

In [Koc92] a modified version of an algorithm by Hopcroft and Tarjan for finding cut points is used. It is a depth first traversal of the vertices in G_r that includes bipartition testing in addition to finding cut-points.

The approach used for pruning the search space has been improved upon in two ways in this thesis. First the scope of detecting cut points has been increased over the implementation for [Koc92]. The changes to the scope allow for the results from the bipartition condition to be compared against the results using cut points. By choosing the set that would return the largest difference that satisfies lemma 1 more pruning may be possible from that point in the search space.

For the second improvement, the point in the search space where detection of separating sets occurs has been changed. This change ultimately reduces the overhead of detecting separating sets without missing any pruning opportunities that would be found using the method described in [Koc92]. Section 9.3 describe this improvement.

9.1 Pruning Mechanism

As already stated the pruning condition provided by lemma 1 can be used to avoid searching entire branches of the search space. This section describes the mechanism used to prune back the search space once a separating set has been found for some G_r in the space.

Let G_r^{k+1} be a reference to some consistent G_r in the search space for G. Assume that the point G_r^{k+1} has a detectable separating set K, with $c = \omega(G_r^{k+1} - K) - |K|$.

Let $c > 0$, then lemma 1 is satisfied and G_r^{k+1} is not hamiltonian.

No more search is possible at G_r^{k+1} and movement up the search tree towards the closest anchor point must begin. Let K' be a new separating set formed from all of the vertices from K and select vertices from $V_{in}(S)$ as they are returned to G_r during the movement up the search tree. Recall from the previous work in this thesis, that vertices are returned to G_r in the reverse order in which they were removed.

Let v the next vertex to be restored to G_r on the way up the search space. v is added to K' so that no change could have occurred to the result of $\omega(G_r^{k+1} - K)$. After v is restored to G_r, v can be added to K' if $\omega(G_r^{k+1} - K) = \omega(G_r - (K' + v))$.

In this way the value of c will only decrease by 1 if v is added to K'.

Determining whether or not to add v to K' so that the number of components remain unchanging is difficult to do optimally. However a heuristic proven in [Koc92] can be applied based on the knowledge of how each v was removed from G_r.

Essentially v can be added to K' when the addition of v back into G_r does not involve any non-segment edges. Typically these are the forced vertices that led towards G_r^{k+1}. When v such as these are reattached to G_r only a virtual edge will have been pulled back a bit and no changes to connectivity will have occurred. Special care must be taken for any v that represent past anchor points on the path up the root of the search tree. Vertices that were once anchor points must be added to K' as they would involve the return of one or more branching edges.

This process of generating larger K' for each v reattached to G_r continues until c has been reduced to 0. G_r^k is the position of the closest anchor point at or above this point. Search for a Hamilton cycle must recommence at G_r^k. Figure 9.1 is an illustration of some hypothetical search space this process is applied to.

9.2 Testing for Separating Sets

9.2.1 Overview of Previous Test

As can be seen from the way the pruning condition climbs up the search space towards G_r^k, the largest possible value for c is desirable. Additionally as stated in the beginning of the chapter, the actual testing for a separating set must be reasonable in its overhead.

To satisfy the latter, [Koc92] uses an $O(n + \varepsilon)$ algorithm based on one developed by Hopcroft and Tarjan. If G_r is connected at the time of testing the algorithm produces a value for c in one of two ways.

The first is if a cut point is detected. In this case the value of c is set to be the number of components surrounding a single cut point reduced by one, as that is the length of the separating set. The choice of the single cut point is that of the lowest cut point in the first branch to contain a cut point in the depth first search.

The second way c is found in a connected G_r is if no cut point is found. In this case the entire graph will have been visited by the depth first search and a determination on whether or not G_r is bipartite can be made. In this case if a bipartition is found, c is set to the absolute value of the difference of the two sets found.

If G_r is disconnected, the depth first search is repeatedly called until all vertices in G_r have been visited. In this case c is set to the number of components visited. This is equal to the number of times the depth first search had to be called before all vertices were visited.

Afterwords if $c > 0$ the multi-path algorithm can stop for G_r and pruning can take place.

9.2.2 Increasing $\omega(G_r - K) - |K|$

If no separating set is found by the depth first search then the worst case performance will have occurred in running the test. Typically this will be the case for the vast majority of the times testing occurs.

Since pruning is the ultimate goal of the test, the algorithm for testing has been modified to always run in the worst case, but with the benefit of having a larger value of c for the pruning to take advantage of.

This larger c is obtained in the following ways:

1. *All* cut points are found in G_r and the subsequent components are calculated during the traversal of the graph by the depth first search.

2. The bipartition condition test is always able to run to completion if G_r is connected.

3. If disconnected, a running total for c is calculated for all components in G_r. Each value of c due to cut points found as in the first way contribute to the total.

If the first two cases occur, the largest value for c is chosen. The third case overrides all of the other cases as it would always be greater than case one, and the bipartition condition can not exist in a disconnected graph.

Please note that during the writing of this work, a way of using these disconnected components as they relate to bipartition was realized. This result is now discussed but not reflected in the code provided or in the analysis.

As in the third way of arriving at a value of c, if a bipartition was found in the disconnected component of G_r, its difference can then be used towards comparing against any cut points for the best separating set to use. If the difference is zero for the bipartition found, a value of 1 can still be used as the choice of simply using the single disconnected component still can be used.

Ensuring a proper count for the separating set size

The depth first search is always called on a consistent G_r. However it may additionally be called after a branching edge has been attempted. The subsequent removal of the branching edge may have a subtle impact on the size of the separating set found by the pruning condition test. If the anchor point attached to that branching edge was not absorbed into $V_{in}(S)$ after the removal of the branching edge, it must be considered part of the separating set for G_r^k. This is taken into account by the implementation when returning the value of c for the pruning operation.

9.3 Reducing the Frequency of Testing

9.3.1 Top Down vs. Bottom Up Testing

The algorithm from [Koc92] uses the top down approach to testing for separating sets. What this means is that just before each anchor point is explored and branching occurs, the test is run against the current consistent G_r for that anchor point.

By only having the tests occur on the extremities of the search space, significant speedups in terms of the time spent testing can be achieved.

In order to only run the test at the extremities of the search space, it must be shown that *no opportunities to prune the search space are missed with respect to the top down approach.*

Overlapping Components for G_r^{k+1} and G_r^{k+2}

Lower instances of G_r in the search space have at least as many components due to cut points as their predecessors, and those components that do exist, overlap.

Assume G_r^{k+1} is the point in the search space where a pruning opportunity would first be detected in the top down approach. Let G_r^{k+2} be an instance of G_r closer to the bottom of the search space that descends from G_r^{k+1}.

As mentioned before in section 9.1, $\mathcal{E}^{k+1} \supseteq \mathcal{E}^{k+2}$. Also any virtual edge only replaces existing connectivity in the graph. Subsequently any cut point in G_r^{k+1} will either still exist in G_r^{k+2}, or be removed. If the cut point was removed, then more disconnected components will be detected in G_r^{k+2}.

Additionally no additional connectivity in the graph can be introduced between

G_r^{k+1} to G_r^{k+2}.

By this reasoning only more components could be created from G_r^{k+1} to G_r^{k+2}.

Bipartitions in G_r^{k+1} and G_r^{k+2}

With similar reasoning with respect to $\mathcal{E}$, any bipartition that would be detected in G_r^{k+1} will also appear in G_r^{k+2} if no disconnection occurs in G_r before then.

Additionally due to the reduced edge count in G_r^{k+2}, the chance of finding a bipartition is greater and the search becomes faster. This is also the case for cut points as well.

9.3.2 Testing at the Extremities of the Search Space

The extremities of the search space for the purposes of this thesis is defined to be the locations of the leaf nodes in the search space. Recall that in chapter 4 leaf nodes were defined to be anchor points that only have leaves descending from them in the search tree.

The following strategy for when to run the test has been adopted by the implementation for this thesis: *The test for a separating set is run on the next consistent state found for G_r after a leaf node has been detected.*

Determining where leaf nodes are in the search space is accomplished in the following way:

1. On the start of the Turning Machine a marker called 'low', that points to a cell on the tape is set to point to the leftmost cell on the tape.

2. If the Turing machine has been moving to the right and encounters a stopping condition, movement commences to the left and stops at a branching edge, this represents an anchor point in the search tree.

   If the 'low' marker points at a cell to the left of the read/write head, it is updated to point to the current position of the read/write head. This represents the lowest anchor point in the tree so far.

   If the 'low' marker points to the current position of the read/write head it means that the same anchor point previously marked is still the lowest point found so far.

   If the 'low' marker points to a position to the right of the read/write head, it means that a leaf node has been reached, exhaustively searched and control is currently at the anchor point directly above where the leaf node was detected. A pruning flag can now be set.

3. If the pruning flag is set, when the next consistent state is reached the test for a separating set can be run, and the 'low' marker can be reset to the leftmost position of the tape.

Figure 9.2 demonstrates this with G_r^A being the point where a leaf node in the search has been detected. G_r^{k+2} is the next consistent state after G_r^A.

9.3.3 Ensuring a Full Climb up the Search Tree

Because the pruning mechanism is not optimal for a value of c, each position found up the search space by the pruning mechanism must be tested again when using the bottom up approach.

To do this, the pruning flag is not reset after pruning has taken place, so on the next consistent state, the test for a separating set is repeated and further climbing can continue. It is not until the test for a separating set comes back with no sets found that the flag can be turned off.

9.4 Implementation

The 'c' source code that implements all of the pruning functionality discussed in this chapter is fully included in Appendix B.

9.4.1 Data Structures

The data structures required to run the depth first search is listed and documented in appendix B.1. A reference to the state information required to run the DFS code is stored by the *HCStateRef* instance.

9.4.2 Code

A modified version of the *runTuringMachine* method called, *runTuringMachineWithPruning* is listed in appendix B.3.2.

Two extra methods along with the variables to control finding the leaf nodes in the search space have been introduced to *runTuringMachineWithPruning*.

The first method introduced is *getCompDiff*, found in appendix B.2.2. This method returns true if a separating set was found and returns the difference via the variable c.

The second method introduced is *pruneSearchSpace*, based upon the pruning mechanism presented in section 9.1, given the current cell position on the tape and the argument c, unrolls the search state to the closest branching edge at or above where c becomes zero. This method is found in appendix B.3.1.

The actual search for a separating set is performed by *dfsCutBipart*, found in appendix B.2.1, and is used solely by *getCompDiff*.

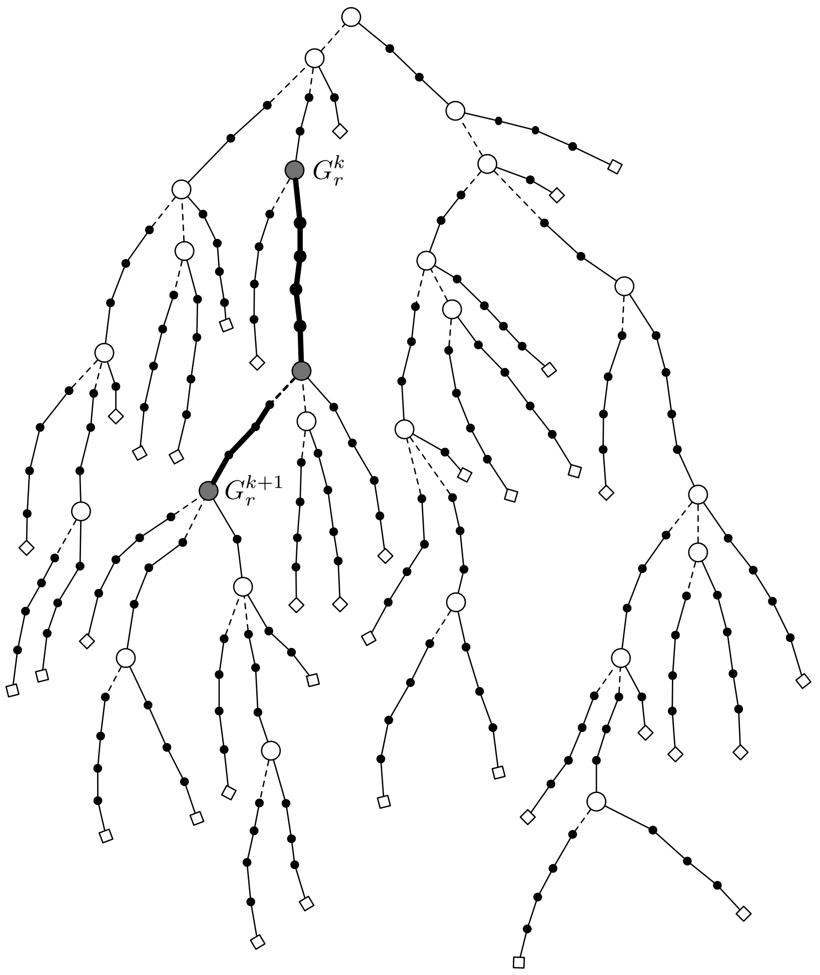


Figure 9.1: Hypothetical search space with pruning opportunity. Consistent graphs G_r^k and G_r^{k+1} are labeled. The smaller points between them represent the vertices that may make up the new separating set K'. In this example a separating set has been discovered at G_r^{k+1} that allows pruning up to G_r^k.

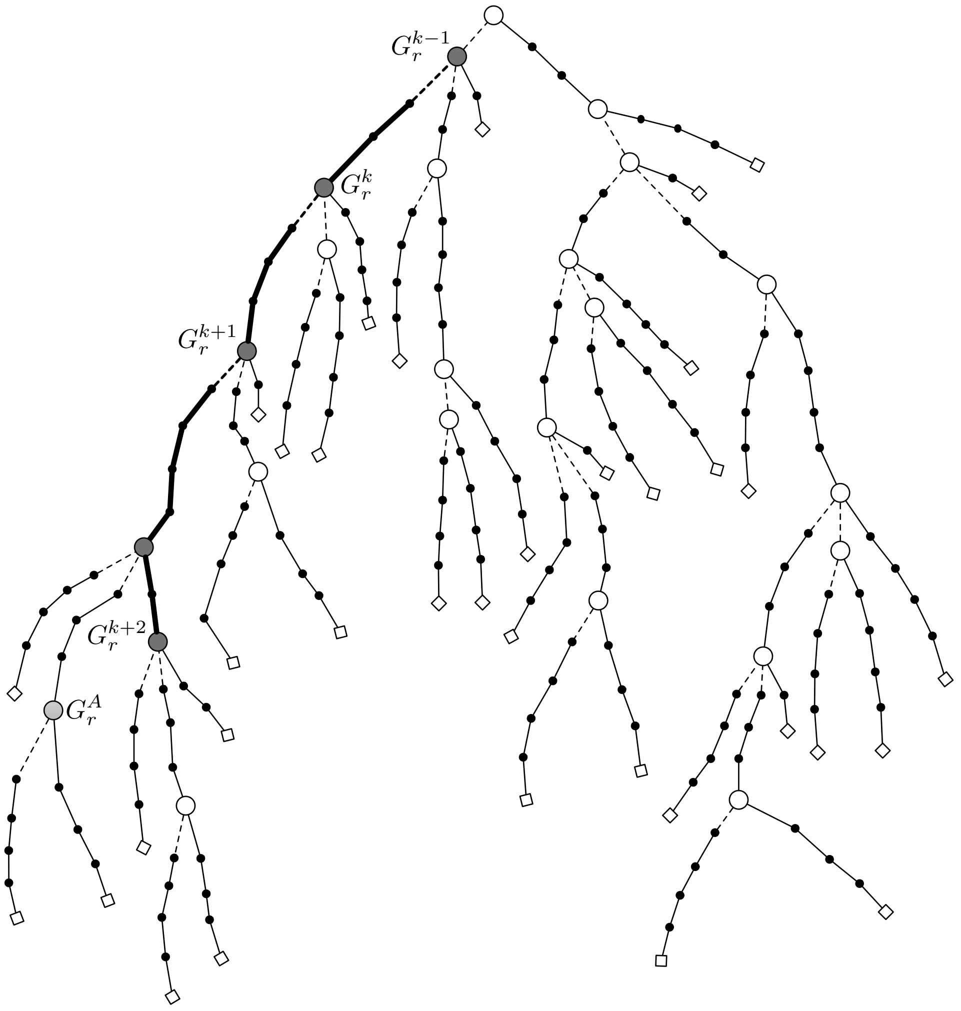


Figure 9.2: Hypothetical search space with pruning opportunity. A leaf node has been detected at G_r^A. A test for a separating set will occur at G_r^{k+2} and allow pruning all the way up to G_r^{k-1}.

Chapter 10

Reduction Technique

The previous work described in this thesis focused on general improvements to the multi-path algorithm that can benefit search with all graphs types. In this chapter a family of graphs is introduced that severely impact the performance of the multi-path method, despite the improvements already made.

A reduction technique is presented in this chapter that can be used to determine if a graph from this family is not Hamiltonian in an extremely short period of time. Further work is needed to use the technique in determining if a graph from this family is Hamiltonian and is discussed in chapter 12.

10.1 Hamiltonian Subgraphs

Definition (external vertex)**.** *Let A be some subgraph of the graph G. A vertex $v \in V(A)$ is said to be an external vertex of A if there exists an edge $\{u,v\} \in E(G)$ such that $u \in V(G - A)$.*

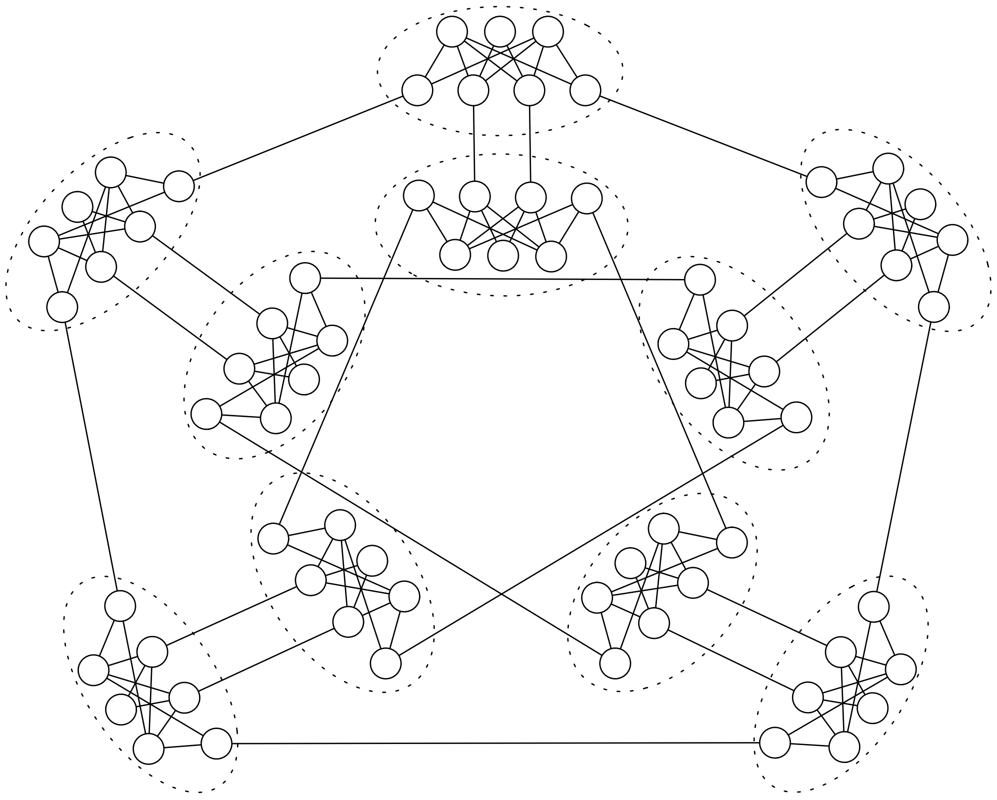


Figure 10.1: The Meredith graph. The multi-path algorithm is severely impacted in performance while searching this graph. This impact is mostly due to the properties of the marked subgraphs.

Definition (reducible subgraph)**.** *Let R be a subgraph from graph G. R is called a reducible subgraph if for any Hamilton cycle or path from G, a Hamilton path, $P \in R$, must exist between some external vertices from R, and P must be fully contained within the Hamilton cycle or path found in G.*

If a graph has a reducible subgraph, then it belongs to this family of graphs that can dramatically slow down the multi-path algorithm.

From the perspective of the rest of the graph when searching for Hamilton cycles, a reducible subgraph must be entered, fully traversed, and then exited, much like a single vertex is.

In figure 10.1 a representative of this family of graphs is shown. This graph, named the Meredith graph, contains ten reducible subgraphs that severely impact the performance of the multi-path algorithm.

The reason for the dramatic effect on performance is that the multi-path algorithm does not know about reducible subgraphs nor does it attempt to find a Hamilton path in one first before searching the rest of a graph. Instead multiple segments can be formed going into and out the external vertices of the individual reducible subgraphs. This can occur at many points in the search space. The state of G_r at each of these points needs to be exhaustively searched which represents an exponential period of time spent fruitless searching for a Hamilton cycle.

The subgraphs found in the Meredith graph are a special case of reducible subgraphs. Called *complete half-bipartite* subgraphs, section 10.1.1 details the specific reasons on why certain half-bipartite subgraphs are reducible. Section 10.1.2 briefly describes a fairly simple method of finding *complete half-bipartite* reducible subgraphs in a graph such as the Meredith graph.

Finally in section 10.2 a process of recursively using collections of disjoint reducible subgraphs to arrive at much a smaller reduced graph is described. These reduced graphs can then be used to determine if a graph from this family of reducible graphs is not Hamiltonian is a very short period of time.

10.1.1 Half-Bipartite Structures

Definition (half-bipartite subgraph)**.** *Let B be some subgraph of graph G. P is the maximal subset of the internal vertices of B such that no two vertices from P share an edge. If $|P| > 1$ then B forms a half-bipartite subgraph in G.*

Lemma 2. *Let P be the subset of internal vertices from a half-bipartite subgraph $B \in G$, with $p = |P|$ and $q = |V(B) - P|$. If $p \geq q$ then no Hamilton cycle can be found in G.*

Proof. Let x be the number of vertices from P visited by a path through B that starts and ends outside of P. Because vertices from $V(B-P)$ can be connected, at least $x+1$ vertices from $V(B-P)$ must be used in that path as no two vertices from P are connected.

If $p > q$ this means that the number of unvisited vertices from $V(B-P)$ will run out before those from P do. From the perspective of finding a Hamilton cycle in G, it becomes impossible to visit all of P, therefore G is not Hamiltonian.

If $p = q$ then a Hamilton cycle could be found in B, but in the process, detaching all of $V(B-P)$ from G with respect to finding a Hamilton cycle in G. Unless $B = G$, G is not Hamiltonian. □

Theorem 10.1.1. *Let P be the subset of internal vertices from a half-bipartite subgraph B, with $p = |P|$ and $q = |V(B) - P|$. If $p = q - 1$ then B is a reducible subgraph.*

Proof. Let B be a half-bipartite subgraph of G as described above and assume that a single cycle segment has entered and left B without visiting all vertices.

Then x visited vertices from P will be gone with respect to finding a Hamilton cycle in G and at least $x+1$ vertices gone from $V(B-P)$, leaving a new half-bipartite subgraph with $p' = p - x$ and $q' = q - x - 1$.

Since $p = q - 1$, then by substitution $p' = q - x - 1$ and $p' = q'$. By lemma 2 this new half-bipartite subgraph can not be used for any Hamilton cycle.

Therefore any cycle segment passing through B must completely visit B before leaving. Therefore B is a reducible subgraph.

□

Definition (complete half-bipartite subgraph)**.** *Let B be a half bipartite subgraph of G. If the subset P of internal vertices share an edge with each vertex from $V(B-P)$, then B is a complete half-bipartite subgraph.*

Each of the subgraphs found in the Meredith graph form a complete bipartite graph with four external vertices in one of the bipartitions and 3 internal ones in the other. These sizes satisfy the requirements for being a reducible half-bipartite subgraph.

10.1.2 Finding Reducible Complete Half-Bipartite Structures

Complete half bipartite subgraphs have the unique property that all of the vertices from P all share edges with the same set of vertices.

A fairly efficient polynomial time utility for finding half-bipartite graphs based upon this property has been created for finding all of the reducible half bipartite subgraphs from a graph.

The algorithm used to find the subgraphs is based on string comparisons of strings generated by listing all of the vertices adjacent to a given vertex in ascending order. By matching strings that repeat to the same half-bipartition, the set P can be generated for each half-bipartite subgraph in a graph.

Past knowing these steps, the design and implementation of the algorithm that implement this idea of string matching is not discussed further as it is beyond the scope this work.

The important points are that reducible complete half-bipartite subgraphs are detectable in polynomial time, and they can be used to generate a graph reduction as described in the next section.

10.2 Reducing G

Let Z_G be a collection of disjoint reducible subgraphs from G. A new reduced graph G' can be formed from G and Z_G by simply copying G into G' and replacing each subgraph, R, by a single vertex v, in G' and ensuring that all edges that connect to vertices from R are represented as edges incident to v, being sure to discard duplicates and loops.

This new graph G' can subsequently be scanned for its own collection, Z'_G, of disjoint reducible subgraphs and the process can be repeated for some new graph G'', and so on. Let α be the number of times a reduction is performed in this way.

The recursively reduced graph, G^{α}, can then be searched with the multi-path algorithm with an exponential increase in performance.

Theorem 10.2.1. *Let G^α be the ultimate graph produced though a series of reductions using disjoint reducible subgraphs from each intermediate graph and the original graph G. If G^α is non-hamiltonian, so too is G*

Proof. Let the series of graphs generated during the reduction of G towards G^α be $G^1, \ldots, G^{\alpha-1}$. Let G^α be a non-Hamiltonian graph. This means that no Hamilton cycle could be formed that enters or exits any of the subgraphs in $G^{\alpha-1}$. Even if Hamiltonian paths could be formed between two external vertices for any reducible subgraph in $Z_G^{\alpha-1}$, none of them could be used. Therefore $G^{\alpha-1}$ is non-Hamiltonian and by induction so is G. $\square$

The use of this technique on the Meredith graph results in a search performed on the Petersen Graph. As shown in chapter 4, this graph is processed by the multi-path with an extremely small search tree. As will be shown in section 11.2.3, simply running the algorithm on the original graph takes a very long time.

10.2.1 False positives: an Unwanted Side Effect of Reduction

If G^α is found to be Hamiltonian, there is no guarantee that G is as well. One of the reasons for this is similar to the proof given for theorem 10.2.1.

With any Hamilton cycle, C, from G^α, the two edges incident to any $v \in V(G^\alpha)$ that represent a reducible subgraph, $R \in G^{\alpha-1}$, must reflect edges from $E(G^{\alpha-1})$ incident to two separate external vertices of R. In fact many combinations of external vertices are possible from R with a single pair of edges incident to v. If none of those combinations of external vertices have a Hamilton path in R, then C does not reflect a Hamilton cycle in $G^{\alpha-1}$.

If this is the case for all C from G^{α}, $G^{\alpha-1}$ is not Hamiltonian.

The other reason why C may not reflect a Hamilton cycle in $G^{\alpha-1}$ is the case where the two edges incident to v only reflect a single external vertex, u, from R. This is caused because the reduction reduces all edges out of R to a single point. Two edges incident u may connect outside of R in a way that is not done through any other points from R. G^{α} does not care about this when a Hamilton cycle is being found, because to it v is a single vertex, not a collection of them.

Chapter 11

Analysis and Verification

11.1 Verification

Verification of the implementation of the multi-path algorithm was performed in two main ways. The first was by comparing the Hamilton cycle counts generated by the implementation for [Koc92] and this thesis. The second was by repeatedly modifying the search space for the same Hamiltonian graph and comparing the Hamilton cycle counts. If the counts stayed the same after many modifications it was assumed that the algorithm was working properly.

The search space was modified by randomly permuting the numerical labels of a graph's vertices and then sorting the adjacency lists in ascending order based upon the new labeling. These isomorphic graphs were then searched using a static vertex ordering of one through n.

Many thousands of permutations were used on the same graph to verify that the cycle counts were unchanging.

In this way the subtle impact of the current anchor point on the size of the separating set while pruning was initially discovered and finally solved. The process of trying to discover where and why the Hamilton cycle counts differed led to many of the improvements introduced in this thesis.

11.2 Analysis

All of the results listed in this section were generated by a PowerPC G5 running at 2.5 GHz. The operating system was Mac OS X 10.4.10 (Tiger). While the machine was a 2 * Dual Core Machine (4 processors), only one processor was used as the algorithm as implemented is purely a sequential one.

A unix command line utility was written using the attached source code from within this thesis to run the implementation of the algorithm from this thesis.

The previous version of the algorithm for [Koc92] was run from within a modified version of Groups and Graphs [gng]. This version was ported from CodeWarrior to the XCode programming environment in order to use the gcc compiler [ccg]. The version label of the compiler is ‘powerpc-apple-darwin8-gcc-4.0.1’.

All test data is based on object code generated using the same optimization parameters during compilation (gcc -fast -mcpu=G5 -mtune=G5).

The source code of the implementation for this thesis should be fully portable across BSD, Linux and other Posix compliant operating environments. No endianness issues exist in the code and it should compile on most computing architectures. Essentially if gcc has been ported with the standard set of c libraries, it should compile with little or no modification.

The testing is broken up into three parts. The first, and most extensive part, demonstrates the performance of the new algorithm versus the old using a set of graphs known as knight's graphs, based upon the movement of knight's on a variably sized chess board. As mentioned in [Van98], graphs of this type are suitable for comparative analysis of Hamilton cycle algorithms because they tend to be difficult to fully explore. These tests attempt to demonstrate the improvements brought on by the new segment extension algorithm, by the vertex selection modifications and the changes to the pruning algorithm.

The second part of the test attempts to compare the two algorithms performance against complete graphs. The characteristics of the complete graphs allow for an examination of the improvements brought by the removal of pass by value data.

The final section of the analysis is based on the reduction algorithm and its use on the Meredith graph, or more specifically on what the consequences are of not using it are.

11.2.1 Knight's Move Graphs

Table 11.1 introduces a set of the knight's graphs used for testing the implementations of the algorithm. Attributes such as number of vertices and edges along with the Hamilton cycle counts generated by the algorithms are listed.

Table 11.2 lists the timing results for various parameters on the implementations of both versions of the algorithm. Time t_{old} represents the only timing results from the previous implementation from [Koc92]. The times for t_{old} were produced using no pruning.

Table 11.1: Set of knight's move graphs used for primary test results. The Hamilton cycle counts found when exhaustive searching the entire search space for each graph is listed.

Graph	n	ε	Hamilton cycles
kn36898	64	136	3018210
kn35341	64	136	2112
kn4x8	32	64	0
kn5x6	30	62	8
kn5x8	40	90	44202
kn6x6	36	80	9862
kn6x7	42	98	1067638
kn7x4	24	86	207360000

Table 11.2: Times generated for the graphs from table 11.1. Each of the times listed represent different algorithm parameters in use. The parameters are discussed throughout the analysis. All times listed are in seconds.

Graph	t_{old}	t_{desc}	t_{max}	t_{asc}	t_{bu}	t_{td}
kn36898	7.58	1.67	2.42	1.78	1.68	3.21
kn35341	0.022	0.011	0.010	0.0086	0.019	0.023
kn4x8	0.00015	0.000077	0.00016	0.00007	0.000120	0.00013
kn5x6	0.00010	0.000026	0.000027	0.000063	0.000038	0.000034
kn5x8	0.087	0.026	0.029	0.026	0.026	0.039
kn6x6	0.022	0.0077	0.0079	0.0059	0.0081	0.011
kn6x7	2.42	0.74	0.78	0.53	0.76	1.18
kn7x4	943.01	61.18	220.64	41.79	41.25	80.37

The times generated with both t_{desc} and t_{asc} correspond to static orderings of the vertices with respect to descending and ascending degree sequences. The times in t_{max} represent times generated using a maximum degree vertex remaining to G_r when selecting the next anchor point. This is the method used from [Koc92].

The times represented by t_{bu} and t_{td} are the results of using the pruning algorithm during the search. Times from t_{bu} are for the bottom up approach of pruning the search space and t_{td} are an approximation of the top down approach applied to the implementation from this thesis. It is approximate to that used by [Koc92] in that the

Table 11.3: Various ratio's of the timing results from table 11.2.

Graph	t_{old}/t_{desc}	t_{old}/t_{max}	t_{max}/t_{desc}	t_{asc}/t_{desc}	t_{bu}/t_{desc}	t_{td}/t_{desc}
kn36898	4.54	3.13	1.45	1.06	1.00	1.92
kn35341	2.00	2.20	0.91	0.78	1.73	2.09
kn4x8	1.95	0.94	2.08	0.91	1.56	1.69
kn5x6	3.85	3.70	1.04	2.42	1.46	1.31
kn5x8	3.35	3.00	1.11	1.00	1.00	1.50
kn6x6	2.86	2.78	1.02	0.77	1.05	1.43
kn6x7	3.27	3.10	1.05	0.72	1.03	1.59
kn7x4	15.41	4.27	3.61	0.68	0.67	1.31

pruning tests are run after each anchor point is selected and just before *extendAnchor* is run. The subtle difference being that the top down pruning occurs after each consistent state is reached due to the selection of a *single* branching edge in [Koc92]. Both t_{bu} and t_{td} use the default descending degree sequence ordering for the vertices.

Table 11.3 provide some ratio's for comparing the performance of the algorithm based on the timings from 11.2.

Effects of Vertex Ordering

In general when comparing the results using the ratios provided in Table 11.3, the changes to the algorithm show a substantial speed up over the previous algorithm from [Koc92]. The exception seems to be in the case when comparing the 'kn4x8' graph with the old algorithm when using the max degree approach in the new algorithm. One possibility here is that because the graph is not Hamiltonian and because the search happens so quickly, the number of branchings that occurs must be relatively few in number. This suggests that the faster sequential copies of the pass by value approach offsets the other gains of the new algorithm when using the max degree approach when a lower number of branches occur. More on this is explored in the second part of the analysis.

One graph in particular seems to gain greatly by just using a descending degree sequence vs using the max degree approach. Graph 'kn7x4' has a great speedup over the original in this case. When removing the affects of the static orderings using t_{max}, the speed up stays in line with the other graphs with larger search areas.

By changing to an ascending degree sequence quite a few of the graphs show a speedup, particularly the 'kn7x4' graph, however the impact on the other graphs suggest that more than just degree values have an impact the size and shape of the search space.

Pruning Algorithm

The improvements brought by changing when to test for a pruning condition are made most apparent by the 'kn7x4' graph. Without the improvement, based on the results shown, it could be assumed that little or no pruning actually takes place in this graph. But the 0.67 ratio for the t_{bu}/t_{desc} suggests that quite a bit of pruning occurs.

In general the bottom up approach seems to provide quite a speed up compared to the top down approach. However the results for the 'kn5x8' graph suggest that in the cases where a smaller search space exists, a larger number of calls to the separating set test probably occur due to the nature of how the algorithm climbs up the tree in the bottom up approach.

11.2.2 Complete Graphs

Table 11.4: Comparison of both multi-path Algorithms using no pruning and the default vertex orderings on a set of complete graphs. All times are in seconds.

n	ε	t_{old}	t_{new}	t_{old}/t_{new}
12	66	11.12	4.19	2.65
13	78	136.00	50.53	2.69
14	91	1811.18	660.83	2.74
15	105	26848.33	9173.40	2.92

Table 11.4 contains the second set of comparative data of the two implementations based on tests of the complete graphs on 12 through 15 vertices. Because of the high degree values of each of the vertices in the complete graph, the impact of the new segment extension algorithm is lessened.

Additionally running the pruning algorithm on these graphs is pointless, as most of the leaves result in Hamilton cycles.

Because of these factors, more emphasis can be placed upon the removal of the pass by value data and the expected $O(n)$ improvements per branching edge. As can be seen in the data provided, as the number of vertices increase so does the speedup. While it is impossible to test, theoretically it would be expected that this speedup would approach n for larger values of n. The initially lower speedup can partially be attributed to the differences in random memory movements vs fast sequential ones, since the new algorithm has more random memory reads and writes while restoring the graph states. As the number of vertices increase, the size of the $O(n)$ sequential blocks used in the pass by value approach increase, and the benefit of the fast sequential copy of data loses ground to the lower $O(\Delta S)$ random memory copies.

11.2.3 Reduction Technique on the Meredith Graph

The Meredith graph is a non-hamiltonian graph that is extremely difficult for the multi-path algorithm to deal with. Much of the search space is spent with multiple segments going into and out of individual half-bipartite subgraphs. This is what the reduction technique is meant to stop. Once the Meredith graph is reduced to the Petersen graph, the time spent testing it is meaningless compared to the original.

While testing a previous version of the new algorithm, one that was closer to [Koc92] in performance, the algorithm spent over twenty five days running the algorithm before a power failure terminated the process. This was one of the prime motivations for moving towards an algorithm that could easily incorporate checkpointing.

The new algorithm without pruning takes 11 days to exhaustively prove that the Meredith graph is non-Hamiltonian. With pruning enabled, this drops to only 4 days! Because of the variability in the speed up between graphs, it is hard to surmise what the actual runtime would be for the implementation from [Koc92].

However 4 days verses less than a second is a good indicator of how powerful the reduction technique could be when fully developed.

Chapter 12

Further Work

During the course of developing the improvements described by this work many ideas of potential further study were envisioned with respect to the multi-path method, pruning the search space, and the reduction technique. A few of these ideas are listed here.

12.1 Multi-Path Algorithm

12.1.1 A Closer Examination of Vertex Order

A more extensive treatment of the static ordering of anchor point selection could allow for the identification of more properties of graphs and allow for further reductions in the size of the search space of the multi-path method.

12.1.2 Algorithm for Directed Graphs

The extension algorithm developed here could be adapted to work for directed graphs as most of the concepts carry over between graph types.

12.1.3 Fine Tuning for the TSP

As seen with the explosive growth in time to exhaustively search complete graphs, it is still not very likely that an exhaustive approach to the Traveling Salesmen Problem, or TSP, will be forthcoming with the results given here.

However the approximate results of the algorithms currently used by the TSP, could be used to seed the search space of the multi-path algorithm. In this way, by fixing certain segments and allowing others to change, a narrower area of the search space could be navigated and possible better solutions found.

12.1.4 A Work Stealing Parallel Implementation

Since most of the time searching is spent close to the right hand side of the tape when visualizing the search space with a Turing Machine, the left hand size of the tape could be used to offload work to parallel searches of the same graph.

A parallel algorithm could be developed that uses the approach of stealing branching edges from anchor points located on the left hand side of the tape.

The communications overhead of such an algorithm would be minimal and the execution would be extremely parallel, leading to potentially extremely good scaling characteristics.

12.2 Pruning Algorithm

12.2.1 Constructing Components for Better Separating Sets

The current mechanism for pruning back the search space approximates the changes to $\omega(G_r - K') - |K'|$. It would be nice to attempt to reconstruct the actual components of G_r while pruning back the search space. This could allow for better detection of what vertices to add to K' and even larger values of $\omega(G_r - K') - |K'|$ as each new member of the separating set may actually disconnect two components joined via the segment the new member came from.

By utilizing existing merge-find techniques for building up components, a method of creating a better separating set may be found.

12.3 Reduction Technique

12.3.1 Finding More Reducible Subgraphs

Finding reducible Complete Half-Bipartite Structures is a fairly limited endeavor for most graphs, especially when applied recursively as in the reduction technique. However reducible Half-Bipartite Structures that are not complete may occur with a great deal more frequency, especially in reduced graphs. Strategies for finding them should be explored.

12.3.2 Reconstructing Hamilton Cycles in $G^{\alpha-1}$

The utility of the reduction technique is currently limited to quickly processing otherwise hard non-hamiltonian graphs.

Since all Hamilton cycles in $G^{\alpha-1}$ must follow the route set out by Hamilton cycles in G^{α}, a way to recursively reconstruct Hamilton cycles from G^{α} up to G can probably be found and some work to that effect has already been started.

Bibliography

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory With Applications.* North Holland, 1976.

[ccg] GNU C Compiler. http://gcc.gnu.org/.

[Chr75] Nicos Christofides. *Graph Theory: An Algorithmic Approach.* Academic Press Inc, 1975.

[gng] Groups and Graphs. http://www.combinatorialmath.ca/G&G/index.html.

[Koc92] William Kocay. An extension of the multi-path algorithm for finding hamilton cycles. *Discrete Mathematics*, (101):171–188, 1992.

[Rub74] Frank Rubin. A search procedure for hamilton paths and circuits. *J. ACM*, 21(4):576–580, 1974.

[Van98] Basil Vandegriend. Finding hamiltonian cycles: Algorithms, graphs and performance. Master's thesis, University of Alberta, 1998.

[Wei95] Mark Allen Weiss. *Data structures and algorithm analysis (2nd ed.).* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1995.

Appendix A

Multi-Path Code

A.1 Data Types

```
typedef signed int    SInt;
typedef unsigned int  UInt;
typedef int           Vertex;

/* Adjacency List. */

typedef struct vertex_arc {
  Vertex            target;  /* Vertex adjacent to current vertex. */
  struct vertex_arc *next;    /* Next arc (null terminated list). */
  struct vertex_arc *prev;    /* Previous arc (circular list). */
  struct vertex_arc *cross;   /* Adjacent vertex's list. */
} VArc;

/* Graph Information Structure Adapted from Groups and Graphs. */

typedef struct graph {
  char     *name;            /* Graph title. */
  UInt      vertex_count;   /* Number of Vertices. */
  UInt      edge_count;     /* Number of edges. */
  VArc    **adj_lists;      /* Array of adjacency lists. */
  bool    **adj_matrix;     /* Adjacency matrix */
  UInt     *degree;         /* degree list */
} GraphInfo;

/* Flags controlling overall state of multi-path algorithm */

typedef struct hc_flags {
  bool usePruning;
  bool pruneOnce;
  bool setPruningFlags;
  bool isHamiltonian;
  bool isHamiltonCycle;
} HCFlags;

/* Bit-field reference for edge status stored in tape entries,
the status flags in code have mostly same meaning to those found in
section 7.3.  The main differences is if the anchor point
is the source vertex of the branching arc, the flag HC_FLIP_SOURCE is used.*/


typedef enum {
  HC_ENDPOINT       = 1,
  HC_ANCHOR_POINT   = 2,
  HC_ANCHOR_EXTEND  = 4,
  HC_FLIP_SOURCE    = 8,
  HC_FORCED_DEG2    = 16,
  HC_FORCED         = 32,
  HC_HAMILTONIAN    = 64,
  HC_TERMINATE      = 128
} HCArcStatus;

/* Storage Type of entries for each position in Turing Machine */

typedef struct hc_tape {
  HCArcStatus  status;
  VArc        *arc;
} HCTape;


/* Multi-Path State */

struct hc_state {
  HCFlags      flags;
  GraphInfo   *graph;              /* reference to target graph for search */
  HCDFSRef     dfs;                /* state for pruning algorithm */
  VArc       **adjList;            /* adjacency lists for each vertex */
  UInt         vertexCount;        /* initial number of vertices in graph */
  HCTape      *pos;                /* current position of read/write head */
  HCTape      *origin;             /* start of Tape for turning machine */
  UInt        *degree;             /* current degree of each vertex */
  Vertex      *virtualEdge;        /* current virtual edges */
  Vertex      *vertexOrder;        /* order for anchor point selection */
  VArc        *removedEdges;       /* current list of removed edges */
  VArc       **removedEdgesStack;  /* stack of lists of removed edges */
  Vertex      *deg2Stack;          /* forced vertex stack */
};

typedef struct hc_state * HCStateRef;
```

A.2 Extending Segments

A.2.1 Support Routines for Extending Segments

```
/* find and remove arc from adjacency list 'Lx' */

void
removeArc( VArc **Lx, VArc *a ) { /* ... */ }


/* insert arc to the top of adjacency list 'Lx' */

void
insertArc( VArc **Lx, VArc *a ) { /* ... */ }


/* Removes the bit flags that indicate an endpoint of a segment.
   Restores the incoming arc 'a' if necessary. */

void
fixInArc( VArc **L, VArc *a, Vertex x, UInt *kPtr )
{
  UInt k = *kPtr;

  if ( k & (HC_ENDPOINT ) ) {
    insertArc( L + x, a );
    k &= ~HC_ENDPOINT;
    *kPtr = k;
  }
} /* fixInArc */


/* Remove all but one of the incoming arcs of the source vertex of arc 'a'.
The initial incoming arc ( a->cross ) is not removed. */

bool
removeForcedD2InArcs( VArc **L, VArc *a, UInt *d, Vertex **d2Ptr )
{
  Vertex  y;
  UInt    dy;

  Vertex *d2 = *d2Ptr;
  VArc   *p  = a->prev;

  do {

    y  = p->target;
    dy = d[y];

    if ( dy == 2 ) break;
    if ( --dy == 2 ) *(++d2) = y;

    d[y] = dy;
    removeArc( L + y, p->cross );
    p = p->prev;

  } while ( a != p );

  if ( a != p) {

    /* deg 1 vertex encountered */
    /* restore from a to n before exiting */

    a = a->prev;
    while ( a != p ){
      y = a->target;
      d[y]++;
      insertArc( L + y, a->cross);
      a = a->prev;
    }

    return true;
  }

  *d2Ptr = d2;
  return false;
} /* removeForcedD2InArcs */
```

A.2.2 Main Routine for Extending Segments

```
/* Extend all segments for this state of the graph.

The extention of the segment is attempted from both sides of the segment until
no further extention possible with current graph state.

This procedure follows the initial vector provided by the arc 'a' and follows
the path until no degree 2 vertex is encountered or until a cycle is created
by hitting vertex 'z' ( the otherside of the segment ).

When the extention in current direction provided by arc 'a' can go no further,
the endpoint 'z' may have been reduced to degree 2. In this case the new
endpoint and z are swapped, a new initial arc is chosen at z and the process
is repeated until search terminated or until no further extention possible
from ether side of the segment.

Once a segment has been completely extended to its maximal length within it's
local domain, the process is repeated on a new segment until the degree 2
```

```
stack 'd2' is reduced to 0.

Procedure returns true if current state of graph terminates search, ie. when a
vertex is reduced to degree of 1 or a cycle is forced.

A cycle can only be forced when the current segment encounters its other
endpoint while trying to extend. */

bool
extendSegments(HCStateRef s, VArc *a, Vertex z, UInt k, Vertex *d2)
{
  Vertex   ex, x;
  VArc     *c;            /* cross arc directed at current endpoint */

  UInt     *d  = s->degree;
  Vertex   *e  = s->virtualEdge;
  VArc    **L  = s->adjList;

  HCTape   *hz = NULL;    /* tape position for other endpoint */
  HCTape   *hx = s->pos;  /* tape position for current endpoint */

extend_segment:

  /* attempt to traverse arc a */

  c = a->cross;
  x = a->target;

  /* store cross in tape so that in negative direction arc target always
     points at a vertex to be restored */

  hx++;
  hx->arc = c;

  if ( x == z ) {

    /* a cycle is forced */

    if (hz) fixInArc( L, hz->arc, z, &hz->status );

    /* restore focal point, no longer nessary to maintain tape position
       at this point */

    d[c->target] = 2;
    s->pos       = hx - 1;

    /* determine if cycle is a Hamilton cycle */

    hx++;
    s->flags.isHamiltonCycle = hx->status == HC_HAMILTONIAN;
    return false;
  }

  if (( ex = e[x] )){

    /* arc has collided with a virtual edge */

    /* attempt remove the target of arc (x) from graph and continue
       to extend the segment. */

    if ( d[x] > 2 ) {
      if ( removeForcedD2InArcs( L, c, d, &d2 ) ){

        /* ensure that segment endpoint info removed and
           that source of arc a is placed back on graph */

        if (hz) fixInArc( L, hz->arc, z, &hz->status );
        hx->status = k;
        s->pos     = hx;
        return false;
      }

      k |= HC_FORCED_DEG2;
    }

    hx->status = k | HC_FORCED;
    d[x]       = 0;

    if ( d[ex] != 2 ) {

      /* The other side of of virtual edge will not allow segment to
         continue in current direction */

      if ( d[z] != 2 ) {
        x = ex;
        goto finish_segment;
      }

      /* Otherside of segment is to be forced onto the cycle, grow
         segment in the x->z direction.

         Two possiblities on what arc to extend at z.

         The first is that z was a degree 2 vertex not on a virtual
         edge.  The L[z] arc is the arc first used to extend segment
         towards current x. Since L[z] is of degree 2 we know that the
         previous arc is the other arc pointing out of z.

         The second case is that L[z] is already on a virtual edge. In
```

```
           this case L[z] will only have one arc in its list and
           L[z]->prev == L[z].

           By choosing L[z]->prev in both cases an expensive branch is
           avoided.

           The same reasoning applies whenever L[z]->prev is chosen when
           switching endpoints */

        a    = L[z]->prev;
        d[z] = 0;

        if (hz) fixInArc( L, hz->arc, z, &hz->status );

        z    = ex;
        hz   = hx;
        k    = 0;

        goto extend_segment;
      }

      /* continue growing segment in the current direction */

      k     = 0;
      d[ex] = 0;
      a     = L[ex];

      goto extend_segment;
    }


    if ( d[x] == 2 ) {

      /* x is degree 2, add arc and continue in same direction */
      hx->status = k;
      k          = 0;
      d[x]       = 0;

      a          = c->prev;
      goto extend_segment;
    }

    /* Segment can not be extended further in current direction,
       remove arc pointing into segment. */

    hx->status = k | HC_ENDPOINT;
    removeArc( L + x, c );

    if ( d[z] != 2 ) goto finish_segment;
```

```
  a = L[z]->prev;

  if (hz) fixInArc( L, hz->arc, z, &hz->status );

  d[z] = 0;

  z     = x;
  hz    = hx;
  k     = 0;

  goto extend_segment;

finish_segment:

  /* A multigraph may have been created. Segment end points x & z may be
      physically adjacent along with their new virtual edge joining them.
      Remove actual edge to take away the multigraph status.    */

  if ( d[z] < d[x] ) {
    a = L[z];
    while ( a && a->target != x ) a = a->next;
  } else {
    a = L[x];
    while ( a && a->target != z ) a = a->next;
  }

  if ( a ) {

    c = a->cross;
    removeArc( L + a->target, c );
    removeArc( L + c->target, a );
    a->next = s->removedEdges;
    s->removedEdges = a;

    if ( --d[x] == 2 ) {

      d[z]--;
      d[x] = 0;

      a = L[x]->prev;
      fixInArc( L, hx->arc, x, &hx->status );
      k = 0;
      goto extend_segment;
    }

    if ( --d[z] == 2 ) {
      a     = L[z]->prev;
      if (hz) fixInArc( L, hz->arc, z, &hz->status );
      d[z] = 0;
      hz    = hx;
      z     = x;
      k     = 0;
      goto extend_segment;
    }

  }

  /* a new virtual edge has been created that is consistant within its local
     area of the graph (its endpoints) */

  e[z] = x;
  e[x] = z;

  /* check if any more segments need to be grown */

  do x = *d2--; while ( x && !d[x] );

  if ( x ) {

    /* there is a degree 2 vertex still on the stack,
       create a new segment or enlarge an existing one */

    if (( z = e[x] ))  d[x] = 0;    /* enlarge a segment */
    else  z = x;                    /* starts a new segment */

    a  = L[x];
    hz = NULL;
    k  = 0;

    goto extend_segment;
  }

  s->pos = hx;
  return true;

} /* extendSegments */
```

A.3 Extending Branches

```
/* The targeted arc represents a vertex already on a segment endpoint and
needs to have all incoming arcs removed but the arc opposite to 'a'.  Returns
updated stack of new forced vertices that are passed via d2. */

Vertex *
removeInArcs( VArc **L, VArc *a, UInt *d, Vertex *d2 )
{
  Vertex  x;
  VArc    *p = a->prev;
  while ( p != a ) {
    x = p->target;
    if ( --d[x] == 2 ) *(++d2) = x;
    removeArc( L + x, p->cross );
    p = p->prev;
  }

  return d2;
} /* removeInArcs */


/* Extend or create a segment and remove the vertex x from the graph by
marking one or two arcs extending out of it as pivot arc(s). The call to
extendAnchor must only be done while the graph is in a consistant state. */

bool
extendAnchor( HCStateRef s, VArc **L, Vertex *e, UInt *d, Vertex x )
{
  Vertex    y, ex;
  VArc     *a;
  UInt      k;

  Vertex   *d2 = s->deg2Stack;

  if (( ex = e[x] )){

  /* case 1: source vertex of current arc is already on a virtual edge
             and must be forced onto the potential hamilton cycle */

    *s->removedEdgesStack++ = s->removedEdges ;
    s->removedEdges         = NULL;

    a    = L[x];
    k    = HC_ANCHOR_POINT | HC_ANCHOR_EXTEND;
    d[x] = 0;
    return extendSegments( s, a, ex, k, removeInArcs( L, a, d, d2));

  }

  a = L[x];
  y = a->target;

  if (( ex = e[y] )) {

    /* case 2: same as first case but origin flipped */

    k = HC_ANCHOR_POINT | HC_FLIP_SOURCE | HC_ANCHOR_EXTEND;
    a = a->cross;
    *s->removedEdgesStack++ = s->removedEdges ;
    s->removedEdges          = NULL;

    d[y] = 0;
    if ( !extendSegments( s, a, ex, k, removeInArcs( L, a, d, d2) ) )
      return false;

    if ( !d[x] ) return true;

    *s->removedEdgesStack++ = s->removedEdges ;
    s->removedEdges          = NULL;

    a    = L[x];
    d[x] = 0;
    k    = HC_ANCHOR_POINT | HC_ANCHOR_EXTEND;

    return extendSegments( s, a, e[x], k, removeInArcs( L, a, d, d2 ) );

  }

  /* case 3: neither the source or the target of the arc is on a virtual
             edge, simple join them by a virtual edge and remove the
             two arcs joining them. */

  *s->removedEdgesStack++ = s->removedEdges ;

  removeArc( L + y, a->cross );
  removeArc( L + x, a );

  e[x] = y;
  e[y] = x;

  s->pos++;
  s->pos->arc    = a->cross;
  s->pos->status = HC_ANCHOR_POINT;

  *s->removedEdgesStack++ = NULL;
  s->removedEdges          = NULL;

  a    = L[x];
  d[x] = 0;
  k    = HC_ANCHOR_POINT | HC_ANCHOR_EXTEND;

  return extendSegments( s, a, y, k, removeInArcs( L, a, d, d2 ) );

} /* extendAnchor */


/* Removed edges form a null terminated singly linked list using the a->next value of each arc, sta
Restore each arc, using the a->cross to find out the origin.  Ensure to update degree values for ea
```

```
void
restoreEdges( VArc **L, VArc *a, UInt *d )
{
  Vertex  u, v;
  VArc    *n;

  while ( a ) {
    n = a->next;
    u = a->target;
    v = a->cross->target;
    insertArc(L + u, a->cross );
    insertArc(L + v, a );
    d[u]++;
    d[v]++;
    a = n;
  }
} /* restoreEdges */


/* Remove branch at hx.  Return the anchor point that the branching edge
was off of.  */

Vertex
rotateAnchorPoint( HCStateRef s, VArc **L, Vertex *e, UInt *d,
 HCTape *hx, Vertex **d2Ptr )
{
  Vertex  *d2 = s->deg2Stack;
  VArc    *a  = hx->arc;
  UInt     k  = hx->status;
  Vertex   x  = a->target;
  VArc    *c  = a->cross;
  Vertex   y  = c->target;

  if ( k & HC_ANCHOR_EXTEND ) {
    unrollArc( L, e, d, a, k );

    e[e[x]] = x;
    d[x]    = 2 + restoreInArcsWithCount( L, c, d );

    removeArc( L + x, c );
    removeArc( L + y, a );

  } else {

    e[x] = 0;
    e[y] = 0;
  }

  /* Restore edges removed during previous branch */
  restoreEdges( L, s->removedEdges, d);

  /* Remove the edge current anchor point represents and give it
  to the closest pivot point to the left to restore.  This indicates
  that no more search possible with the edge in question. ( or at least
  until pivot point to the left encountered.) */

  a->next          = *--s->removedEdgesStack;
  s->removedEdges = a;

  if ( --d[y] == 2 ) *(++d2) = y;
  if ( --d[x] == 2 ) *(++d2) = x;

  *d2Ptr = d2;

  s->pos = hx - 1;

  if ( k & HC_FLIP_SOURCE ) return y;
  return x;

} /* rotateAnchorPoint */


Vertex
ensureConsistent( HCStateRef s, VArc **L, Vertex *e, UInt *d, Vertex *d2,
Vertex x, Vertex *nv )
{
  Vertex ey;
  Vertex y  = *d2;

  /* no vertices have been forced, return x as next pivot point */

  if ( !y ) return x;

  /* forced vertices, ensure graph is consistant */

  ey = e[y];
  if ( ey ) d[y] = 0;
  else ey = y;

  if ( !extendSegments( s, L[y], ey, 0, --d2)) return 0;

  /* x may have been absorbed by a segment, ensure return of next
  available pivot */

  if ( !d[x] ) {
    s->pos->status |= HC_ANCHOR_TYPE1;
    do x = nv[x]; while ( !d[x] );
  }
  return x;
}



/* Initialize tape to first leaf in search space. Returns true only if
runTurningMachine can be entered.  */

static Vertex
primeTape( HCStateRef s, EList *requiredEdges )
{
  UInt      dx;
  Vertex    ex;

  Vertex    x   = s->vertexCount + 1;
  VArc    **L   = s->adjList;
  Vertex   *e   = s->virtualEdge;
  UInt     *d   = s->degree;
  Vertex   *nv  = s->vertexOrder;     /* nv[x] is the next vertex after x */
  Vertex   *d2  = s->deg2Stack; /* stack of forced vertices, 0 means empty */

  /* check for any degree 2 vertices, or stop condition */
  while ( --x ){
    dx = d[x];
    if ( dx < 2 ) return false;
    if ( dx == 2 ) *(++d2) = x;
  }

  /* force any degree 2 vertices onto segments */
  x = *d2;
  if ( x ) {

    ex = e[x];
    if ( ex ) d[x] = 0;
    else ex = x;

      if ( !extendSegments( s, L[x], ex, 0, --d2) )
      return !s->flags.isHamiltonCycle;
  }

  /* repeatedly place branching edges until stop condition reached */
  x = 0;
  do {
    do x = nv[x]; while ( !d[x] );
  } while ( extendAnchor( s, L, e, d, x ) );

  return !s->flags.isHamiltonCycle;

} /* primeTape */
```

A.4 Restoring Graph

```
static inline UInt
restoreInArcsWithCount( VArc **L, VArc *a, UInt *d ) {

  Vertex  v;

  UInt     c = 0;
  VArc    *p = a->prev;

  while ( p != a ) {
    v = p->target;
    insertArc( L + v, p->cross );
    d[v]++;
    p = p->prev;
    c++;
  }

  return c;
} /* restoreInArcsWithCount */


static inline void
unrollArc( VArc **L, Vertex *e, UInt *d, VArc *a, UInt k )
{
  Vertex x;

  /* Restore arc: y<---a----x  */

  if ( k & HC_ENDPOINT ) {
    x = a->cross->target;
    insertArc( L + x, a );
    e[x] = 0;
  } else if ( k & HC_FORCED ) {
    x     = a->cross->target;
    e[e[x]] = x;
    d[x]    = (k & HC_FORCED_DEG2)?
                  restoreInArcsWithCount( L, a, d ) + 2 : 2;
  }

} /* unrollArc */


static inline HCTape *
unwindSearchEdge( VArc **L, Vertex *e, UInt *d, HCTape *hx )
{

  UInt          k = hx->status;
  Vertex        x;
  VArc        *a;

  /* roll back graph state to closest anchor point */
  while( !(k & (HC_ANCHOR_POINT | HC_TERMINATE) ) ) {

    a = hx->arc;
    x = a->target;

    /**** NEGATIVE DIRECTION *****/

    /* restore vertex */

    unrollArc( L, e, d, a, k );

    d[x]    = 2;
    e[e[x]] = x;

    /* move tape head to the left and restart loop */
    hx--;
    k = hx->status;
  }

  return hx;

} /* unwindSearchEdge */
```

Appendix B

Pruning Algorithm

B.1 Data Types

```
/* Structure storing DFS state information.  This is for a non-recursive
dfs algorithm that processes the entire stack manually. */

struct dfs_compbipt {


  /*! Flag indicating if target graph is bipartite.  If initially set
      to false the bipartition algorithm is not run. */

  bool      bipartite;

  /*! Number of components found. */

  UInt      components;

  /*! Number of cutpoints found. */

  SInt      cutpoints;


  /*! Difference in bipartition set sizes. */

  SInt      bipartite_diff;

  /*! The current count (dfnum) of the number of vertices traversed by the
      dfs algorithm */

  UInt      depth;

  /*! Array indexed by vertices.  Indicates traversal number of each
      indexed vertex. */
  UInt    *visit;

  /*! Array indexed by vertices.  Indictates lowest traversal value
       encounterd by the indexed vertex during the dfs. */

  UInt    *low;

  /*! Array indexed by vertices. A non-zero value at a vertex's index
       indicates that said vertex is a cutpoint.  The value indicates
       the number of potential components surrounding the indexed vertex. */

  UInt    *branches;

  /*! Array indexed by vertices. Contains bipartite set membership for
       indexed vertices. */

  bool    *colour;

  /*! Array indexed by vertices. Contains node pointer of a vertex's arc
       pointing to the return vertex for the non-recursive dfs algorithm.
       The dfs algorithm in dfs_ccb uses the circularly linked prev
       value of a node to traverse the target graph. The value stored
       by the stop array is used to stop iteration over a vertex and
       ascend back up the search tree. */

  VArc   **stop;

  /*! Array indexed by vertices. Contains node pointer of next arc to
       traverse when control passes back to the indexed vertex during
       the dfs algorithm in dfs_ccb.  */

  VArc   **iterator;

  /*! Number of vertices structure allocated with */

  UInt stateSize;


  bool isXCutPt;

};
```

B.2 Depth First Search (DFS)

B.2.1 Main DFS algorithm

```
/*! Initialize dfs state for first call to dfsCutBipart

  \param s     dfs state to be initialized
  \param pts   Number vertices in unmodified target graph.
  \param bp    Flag if bipartition algorithm should be run
               by dfs_ccb to check if graph is bipartite.

*/

static void
initDFSCutBipart( HCDFSRef dfs, bool bp )
{
  UInt bl = dfs->stateSize * sizeof( UInt );

  memset( dfs->visit,     0, bl );
  memset( dfs->branches, 0, bl );

  dfs->depth      = 0;
  dfs->cutpoints  = 0;
  dfs->components = 0;
  dfs->bipartite  = bp;
  dfs->isXCutPt   = false;

  dfs->bipartite_diff = 0;

} /* init_dfs_compbipt */


/*!

  Traverses graph in a depth first search, finding components, cutpoints
  and bipartitions.

  This non recursive version requires entire stack to be allocated at once.
  This is handled by alloc_dfs_compbipt.

  Prior to the first call to dfs_ccb for a target graph, init_dfs_compbipt
  must be called.

  Procedure may be called repeatedly until returned value equals number
  of vertices in target graph.  This will ensure proper values for
  the number of components and cutpoints found in the graph. The
  starting vertex 'x' must be set to an unvisited vertex for these
  calls to dfs_ccb.

  The behaviour of dfs_ccb for subsequent calls after returned value equals
  number of vertices in graph is undefined unless dfs state is re-initialized.

  See documentation for 'struct dfs_compbipt' for details on
  how graph is traversed by dfs algorithm.

  Note: The initial low point of a vertex is set to its visit order rather
        than the visit order of the previous vertex.  This allows for
        proper component counts to be achieved.
  Note: An edge in an undirected graph is considered to be composed of two
        arcs,  x->y and y->x.  Variables of type VArc are usually refered
        to as 'arcs' by the documentation.


  \param s      Current dfs state
  \param L      Adjacency lists for target graph
  \param e      Virtual edges currently found for target graph
  \param x      An unvisited vertex to start traversal
  \returns      depth / visit number where dfs stopped searching

  \sideeffects

   Important values returned through struct dfs_compbipt:

    s->bipartite
    s->components
    s->cutpoints
    s->bipartite_diff


 This procedure is not threadsafe.

*/

UInt
dfsCutBipart( HCDFSRef dfs, VArc **L, Vertex *e, Vertex x )
{
  UInt       lx, vy;                    /* low pt of x, visit order of y */
  Vertex     y;                         /* destination of current arc */
  VArc      *cnx;                       /* current node/arc of interest */

  /* initialization */

  UInt      *v       = dfs->visit;       /* visit markers */
  UInt      *l       = dfs->low;         /* low points */
  VArc     **stp     = dfs->stop;        /* return arcs */
  VArc     **itr     = dfs->iterator;    /* arc iterators */
  UInt      *br      = dfs->branches;    /* branch counts */
  Vertex     z       = x;                /* start/stop vertex */
  VArc      *w       = L[x];             /* stop/return arc */
  VArc      *nx      = w;                /* next arc */
  Vertex     ex      = e[x];             /* other side of virtual edge */
  UInt       cp      = 0;                /* cut points found */
  UInt       cm      = 0;                /* components found */
  bool       bp      = dfs->bipartite;   /* bipartite flag */
  bool      *c       = dfs->colour;      /* bipartition set membership */
  bool       cx      = false;            /* x's colour/set value */
  SInt       bdiff   = -1;               /* difference in bipartite set sizes */
  UInt      d      = dfs->depth + 1;  /* current traversal depth */


  c[x]   = cx;
  stp[x] = w;
  v[x]   = d;
  l[x]   = d;

  goto start;

ascend:

  if ( nx ) goto iterate;

  /* finished with x, ascend */

  if ( x == z ) goto done;

  /* check if segment followed to get to this level */

  if ( ex < 0 ) {

    /* set destination and repair segment */

    y    = -ex;
    e[x] =  y;

  } else {

    y   =  w->target;

  }

  /* check for cut point, component, update low point */

  lx = l[x];
  vy = v[y];

  if ( lx >= vy ) {

    /* a branch has been found for y->x */

    br[y]++;

    /* if vertices after x encounter y, update component count,
       otherwise a x connects to y by a single edge only */

    if ( lx == vy ) cm++;

  } else if ( l[y] > lx ) l[y] = lx;
  /* update cut points */

  if ( br[x] ) cp++;

  /* advance */

  x  = y;
  ex = e[x];
  nx = itr[x];
  w  = stp[x];

  goto ascend;


iterate:

  /* iterate to next arc */

  if ( nx != w ) goto start;

  /* possibly finished with x, mark current and next arc as completed */

  cnx = nx = NULL;

  /* a segment edge may still have to be checked */

  if ( ex > 0 ) y = ex;
  else goto ascend;

  goto check;

start:

  cnx = nx;
  y   = nx->target;
  nx  = nx->prev;

check:

  /* check if visited, update low points */

  vy = v[y];

  if (!vy) goto descend;
  if ( vy < l[x] ) l[x] = vy;
  if ( bp && cx == c[y] ) bp = false;
  goto ascend;

descend:
  /* vertex y is unvisited, initialize and descend into it */

  itr[x] = nx;
  x      = y;
  v[x]   = ++d;
  l[x]   = d;
  ex     = e[x];

  if ( bp ) {

    /* bipartite condition still holds, init colour of x */

    cx     = !cx;
    c[x]   = cx;
    bdiff += (cx)? 1 : -1;

  }

  /* ensure that return arc set correctly */

  if ( cnx ) {

    w   = cnx->cross;
    nx  = w->prev;

  } else {

    /* decend by following segment, mark return path by negating */

    w     =  L[x];
    nx    =  w;
    ex    = -ex;
    e[x] =  ex;
  }

  stp[x] = w;

  goto start;

done:

  if (( dfs->isXCutPt = (br[z] > 1) )) cp++;

  dfs->cutpoints      += cp;
  dfs->components     += cm;
  dfs->bipartite       = bp;
  dfs->bipartite_diff = (bp)?bdiff:0;
  dfs->depth           = d;
  return d;

} /* dfs_ccb */
```

B.2.2 Routine for calculating Component / Separating Set differences.

```
/*! \returns true if a pruning condition found.  The difference in the number
 of components vs separating set is returned in c.  The inSepSet flag indicates
that x should be considered a part of what ever separating set found.  */

bool
getComponentDiff( HCDFSRef dfs, VArc **L, Vertex *e, UInt *d, Vertex *nv,
Vertex x, SInt *c, bool inSepSet )
{
  SInt     m1, m2;

  bool     xCut  = false;       /* is x a cut point */
  UInt    *v     = dfs->visit;
  UInt     left  = (UInt)*c;

  /* initialize dfs for first call with current version of graph */

  initDFSCutBipart( dfs, true );

  /* find components, cutpoints and bipartition */

  if ( dfsCutBipart( dfs, L, e, x ) < left ) {

    xCut = dfs->isXCutPt;

    /* disconnected graph, find any remaining cut points, components */

    dfs->bipartite      = false;
    dfs->bipartite_diff = 0;

    /* keep running dfs on unvisited vertices until all visited */

    do {
      while ( v[x] || !d[x] ) x = nv[x];
    } while ( dfsCutBipart( dfs, L, e, x ) < left );


    m1  = dfs->components - dfs->cutpoints
        - ((!xCut && inSepSet )?1:0);

    *c = m1;
    return true;
  }
```

```
xCut = dfs->isXCutPt;

/* either components and cut points, or a bipartition with non-zero
   difference in set sizes has been found. */

if ( dfs->components <= 1 ) {

  m2 = dfs->bipartite_diff;

  if ( !dfs->bipartite || !m2 ) return false;


  if ( m2 < 0 ) {

    /* swap which set is M and which is S */

    m2 = -m2;

    /* When the dfs starts, x is on the 'negative' side of the set
       difference, if an edge reduction occurs the other vertex incident
       to that reduced edge may be in the same set as x.  In this case
       too many components will be reported.  This is made worse because
       more than one edge may have been reduced about x at this point.

       To deal with this treat x as a member of the separating set
       rather than a member of the components resulting from the
       separating set.  Let M be the set of components.
       Let S be the separating set.  In the bipartite condition
       M and S are made up of individual vertices that exactly
       correspond to the two bipartite sets.

       If x originally is in M and we want to move it to S.
       So the following occurs to the difference between the
       sizes of M and S.

       (|M| - 1 ) - (|S| + 1 ) == |M| - |S| - 2

       So we must reduce the known difference by 2.

       */

      if ( inSepSet ) m2 -= 2;
  }


  *c = m2;
  return true;
}
```

```
  m1 = dfs->components - dfs->cutpoints - ((!xCut && inSepSet )?1:0);
  m2 = dfs->bipartite_diff;

  if ( dfs->bipartite && m2 ) {

    if ( m2 < 0 ) {

      m2 = -m2;

      /* see above for why next step required */

      if ( inSepSet ) m2 -= 2;

    }

    /* we have two choices for ( M, S ) here */
    /* choose set with larger difference between M and S */

    *c = ( m1 > m2 ) ? m1 : m2 ;

    return true;
  }

  *c = m1;

  return true;

}
```

B.3 Turing Machine for Pruning Condition

B.3.1 Pruning

```
/* return true if unwound all the way back to initial graph state */

static HCTape*
pruneSearchSpace( HCStateRef s, SInt c )
{
  HCTape  *hx;

  UInt     *d  = s->degree;
  Vertex   *e  = s->virtualEdge;
  VArc    **L  = s->adjList;

  HCTape  *stop = hx = s->pos;
  UInt      k    = stop->status;

  while ( !( k & HC_TERMINATE ) && (c > 0) ) {
    if ( k & HC_ANCHOR_TYPE1 ) c--;
    if ( k & HC_ANCHOR_POINT ) c--;
    if ( k & HC_FORCED_DEG2 )  c--;

    stop--;
    k = stop->status;
  }

  stop++;

  hx = unwindSearchEdge( L, e, d, hx );
  while (  hx > stop  ) {
    restoreAnchorPoint( s, L, e, d, hx );
    hx = unwindSearchEdge( L, e, d, hx - 1 );
  }

  return hx;
}
```

B.3.2 Main Turing Machine Code

```
/* Run Turing Machine until a Hamilton cycle is found or search space is
 exhausted.  Ensure that tape is primed first before calling.  Returns false
 when search space exhausted.

 The version of the Turing Machine for the Multi-Path algorithm that
 implements the pruning algorithm using the bottom-up approach to the
 pruning. The leaf node, or lowest anchor point, is tracked using the
 lowCell and highCell variables.  The lowCell variable is updated to always
 point to the lowest branching edge in the tape.  If hx is ever found to be
 higher up the search space then lowCell, an exhustively searched leaf node
 will have been found. At this point the prune flag can be set and a test
 for a pruning condition can occur at the next consistent state in the search
 space.  One caviot is that some leaf nodes will be missed when re-entering
 the machine after finding a Hamilton cycle.  Generally the leaf nodes
 closest to the stopping condition that found the Hamilton cycle are avoided,
 as no separating set is likely to be found that satisfies the pruning lemma.
 */

static bool
runTuringMachineWithPruning( HCStateRef s )
{
  Vertex  *d2, x1, x, v;
  SInt     c;
  HCTape  *hx;        /* read/write head for tape */


  HCTape  *highCell = s->origin;     /* tracks reset point for lowCell */
  HCTape  *lowCell  = highCell;      /* tracks leaf node in search space */
  bool     prune    = false;         /* indicates pruning should occur */

  UInt    *d  = s->degree;
```

```
Vertex  *e  = s->virtualEdge;
Vertex  *nv = s->vertexOrder;
VArc   **L  = s->adjList;

s->flags.isHamiltonCycle = false;

/* tape head must be at a leaf in search space, move tape head left to the
   closest branching edge */

hx = unwindSearchEdge(L, e, d, s->pos);

while ( !(hx->status & HC_TERMINATE) ) {

  /* remove the exhausted branching edge and switch directions*/
  x1 = rotateAnchorPoint(s, L, e, d, hx, &d2);

  /* removal of branching edge may have created an inconsistent state */
  x  = ensureConsistent( s, L, e, d, d2, x1, nv );

  if ( x ){

    if ( prune ) {

      /* reset leaf node */
      lowCell = highCell;

      /* count current number of vertices left in reduced graph */
      c     = 1;
      v     = x;
      while (( v = nv[v] )) if ( d[v] ) c++;

      /* get component difference from separating set test

        Note that x==x1 refers to a change in anchor points
        when consistency was ensured after the last anchor point
        rotation.  The last anchor point x1 was absorbed into
        the partial cycle found so far.  This means that when
        x and x1  are different  , x does not have to be considered
        part of the separating set, as no branching edges will
        have been removed from it yet. */

      if ( getComponentDiff( s->dfs,L,e,d,nv,x,&c, x==x1) ){

        /* prune back the search space as far as c lets us */
        hx = pruneSearchSpace( s, c );

        /* stopped at an anchor point, keep testing for
           pruning condition until none are found */
        continue;
      }
```

```
        /* no more testing needs to occur for this level */
        prune = false;
      }

      /* keep extending branching edges from anchor points */
      while ( extendAnchor( s, L, e, d, x ) ){
        do x = nv[x]; while ( !d[x] );
      }
    }

    /* stopping condition encountered (a leaf), stop movement to the right */
    if ( s->flags.isHamiltonCycle ) return true;
    hx   = unwindSearchEdge( L, e, d, s->pos );

    /* ensure that lowCell refers to the lowest anchor point and check if
       a leaf node has been exhausted. */

    if ( hx > lowCell ) lowCell = hx;
    else prune = hx < lowCell;

  }

  s->pos = hx;
  return false;

} /* runTuringMachineWithPruning */
```