Graph Searching

by

Richard Melvin Krueger

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Graph Searching

Richard Melvin Krueger

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2005

Graph searching is a simple and widely used technique for exploring the vertices and edges of a graph. A graph search starts by visiting an initial vertex and systematically visits new vertices by iteratively traversing edges incident with a vertex previously visited. The order in which vertices are discovered yields an ordering of the vertices of a graph.

Two well-known forms of graph search are Breadth-First Search and Depth-First Search. Recently, two restricted versions of graph search, Lexicographic Breadth-First Search and Maximum Cardinality Search, have been applied to a wide variety of problems. Many of these results rely on simple characterizations of vertex orderings that these algorithms can compute.

We find similar characterizations of vertex orderings for all other major graph search paradigms. These simple characterizations give a unified view of graph searching, and lead to the identification of two new search paradigms, Lexicographic Depth-First Search and Maximal Neighborhood Search.

To further unify our understanding of graph searching, we introduce a generalized technique called Maximal Label Search to express graph labelling algorithms. We characterize labelling schemes that correspond to each of the major search paradigms using our vertex ordering characterizations. We show that labelling schemes can capture some, but not all, types of graph search on complements of graphs without the need to compute the complement.

We illustrate the power of our new characterizations by addressing the problem of finding minimal triangulations of arbitrary graphs. We show that a class of algorithms derived from some of the main graph search paradigms can compute minimal elimination orderings, and show how this generalizes many known results.

Acknowledgements

I would like to thank the many people who have influenced and supported me as I studied at the University of Toronto. I am greatly indebted to you all, though I can only name a small subset of you here.

First of all, I am very grateful to my advisor, Derek Corneil, for all the time he devoted to supporting this dream. I will never forget the moment he suggested that I could write this entire thesis in just two weeks. It may have taken me somewhat longer than that, but I could always count on his superb guidance whenever it was needed.

I especially want to thank my collaborators, Anne Berry and Geneviève Simonet. It was a pleasure working with them, and I have benefited greatly from their insights.

I also want to acknowledge all the work the members of my committee have put into reading and critiquing my research and this thesis. I particularly want to thank Faith Fich for all her excellent suggestions, which greatly improved this thesis.

I have been fortunate to have such a wonderful group of fellow students and colleagues surrounding me during this period. Whether you are one of my colleagues in the department, fellow representatives at the GSU or SGS, ultimate (frisbee) teammates, or fellow karate-ka, I have cherished our relationship and each and every distraction you provided to my research. Thank you, Harold, for making my time here bearable.

My parents, Felix and Joan, have been an unwavering source of support throughout my many years of school. I hope I can make you as proud of me as I am of you. And to David and Kimberley, you have been wonderfully supportive siblings.

Finally, I want to express my appreciation for the financial support I received from the National Science and Engineering Research Council, the Ontario Graduate Scholarship Program, and the University of Toronto, Department of Computer Science, without which I would not have embarked on this fabulous journey.

Richard Krueger

Toronto, September 2005

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Graph search is fundamental. Most graph algorithms employ some mechanism for systematically visiting the vertices and edges in a given graph, and graph search is often used as an easy and simple tool to accomplish this. Graph search is also an incredibly efficient way to gather particular information about the structure of graphs. The simplicity of these algorithms allows even novice programmers to implement basic graph searches to run in asymptotically optimal time.

Graph search was originally published as a method for exhaustively searching a maze and ensuring that some solution would be found [Luc82, Tar95, Moo59]. The idea behind graph search is simple. After choosing an initial vertex, a search of a connected graph visits each of the vertices and edges of the graph such that a new vertex is visited only if it is adjacent to some previously visited vertex. The order in which we first visit each vertex gives an ordering of the vertices of the graph.

We observe that, while executing a graph search, it may be that there are several different vertices that could be visited next. Several strategies exist to help choose among these vertices, each exhibiting properties that may be desirable. The *breadth-first search* (BFS) and *depth-first search* (DFS) algorithms are two traditional strategies to choose the next vertex to visit. BFS tries to traverse an untraversed edge incident with the least recently visited vertex, whereas DFS tries to traverse an untraversed edge incident with the most recently visited vertex. BFS ensures that vertices that are nearby are visited first, in contrast to DFS which tries to follow a path through the graph as long as possible before backtracking.

In the mid 1970s, Rose, Tarjan and Lueker [RTL76] introduced a restricted form of BFS for determining perfect elimination orderings of chordal graphs. Their *lexicographic breadth-first search* (LexBFS) has garnered much attention recently and has become an important search algorithm used for recognition of various graph families, diameter approximation, and finding key structures in certain graph classes [Cor04a]. One of the key points regarding LexBFS, and one of the reasons for its usefulness, is that there is a very simple characterization of the vertex orderings that can be computed by LexBFS [BDN97]. This characterization is critical to the proofs of correctness of various algorithms. Other than LexBFS, only *maximum cardinality search* (MCS), presented in Tarjan and Yannakakis [TY84], has a known characterization of the orderings it may compute [BDN97].

In light of the power of such a vertex characterization, it is somewhat surprising that the traditional searches of BFS and DFS are not known to have similar characterizations. Despite the heavy use of these algorithms over the past half century, our understanding of graph search is incomplete. The goal of this work is to fill in our understanding of the nature of graph search orderings and give us a more complete view of the properties which are shared among or differentiating various graph searches.

Our work on characterizing these graph search orderings leads to the discovery of two new varieties of graph search: *lexicographic depth-first search* (LexDFS) and *maximal neighbourhood search* (MNS). The first, LexDFS, answers a long-standing question within the graph search community of whether there is a depth-first analogue to LexBFS. The second, MNS, explains why algorithms such as LexBFS and MCS have both been so successfully applied to a variety of areas, such as separation and minimal triangulation.

These two algorithms not only complete our hierarchy showing the relationship between the different types of graph search algorithms, but also lead to extensions that can efficiently compute minimal triangulations of arbitrary graphs, addressing an important problem in graph theory [Heg04].

All these graph search algorithms can be simply described as variants of a single paradigm. Conceptually, each type of search relies on some temporary labelling of unvisited vertices during the search to help direct the search and ensure the search completely explores the graph. In this work we formalize the concept of a vertex labelling algorithm, a paradigm that includes all our basic graph search algorithms. With such a universal view, interesting properties and generalizations can be identified for a variety of algorithms. We use this concept to address issues of implementation, and show how problems related to graph search can be solved in the same framework.

This work aims to lay a modern framework for graph search. We develop a hierarchy of search paradigms based on simple vertex characterizations. We establish a common algorithmic basis that captures not only graph search but also a wider family of vertex labelling algorithms. We provide tools, which have demonstrated their power numerous times in restricted settings, that could expand the domain of graph search and enhance its position as a fundamental algorithm in computer science. We show but a few examples suggesting the potential. The rest, as they say, is still to come.

1.1 Outline of the thesis

We now briefly outline the structure of this thesis. After a short chapter defining our graph search terminology and reporting on related work, we describe in Chapter 3 our characterizations of graph search orderings. From this work we derive two new search algorithms, LexDFS and MNS, and present our unified hierarchy of graph search.

In Chapter 4 we view graph search from an algorithmic perspective. We introduce a generic vertex labelling algorithm called Maximal Label Search (MLS) that captures each of the major graph search paradigms, and show that it also captures some generalizations of graph search. This single unified algorithm leads us to consider implementations of graph search in Chapter 5. We address the question of efficient implementation of our labelling algorithm, and derive label-free implementations of our new graph search paradigms, LexDFS and MNS.

For the remainder of the thesis we turn to illustrating the power of our unified views of graph search. In Chapter 6 we address the question of efficiently finding particular vertex orderings of graphs derived from an input graph via some function. We show how our labelling algorithm can be used to find graph search orderings of the complement of a graph without computing the complement, and how our ordering characterizations can help us to determine when a search ordering of a graph is also a search ordering of its power graphs, again without computing the power graphs. We further apply our unified views of graph search in Chapter 7 where we address the important problem of finding minimal triangulations of graphs. We devise an algorithm based on MNS, and show how this algorithm generalizes all known results on finding minimal elimination orderings with search-based algorithms. We discover that the set of minimal triangulations computable by such algorithms is surprisingly robust. We conclude the thesis with some words about various extensions of the unification results of Chapter 3 and provide some directions for future work.

Portions of the work reported in this thesis were completed in collaboration with coauthors and are in the process of being disseminated. We will follow the convention that results primarily due to coauthors will be cited and generally have their proofs omitted, whereas results that are stated without citation are either original or discovered independently by the author of this thesis.

Chapter 2

Background

2.1 Basic graph notions

We introduce the following terminology to be used freely throughout this thesis. Let V be a set. We use $P(V)$ to denote the power set of V, which consists of all subsets of V. The pair $G = (V, E)$ is a *directed graph* (or *digraph*) on the vertex set V and the edge set E if $E \subseteq V \times V$. The pair $G = (V, E)$ is an *undirected graph* on the vertex set V and the edge set E if E consists of unordered pairs of vertices in V (a subset of the two-element subsets in $P(V)$). The pair $\mathcal{H} = (V, \mathcal{E})$ is a *hypergraph* on the vertex set V and hyperedges $\mathcal{E}$ if $\mathcal{E} \subseteq P(V)$ (so a hyperedge is of arbitrary size).

Suppose $G = (V, E)$ is a graph. We say $n(G) = |V|$ is the order of G, and $m(G) = |E|$ is the size of G. When the graph G is clear from the context, we simply write n and m. If $|V|$ is finite, we say G is finite. If $(u, v) \in E$, we say (u, v) is an edge of G with endpoints u and v, and abbreviate it to uv; we use this notation for both undirected and directed graphs. If G is a directed graph, we say $uv \in E$ is an arc, to remind the reader that the edge is directed. If $v \in V$, we call vv a loop, and say G is loopless if it contains no loop. If an edge may be repeated in E (i.e., E is a multiset), we say G is a *multigraph* (for either the directed or undirected case) and if an edge uv appears multiple times in

E, we say uv is a parallel edge. If G is loopless and contains no parallel edges, we say G is *simple*. Unless otherwise stated, all graphs in this thesis are assumed to be finite, undirected and simple.

Suppose $H = (U, F)$ is a graph. We say H is a *subgraph* of $G = (V, E)$ if $U \subseteq V$ and $F \subseteq E$. A subgraph H of G is a proper subgraph of G if $G \neq H$. If U is a subset of V, we define $G[U] = (U, \{uv \in E : u, v \in U\})$ as the subgraph of G *induced* on the vertex set U. The *neighbourhood* of a vertex $v \in V$ in an undirected graph G is $N_G(v) = \{u : uv \in E\}$. A sequence $P = (v_1, \ldots, v_k)$ of vertices is a *path* in G if $v_1v_2, v_2v_3, \ldots, v_{k-1}v_k \in E$. These edges are the edges of the path P and the length of P is the number of edges in the path. If $u = v_1$ and $v = v_k$, we say P is a u, v-path. A path is *simple* if its vertices are distinct. The *distance* $d(u, v)$ between two vertices u and v in G is the minimum length of a u, v-path, or ∞ if no u, v path exists. The *diameter* of a graph G is the maximum distance between two vertices of G. A graph G is *connected* if there is a path between every pair of vertices of G. A maximal set of vertices of G such that there is a path between each pair is called a *connected component* (or component) of G. A sequence $C = (v_1, \ldots, v_k)$ is a *cycle* in G if $v_1v_2, \ldots, v_{k-1}v_k, v_kv_1 \in E$; these are the edges of C and the number of edges in C is its length. A *chord* of a cycle (resp. path) is an edge between two vertices of the cycle (path) that is not an edge of the cycle (path). A cycle (or path) is chordless if it has no chord. A graph containing no cycles is called a *forest*, and a forest with one component is called a *tree*.

The *complement* of a graph $G = (V, E)$, written $\overline{G} = (V, \overline{E})$, is defined by $\overline{E} = \{xy : x, y \in V \text{ with } x \neq y \text{ and } xy \notin E\}$. For a positive integer k, the k^{th} *power* of a graph G, written $G^k = (V, E^k)$, is obtained from G by including an edge between every pair of vertices that are of distance at most k in G. (Hence G^1 is precisely G itself.) The *line graph* of a graph $G = (V, E)$ is the graph $H = (E, E')$ with $e_1e_2 \in E'$ if $e_1, e_2 \in E$ and e_1 and e_2 share a common endpoint.

A binary relation R on set V is a subset $R \subseteq V \times V$. The binary relation R is reflexive if $uu \in R$ for all $u \in V$, antisymmetric if $uv \in R$ and $vu \in R$ implies $u = v$ for all $u, v \in V$, and transitive if $uv \in R$ and $vw \in R$ implies $uw \in R$ for all $u, v, w \in V$. A *partial order* on a set V is a binary relation R that is reflexive, antisymmetric and transitive. We write $u \preceq v$ to mean $uv \in R$ and $u \prec v$ to mean $u \preceq v$ and $u \neq v$. We say u and v are incomparable, and write $u \parallel v$, if $u \not\preceq v$ and $v \not\preceq u$. A *total order* is a partial order with no pair of elements in V incomparable. A total order $\leq$ is a *linear extension* of a partial order $\preceq$ if $u \preceq v$ implies $u \leq v$ for all $u, v \in V$.

An *ordering* of V is a bijection $\sigma : \{1, \ldots, n\} \to V$. We call $\sigma^{-1}(v)$ the number assigned to v by σ. We use the notation $\sigma = (v_1, \ldots, v_n)$ to denote $\sigma(i) = v_i$ for $1 \leq i \leq n$, and use $\sigma_i = (v_1, \ldots, v_i)$ to denote a prefix of the ordering σ for i, $1 \leq i \leq n$. The ordering σ implicitly imposes a total order on V; we say $v_1 <_\sigma v_2 <_\sigma \cdots <_\sigma v_n$, and omit the subscript σ when the ordering is clear.

A *partition* P of a set V is a set of pairwise disjoint subsets of V whose union is V. The sets of P are called its *classes*. If P and Q are partitions of V, we say Q is a *refinement* of P if every class of Q is contained in a class of P.

For an undirected graph G, a set of vertices $U \subseteq V$ is a *clique* of G if $uv \in E$ for every $u, v \in U$ (i.e., $G[U]$ is complete). A vertex v is *simplicial* in G if $N_G(v)$ is a clique in G. A graph is said to be *chordal* (or *triangulated*) if every cycle of length at least 4 has at least one chord. An ordering $(v_1, \ldots, v_n)$ of V is a *perfect elimination ordering* (or peo) of G if v_i is simplicial in $G[v_i, \ldots, v_n]$ for $1 \leq i \leq n$. Fulkerson and Gross showed that a graph is chordal if and only if it has a perfect elimination ordering [FG65]. A graph is *bipartite* if its vertex set can be partitioned into two disjoint sets X, Y such that all edges are of the form xy for some $x \in X$ and $y \in Y$.

We will occasionally, in this thesis, make tangential reference to various other graph classes; the exact definitions of these classes are not germane to this thesis. Exact definitions of these classes can be found in Brandstädt, Le and Spinrad [BLS99].

2.2 Classical graph searches

To provide a common basis from which we can view graph searches, we examine what is meant by a "generic graph search." We describe a graph search in style similar to Tarjan [Tar72]. Given a possibly disconnected graph G whose vertices are all initially unvisited, we choose an arbitrary vertex of G to visit and choose one of its incident edges to follow. Traversing this edge leads to a vertex, which we now consider visited. We continue, at each step selecting an untraversed edge incident with a vertex we have already visited. Traversing this edge leads either to a new vertex or to one previously visited. Whenever we run out of untraversed edges incident with visited vertices and have not yet visited all vertices, we choose some new unvisited vertex to visit and begin again from there. Each edge is traversed exactly once, and we eventually visit all vertices in the graph. The first time a vertex v is encountered we say v is *first visited*. The order in which the vertices of G are first visited forms an ordering $\sigma = (v_1, \ldots, v_n)$ of V, where v_1 is the first vertex to be visited and v_n is the last.

We note that at each step of this algorithm, there may be many untraversed edges from which we may choose. How we choose amongst these edges to traverse causes the resultant vertex ordering to exhibit various properties. Traditionally, there have been two basic approaches to such a selection rule: depth-first search and breadth-first search.[1]

Depth-first search has been known since the 19th century, when Lucas reported on Trèmaux's technique for threading mazes [Luc82]. It has been used widely since the 1950s, though it was not until Tarjan's work [Tar72, HT73] that its power for solving various graph problems was recognized. Such problems included graph connectivity, planarity, topological ordering and strong connectivity of digraphs. One can think of DFS as walking through a graph, vertex by vertex, as long as possible before backtracking. As an example, a DFS may visit the vertices of the graph in Figure 2.1 in the order *cdefghba*.

[1]A formal presentation of these classical algorithms appears in Chapter 3.

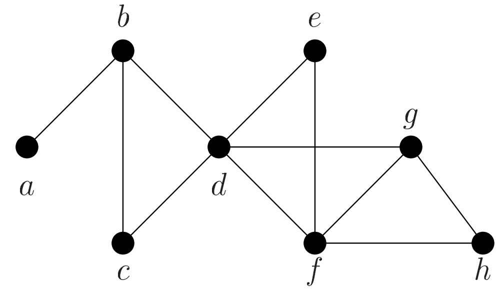


Figure 2.1: A graph, used as an example.

Breadth-first search, though having a somewhat shorter recorded history, has been exploited in several areas since the mid-twentieth century. BFS was originally presented by Moore [Moo59] in the context of finding paths through mazes, and discovered independently by Lee [Lee61] for routing wires on circuit boards. BFS has been applied to many graph problems, including shortest paths [Eve79], network flows [EK72], diameter estimation [CDK03], and recognition of various graph classes (such as unit interval graphs [CKN$^+$95]). As an example, a BFS may visit the vertices of the graph in Figure 2.1 in the order *cdbefgah*. BFS is commonly used to compute a distance layering of a graph, which is a partitioning of the vertices according to their distance from a source vertex s. A vertex ordering that yields a distance layering need not be a BFS ordering: the ordering *cdbaefgh* of the graph in Figure 2.1 yields a distance layering, but is not an ordering that can be found by BFS.

When we apply either of these searches to a connected graph, we construct a *search tree* (or, more generally, a *search forest* for possibly disconnected graphs) that describes the manner in which the algorithm explores the graph. When performing a search, we label as a tree edge each edge we used to discover a vertex, and we label the remaining edges as non-tree edges. The resulting search tree (or search forest) is a spanning tree (forest) of the graph. These search trees (forests) have several special properties that are exploited to solve a variety of problems. We show two examples of search trees in Figure 2.2.

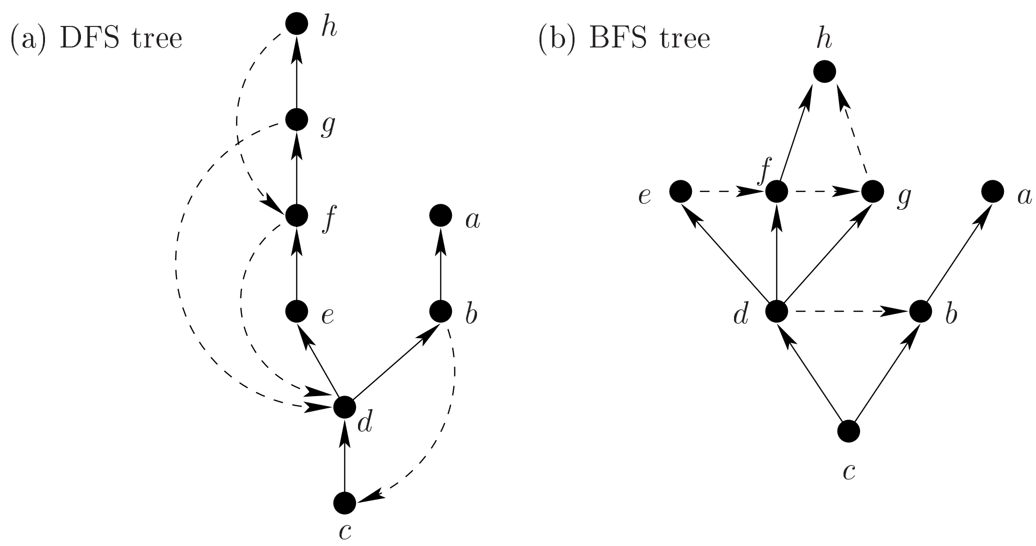


Figure 2.2: Examples of search trees of the graph G in Figure 2.1. (a) A DFS tree of G corresponding to ordering *cdefghba*. (b) A BFS tree of G corresponding to ordering *cdbefgah*. Dashed edges indicate non-tree edges of G, and the arrows indicate the direction of traversal.

2.3 Lexicographic Breadth-First Search

A common and widely used restriction of BFS is *lexicographic breadth-first search* (LexBFS), introduced by Rose, Tarjan and Lueker [RTL76] under the name Lex P. In the standard description of the LexBFS algorithm, vertices are assigned labels that are updated each time a neighbour is visited. Each label is a string of digits, and to decide which vertex to visit next, the algorithm picks an unvisited vertex whose label is largest according to the standard lexicographic order, hence the name of the algorithm.

The lexicographic order is also known as the standard dictionary order. Suppose $s_1 = (a_1, \ldots, a_k)$ and $s_2 = (b_1, \ldots, b_l)$ are two strings of digits selected over the alphabet $A = \{1, \ldots, n\}$ and $<_A$ is a total order on A. We say that s_1 is lexicographically smaller than s_2 (and write $s_1 < s_2$) if

1. there is an index $i \leq \min\{k, l\}$ such that $a_i <_A b_i$ and all previous digits are equal,

or

2. $k < l$ and the strings are equal up to index k.

As an example, the lexicographic ordering of strings of length at most 3 over the alphabet $\{0, 1\}$ is $(\varepsilon, 0, 00, 000, 001, 01, 010, 011, 1, 10, 100, 101, 11, 110, 111)$, assuming $0 < 1$. We denote by ε the empty string that consists of no digits.

Briefly, the LexBFS algorithm proceeds as follows. Initially, the label ε is assigned to all vertices. Iteratively, an unvisited vertex v with lexicographically maximum label is chosen and removed from the graph. If v is chosen as the i^{th} vertex, then each of its neighbours that are still unnumbered have their label updated by having the digit $(n-i)$ appended to their label. The digits in the label of any vertex are always in decreasing order, which ensures that all neighbours of the first vertex chosen are numbered before its nonneighbours, and, by extension, ensures all vertices are visited in a breadth-first order. We present the traditional LexBFS labelling algorithm as Algorithm 1.

Algorithm 1: Lexicographic Breadth-First Search (LexBFS)

input : a graph $G = (V, E)$

output: an ordering σ of V

assign the label ε to all vertices
for $i \leftarrow 1$ *to* n **do**
    pick an unnumbered vertex v with lexicographically largest label
    $\sigma(i) \leftarrow v$ //This assigns to v the number i
    foreach *unnumbered vertex w adjacent to v* **do**
        append $(n-i)$ to $label(w)$

LexBFS was originally introduced as a simplification of a recursive BFS algorithm for computing perfect elimination orderings of chordal graphs, where the ordering of the vertices within each level of the BFS tree is determined by a recursive application of

BFS. Using doubly-linked lists to store vertices with identical labels, and using another doubly-linked list to order the lists of vertices by their label, LexBFS can be implemented in linear time [RTL76, Gol80].

A simpler linear time implementation of LexBFS is presented by Habib, McConnell, Paul and Viennot [HMPV00] using partition refinement. In this approach, a partition of the unnumbered vertices is maintained and refined each time a vertex v is visited. The refinement step splits each class into two, placing the neighbours of v from the old class into the first new class and placing the the nonneighbours of v in the second new class. We describe this method as Algorithm 2, and illustrate an example in Figure 2.3.

Algorithm 2: Partition refinement version of LexBFS

input : a graph $G = (V, E)$

output: an ordering σ of V

create one partition class and place all vertices in it
for $i \leftarrow 1$ *to* n **do**
    choose and remove a vertex v from the first class
    $\sigma(i) \leftarrow v$ //This assigns to v the number i
    foreach *class* C **do**
        split class C into two classes, $C' = C \cap N(v)$ followed by $C'' = C \backslash C'$
        if C' is empty, delete it; likewise if C'' is empty, delete it

Though LexBFS was generally ignored for a decade after it was first discovered, it has recently experienced a surge in popularity, being applied to a wide variety of problems (see the survey of Corneil [Cor04a]). Fundamental to many of the proofs for these applications is a simple characterization of the vertex orderings that can be obtained by LexBFS. To introduce this characterization, we concentrate on the following situation. Suppose that during a search of a graph we have three vertices, a, b and c, where a was visited before b, which was visited before c, and edge ac is present in our graph, but edge ab is absent

1: $\boxed{\underline{c}\ a\ d\ b\ e\ f\ g\ h}$

2: $c\ \boxed{\underline{d}\ b}\ \boxed{a\ e\ f\ g\ h}$

3: $c\ d\ \boxed{\underline{b}}\ \boxed{e\ f\ g}\ \boxed{a\ h}$

4: $c\ d\ b\ \boxed{\underline{e}\ f\ g}\ \boxed{a}\ \boxed{h}$

5: $c\ d\ b\ e\ \boxed{\underline{f}}\ \boxed{g}\ \boxed{a}\ \boxed{h}$

Figure 2.3: The first few iterations of a LexBFS partitioning of the graph in Figure 2.1 resulting in ordering *cdbefgah*. The underlined vertex is visited at each iteration.

(see Figure 2.4). The question is "how could vertex b have been chosen before vertex c by our search?" With the limited information we have about this situation, it seems that c should have been chosen before b.

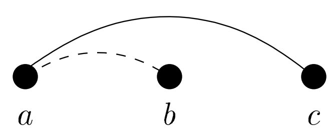
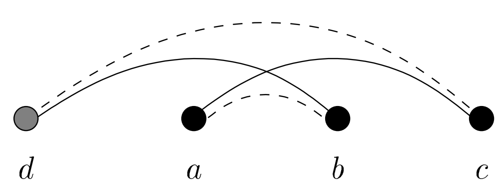


Figure 2.4: A curious triple of vertices in a search ordering.

For LexBFS, we can infer details about the structure of the graph. We know that there must exist another vertex d in the graph that was chosen earlier than a such that db is an edge, but dc is not an edge of G [Gol80]. We illustrate this below and formally state this as property (LB).



(LB): Given an ordering σ of V, if $a < b < c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex $d < a$ such that $db \in E$ and $dc \notin E$.

Though this has long been known to be a property of LexBFS orderings [RTL76], the reverse direction showing (LB) characterizes LexBFS orderings has only recently been shown by Brandstädt, Dragan and Nicolai [BDN97] (and, in fact, independently and subsequently by several different groups).

Theorem 2.3.1 ([BDN97]). *For an arbitrary graph G, a vertex ordering σ is a LexBFS ordering of G if and only if σ has property (LB).*

Experience with LexBFS has shown that this characterizing property is instrumental in the applicability of the search to various problems. For example, the LexBFS characterization is fundamental in proving the following result which inspired the algorithm.

Theorem 2.3.2 ([RTL76]). *If G is a chordal graph and σ is a LexBFS ordering of G, then the reverse of σ is a peo of G.*

Many of the recent LexBFS-based algorithms use multiple sweeps of LexBFS to reveal structure of the given graph (such as [Cor04b, CHK99, COS98, BCHP03, Bre05]). Each subsequent search uses information from the previous sweep(s) to break ties over which vertex to visit next. The related proofs of correctness rely heavily on the LexBFS characterization. The characterization can also be used to show that there is some LexBFS ordering that certifies a graph as being a cocomparability graph, even though we do not know how to find such an ordering efficiently [COS99].

The usefulness of this simple characterization result for LexBFS provides inspiration for asking similar questions for the other basic types of graph search, which we address in Chapter 3. The only other search that was previously known to have a similar vertex ordering characterization is maximum cardinality search, which we examine in the next section.

2.4 Maximum Cardinality Search

Maximum cardinality search (MCS) was first presented by Tarjan in an unpublished work [Tar76] as an alternate method for finding a perfect elimination ordering of chordal graphs. Yannakakis found that both LexBFS and MCS could be applied to problems arising in the study of relational databases. Tarjan and Yannakakis presented MCS and its applications to recognition of chordal graphs, testing acyclicity of hypergraphs, and how to selectively reduce an acyclic hypergraph to more efficiently compute database queries [TY84]. Inspired by a question of Edmonds, an addendum [TY85] shows how a chordless cycle can be found in a non-chordal graph.

The MCS algorithm is similar to LexBFS but uses a single integer as a vertex label instead of a string. Simply stated, MCS iteratively chooses an unnumbered vertex which has the maximum number of already numbered neighbours, hence the name of the search. We can track how many neighbours have been visited for each vertex by attaching an integer label to each vertex. We present MCS as Algorithm 3, and mention that, like LexBFS, MCS has a linear time implementation.

Algorithm 3: Maximum Cardinality Search (MCS)

input : a graph $G = (V, E)$

output: an ordering σ of V

assign the label 0 to all vertices
for $i \leftarrow 1$ *to* n **do**
    pick an unnumbered vertex v with maximum label
    $\sigma(i) \leftarrow v$ //This assigns to v the number i
    foreach *unnumbered vertex w adjacent to v* **do**
        increment $label(w)$ by 1

As an example, consider the graph in Figure 2.1. One possible ordering that MCS can compute is *cdbaefgh*. We note that LexBFS could not have found this ordering. In general, there are MCS orderings that are not LexBFS orderings (or even BFS orderings), and there are LexBFS orderings that are not MCS orderings. However, like LexBFS, every MCS ordering of a chordal graph yields a peo.

Theorem 2.4.1 ([TY84]). *If G is a chordal graph and σ is a MCS ordering of G, then the reverse of σ is a peo of G.*

Its similarity to LexBFS extends through to the existence of a vertex ordering characterization. We consider the question raised for LexBFS and illustrated in Figure 2.4: "if $a < b < c$ but $ac \in E$ and $ab \notin E$, how could MCS choose vertex b before vertex c?" Tarjan and Yannakakis [TY84] show that there must be some vertex d chosen before b such that $db \in E$ and $dc \notin E$.

It turns out that this statement can be strengthened. We consider the following question: "if $A = \{a_i\}_{i=1}^{k}$ is a set of k distinct vertices with $a_i < b < c$ and such that $a_ic \in E$ and $a_ib \notin E$ for $1 \leq i \leq k$, how could MCS choose b before c?" We must have a set of distinct vertices D with the same cardinality as A such that for $d \in D$, d was chosen before b and $db \in E$, but $dc \notin E$. We express this property formally below.

(MC): Given an ordering σ of V, if there exists vertices $a_1 < \cdots < a_k < b < c$ and, for $1 \leq i \leq k$, we have $a_ic \in E$ and $a_ib \notin E$, then there exists a set of vertices $d_1 < \cdots < d_k < b$ with $d_ib \in E$ and $d_ic \notin E$ for $1 \leq i \leq k$.

Brandstädt, Dragan and Nicolai [BDN97] found that this property characterizes the vertex orderings that MCS may find. We state this fact as the following theorem.

Theorem 2.4.2 ([BDN97]). *For an arbitrary graph G, an ordering σ of V is an MCS ordering if and only if σ has property (MC).*

MCS computes a perfect elimination ordering of a chordal graph. If some powers of a graph are also chordal, one may be interested in computing a common peo for these powers. Brandstädt, Chepoi and Dragan [BCD96] generalizes MCS to find a common peo for all chordal powers of an arbitrary graph. The resulting algorithm, *lexicographic maximum cardinality search* (LMCS), assigns labels to vertices which correspond to a list of cardinalities of the visited neighbourhoods in each of the chordal powers of the graph. Ties in MCS are broken by comparing these vertex labels and choosing a vertex with the lexicographically largest label, somewhat like LexBFS. Assuming we know a set $\{G^{i_1}, \ldots, G^{i_p}\}$ of chordal powers of G, LMCS finds a common peo for these chordal powers of G [BCD96]. In particular, if both G and G^2 are chordal, then LMCS finds a common peo for all powers of G (since then all powers of G are chordal by a result of Duchet [Duc84]). However, it is not clear how to implement LMCS in linear time; [BCD96] claims only a running time of $O(nm)$. Brandstädt, Dragan, and Nicolai [BDN97] shows that MCS itself cannot find a common peo of the chordal powers of a graph, but LexBFS can. In fact, every LexBFS ordering σ of a chordal graph is a common peo of all odd powers of the graph [BDN97], although σ may not necessarily be a LexBFS ordering of the powers [CK].

It is well-known that not all perfect elimination orderings can be generated by either LexBFS or MCS. It may be useful to have a scheme that is capable of generating all peos of a chordal graph. Suppose S_i is the set of vertices already chosen by some search. Both LexBFS and MCS must select a vertex whose label is maximum over all unnumbered vertices, $V - S_i$. Shier [Shi84] observes that to find a peo, the next vertex chosen merely needs to have label maximal within its connected component in $G[V - S_i]$. With this observation, Shier [Shi84] devises two graph search schemes, *maximal element in component* (MEC), in which we choose a vertex with maximal (with respect to set inclusion) neighbourhood within a component of S_i, and *maximum cardinality neighbourhood in component* (MCC), in which we choose a vertex with a neighbourhood of largest cardinality within a component of S_i. Both LexBFS and MCS are special cases of these schemes. These schemes can be shown to generate exactly the peos of a graph.

Theorem 2.4.3 ([Shi84]). *For any graph G and ordering σ of V, the following statements are equivalent:*

1. *σ is the reverse of a peo of G.*

2. *σ is an MCC ordering of G.*

3. *σ is an MEC ordering of G.*

Though both MCC and MEC can be implemented in linear time (simply by using an efficient MCS implementation), it is not clear how to efficiently implement these algorithms to possibly find every peo. However, all peos of a chordal graph can be listed in constant amortized time using a Gray code algorithm [CIRS03].

According to Shier [Shi84], in unpublished work, Hoffman and Sakarovich generate all peos by imposing a particular partial ordering on the remaining candidate vertices and selecting a maximal element in this partial order. Specifically, if y is on some chordless path from x to a vertex already selected (i.e., in G_i), y is considered larger than x in the partial order (and thus x cannot be chosen). Shier [Shi84] shows this scheme is equivalent to MEC.

2.5 Notation for graph search orderings

The following notation will be convenient when discussing the execution of our graph search algorithms. Many of our algorithms conceptually assign labels to vertices, which are updated each time a new vertex is visited. When discussing an execution of some search algorithm that computed the vertex ordering σ, we let $label_i(v)$ denote the label that was assigned to vertex v at the moment that $\sigma(i)$, the i^{th} vertex numbered, is assigned. One may interpret $label_i(v)$ as the label assigned to v at the beginning of iteration i of the search. If $w = \sigma(i)$, we write $label_w(v)$ as shorthand for $label_{\sigma^{-1}(w)}(v)$.

We draw particular attention to the fact that the search algorithms in this thesis generate vertex orderings numbering vertices from 1 to n, and we write these orderings from left to right in this order. We say a vertex v is *left* or *right* of a vertex w in the search ordering σ if $\sigma^{-1}(v) < \sigma^{-1}(w)$ or $\sigma^{-1}(v) > \sigma^{-1}(w)$, respectively. In many applications it may be useful to reverse the ordering generated by search, and we caution the reader that some of the existing LexBFS literature considers this reversal as part of the search algorithm.

Chapter 3

Characterizing Graph Search

In this chapter we present a unified view of graph searching from the perspective of vertex orderings.[1] Inspired by the usefulness of the LexBFS vertex ordering characterization, and observing how critical such a property is to proving correctness of LexBFS-based algorithms, we set out to find related properties for other types of graph search. We focus our discussion around the key question, "if $a < b < c$ and ac is an edge but ab is not an edge, how could a graph search have chosen vertex b before vertex c?" We find that this single question leads to simple four-vertex characterizations for each of the basic graph search paradigms.

Along the way we discover some holes in our existing knowledge of the graph search landscape. We use the insight gained by our new vertex ordering characterizations to identify two new graph search algorithms, *lexicographic depth-first search* and *maximal neighbourhood search*. We find that these new varieties of graph search are well defined and quite useful, as later chapters will demonstrate. At the conclusion of this chapter, we construct a hierarchy of graph searching, illustrating the fundamental relationships amongst the basic search paradigms as exposed by our four-vertex characterizations.

[1] Many of the results of this chapter are reported in [CK].

3.1 Generic Search

We begin by formalizing generic search for connected graphs, similar to Tarjan's description in [Tar72], and present it as Algorithm 4. We use a generic "set" data structure S to store the candidate vertices we may next visit during our search. We note that we impose no selection criteria on which member of S we choose in line 3 beyond that it must be in S.

Algorithm 4: Generic Search on connected graphs

input : a connected graph $G = (V, E)$ and start vertex $s \in V$

output: an ordering σ of V

1 $S \leftarrow \{s\}$
2 **for** $i \leftarrow 1$ *to* n **do**
3   pick and remove an unnumbered vertex v from S
4     $\sigma(i) \leftarrow v$ //This assigns to v the number i
5   **foreach** *unnumbered vertex* w *adjacent to* v **do**
6   add w to S

We call any ordering of V that can be generated by Algorithm 4 a *generic graph search ordering*, or just a *search ordering*, of the graph G. As an example, we notice that (1,2,4,3,5,6) is a valid search ordering of the 3-sun in Figure 3.1. The reader will notice that this ordering is neither a BFS nor a DFS. We may opt to use a more restrictive data structure to obtain other properties for our search, such as replacing set S with a queue to obtain BFS. Note also that, as stated, a given vertex may be added to S multiple times as each of its neighbours is numbered. How S deals with this can change the nature of the search. For example, using a variant of a stack where the add operation moves a vertex to the top of the stack if that vertex was added previously will yield DFS.

We now proceed to answer the fundamental question raised above about our search

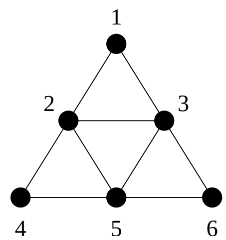
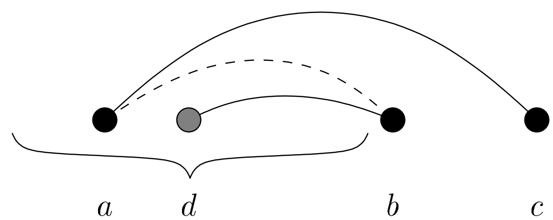


Figure 3.1: The 3-sun, used as an example for our graph searches.

orderings: how could b have been chosen before c? We give a simple answer, which turns out to characterize search orderings.



(S): Given an ordering σ of V, if $a < b < c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex $d < b$ such that $db \in E$.

We note that (S) makes no statement to the relative order of a and d in σ. It simply says that there must be a vertex adjacent to b visited sometime earlier than b in σ. We will now show that (S) characterizes search orderings. That is to say that any ordering that the generic search algorithm can produce has this property, and that any vertex ordering with this property can be produced by the generic search algorithm when we break ties suitably.

Lemma 3.1.1. *For a connected graph G, an ordering σ of V is a search ordering of G if and only if σ has property (S).*

Proof. For the forward direction, suppose σ is an ordering generated by Algorithm 4. Let $a < b < c$ be a triple of vertices in σ with $ac \in E$ and $ab \notin E$. At the point when b is chosen, both b and c must be in S. Since b is in S, some neighbour of b must have already been chosen, call this neighbour d. Thus $d < b$ and $db \in E$. Since the triple a, b, c was arbitrary, property (S) holds on σ.

For the converse, suppose $\sigma = (v_1, \ldots, v_n)$ is an ordering respecting property (S). Suppose for contradiction that $(v_1, \ldots, v_{i-1})$ can be obtained by Algorithm 4, but $(v_1, \ldots, v_i)$ cannot, for some i, $1 < i \leq n$. That is, v_i is the first vertex in σ that cannot be chosen next by the algorithm. Consider the next iteration of the algorithm after choosing v_{i-1}. Since v_i cannot be chosen next, $v_i \notin S$. Let u be a vertex that the algorithm may choose next. Then u must be in S. Let w be the neighbour of u which caused u to be added to S. Since $v_i \notin S$, we know that $wv_i \notin E$. By applying property (S) to the vertex triple (w, v_i, u), we know there exists a vertex $d < v_i$ with $dv_i \in E$. Hence $v_i \in S$, contradicting our assumption that v_i could not be chosen by Algorithm 4. □

It is straightforward to extend this result to arbitrary graphs. We extend Algorithm 4 by adding a wrapper which, once a connected component is exhausted, allows a new component to be started. We present this extension as Algorithm 5. Using this algorithm, the following theorem is straightforward.

Algorithm 5: Generic Search on arbitrary graphs

input : a graph $G = (V, E)$

output: an ordering σ of V

$i \leftarrow 0$
while $i < n$ **do**
  choose an arbitrary unnumbered vertex s
  let C be the connected component of G containing s
  call Algorithm 4 on pair (C, s), yielding output τ
  for $j \leftarrow 1$ *to* $|C|$ **do**
    $\sigma(i + j) \leftarrow \tau(i)$
  $i \leftarrow i + |C|$

Theorem 3.1.2. *For an arbitrary graph G, an ordering σ of V is a search ordering of G if and only if σ has property (S).*

Proof. If σ is an ordering generated by Algorithm 5, suppose $a < b < c$ is a triple satisfying the antecedent of (S). Since $ac \in E$, a and c are in the same component, and, since Algorithm 5 does not start on a new component until the current component has been completely numbered, b must be contained within the same component. Thus the triple lies entirely within a single component, and (S) holds on σ by Lemma 3.1.1.

On the other hand, if σ respects (S), then the vertices of each component of G must appear consecutively in σ (if they are not consecutive, there must be a triple $a < b < c$ with a and c in the same component and b in another that violates (S)). Thus, by Lemma 3.1.1, σ is the concatenation of search orderings of the components of G, which can be computed by Algorithm 5 with suitable choices for s. □

3.2 Breadth-First Search

We now proceed to the familiar breadth-first search algorithm. BFS is a restriction of generic search in that we use a queue to determine which vertex we may choose next at our selection step. BFS does not, however, determine which order to push the neighbours of a numbered vertex onto the queue. Algorithm 6 should be familiar to most readers.

To illustrate BFS and its difference from LexBFS, we note that (1,2,3,4,5,6) is a BFS ordering of the 3-sun in Figure 3.1, but is not a LexBFS ordering. For LexBFS, once 1,2,3 are picked, 5 must be chosen over 4 since $label(5) = \{4, 3\}$ while $label(4) = \{4\}$. The ordering (1,2,3,5,4,6) is a valid LexBFS ordering of the 3-sun, which is also a valid BFS ordering.

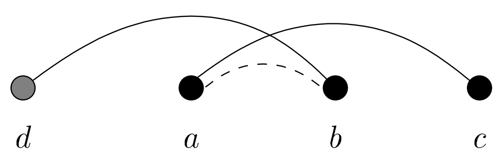
We now key in on our central question: how could b have been chosen before c in a BFS? The answer is a simple restriction to the answer we found for generic search.

Algorithm 6: Breadth-First Search (BFS) on connected graphs

input : a connected graph $G = (V, E)$ and start vertex $s \in V$

output: an ordering σ of V

```
initialize queue to {s}
for i ← 1 to n do
    dequeue v from beginning of queue
    σ(i) ← v      //This assigns to v the number i
    foreach unnumbered vertex w adjacent to v do
        if w in not already in queue then
            enqueue w to end of queue
```



(B): Given an ordering σ of V, if $a < b < c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex $d < a$ such that $db \in E$.

This property is merely property (S) with the added restriction that $d < a$. Interestingly, we find that this is sufficient to characterize all BFS orderings.

Theorem 3.2.1. *For an arbitrary graph G, an ordering σ of V is a BFS ordering of G if and only if σ has property (B).*

Proof. We prove the statement for connected graphs; the extension to arbitrary graphs is straightforward and similar to the generic search case.

For the forward direction, suppose σ is a BFS ordering. Let $a < b < c$ be a triple of vertices in σ with $ac \in E$ and $ab \notin E$. If b was dequeued (and thus enqueued) before c, it must have a neighbour d that was numbered as least as early as a. Since a does not have b as a neighbour, d must be earlier than a.

For the reverse direction, suppose for contradiction that $\sigma = (v_1, \ldots, v_n)$ is an ordering satisfying property (B) such that $(v_1, \ldots, v_{i-1})$ can be produced by BFS, but $(v_1, \ldots, v_i)$ cannot, for some i, $1 < i \leq n$. That is, v_i cannot be chosen next by any valid execution of BFS on G. Let u be a vertex that may be chosen next by some execution of BFS. Since u is dequeued, it must have been enqueued when some neighbour v_j with $j < i$ was previously numbered. Then $v_j v_i \notin E$, otherwise some valid execution of BFS can enqueue v_i before u. Applying (B) to the triple $v_j < v_i < u$, there must exist a vertex $d < v_j$ with $dv_i \in E$. Thus v_i is enqueued before, and dequeued before, u, contradicting our assumption above. □

To illustrate an application of this property, we now prove the correctness of a generalization of a well-known technique for extracting a shortest s, w-path (for arbitrary w) given an ordering with property (B) starting with s. For notation, when given a vertex ordering σ of V and a subset $T \subseteq V$, we say $\sigma|_T$ is the restriction of σ to T.

Corollary 3.2.2. *Let σ be an arbitrary ordering of the vertices of a graph G, and let $H = G[T]$ be a connected induced subgraph of G such that $\sigma|_T$ satisfies (B). Suppose s is the first vertex in $\sigma|_T$, and let $w = w_0$ be an arbitrary vertex in T. For $i > 0$, define w_i to be the leftmost (in $\sigma|_T$) neighbour of w_{i-1}, until s is reached. Then $(s = w_k, \ldots, w_1, w_0 = w)$ is a shortest s, w-path in H.*

Proof. Suppose, for contradiction, that $(s = w_k, \ldots, w_1, w_0 = w)$ is not a shortest s, w-path in H. Let $(s = x_{k'}, \ldots, x_1, x_0 = w)$ be a shorter s, w-path, with $k' < k$. Note that $x_{k'} < w_{k'}$. Choose the smallest $i > 0$ such that $x_i < w_i$. Then $w_{i-1} \leq x_{i-1}$. Since $x_i x_{i-1} \in E$ and w_i is the leftmost neighbour of w_{i-1}, we know that $x_i w_{i-1} \notin E$ (and thus $w_{i-1} \neq x_{i-1}$). Consider the triple $x_i < w_{i-1} < x_{i-1}$. Since $x_i x_{i-1} \in E$ but $x_i w_{i-1} \notin E$, property (B) says there is a vertex $d < x_i$ with $dw_{i-1} \in E$. This contradicts w_i being the leftmost neighbour of w_{i-1}. Hence our original path was, in fact, shortest. □

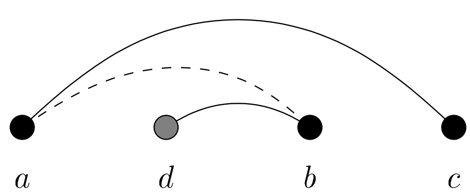
Remarkably, the requirement of dc not being an edge is the only difference between lexicographic and traditional breadth-first search. We obtain the immediate corollary, without any mention of the respective algorithms, that a LexBFS ordering is a BFS ordering, since (B) subsumes (LB).

3.3 Depth-First Search

We now turn our attention to depth-first search. DFS is very different from BFS in that it progresses forward through the graph as much as possible, backtracking only when necessary, whereas BFS first systematically explores close vertices before venturing out to the far reaches of the graph. One also notes that DFS produces two vertex orderings: a discovery ordering, which is the order vertices are first visited, and a finishing ordering, indicating the order in which each vertex's neighbourhood is fully explored with all incident edges traversed and causing a backtracking step. Here we concentrate solely on the discovery orderings of depth-first searches.

We start by presenting a DFS algorithm exploiting a stack data structure to store candidate vertices to visit next. We present DFS as Algorithm 7 in a style similar to our generic search algorithm in Section 3.1. We note that vertices may be pushed on the stack multiple times, so we must repeatedly pop elements at line 4 until we find an unnumbered vertex.

We again return to our key question: given a DFS ordering with $a < b < c$ with ac an edge but ab not an edge, how could DFS have chosen b over c? The answer is property (D), which we will show characterizes DFS orderings.



(D): Given an ordering σ of V, if $a < b < c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex d with $a < d < b$ such that $db \in E$.

Algorithm 7: Depth-First Search (DFS) on connected graphs

```
input  : a connected graph G = (V, E) and start vertex s ∈ V
output: an ordering σ of V

initialize stack to {s}
for i ← 1 to n do
    repeat
        pop v from top of stack
    until v is unnumbered
    σ(i) ← v      //This assigns to v the number i
    foreach unnumbered vertex w adjacent to v do
        push w on top of stack
```

Notice that this property restricts (S) only in that d must also appear to the right of a. We contrast this with BFS property (B) where d was to the left of a. In this sense, DFS is the opposite search paradigm to BFS.

Theorem 3.3.1. *For an arbitrary graph G, an ordering σ of V is a DFS ordering of G if and only if σ has property (D).*

Proof. We prove the statement for connected graphs; the extension to arbitrary graphs is straightforward and similar to the generic search case.

For the forward direction, suppose σ is an ordering produced by DFS. Suppose we have a triple of vertices $a < b < c$ in σ with $ac \in E$ and $ab \notin E$. Since b is numbered (and $b \neq s$, the initial vertex), b must have been pushed on the *stack* when one of its neighbours was previously numbered. Let d be the rightmost neighbour of b appearing before b in σ. Then $d < b$ and $d \neq a$. Suppose $d < a$. Since a was numbered after d, c is pushed on the stack above b. As there are no other neighbours of b before b is numbered, c is higher on the stack than b and must be numbered before b, a contradiction. Thus we must have $a < d$, and property (D) holds.

Now for the reverse direction, suppose $\sigma = (v_1, \ldots, v_n)$ is an ordering respecting property (D). Suppose $\sigma_{i-1} = (v_1, \ldots, v_{i-1})$ can be obtained by DFS, but $(v_1, \ldots, v_i)$ cannot. That is, v_i is the first vertex in σ that cannot be chosen next by DFS. Let u be a vertex that DFS may choose next. Then $v_i < u$. Let v_j be the rightmost vertex in σ_{i-1} such that $v_j u \in E$ (that is, v_j is the parent vertex of u in the DFS-tree). Then $v_j v_i \notin E$ (otherwise v_i could be chosen next) and, more generally, $v_k v_i \notin E$ for all k, $j \leq k < i$. By applying property (D) to the triple (v_j, v_i, u), we know there exists a vertex d between v_j and v_i such that $dv_i \in E$, a contradiction. □

We can apply (D) repeatedly to obtain the following simple corollary.

Corollary 3.3.2. *Given a graph G and an arbitrary DFS ordering σ of V, if $a < b < c$ and $ac \in E$, then there exists a path P from a to b whose vertices increase in σ. That is, there exists an a,b-path $P = (a = d_k, \ldots, d_1, d_0 = b)$ in G with $\sigma^{-1}(d_i) < \sigma^{-1}(d_{i-1})$ for each $d_i \in P \backslash \{b\}$.*

Proof. Let $d_0 = b$. Once we assign d_i, $i \geq 0$, either $ad_i \in E$, and we complete the path by assigning $d_{i+1} = a$, or (D) applies to $a < d_i < c$, and we find a vertex d_{i+1} with $a < d_{i+1} < d_i$ and $d_{i+1}d_i \in E$. Since G is finite, this process must terminate, and such an a,b-path exists. □

Most previous results regarding DFS rely on properties of the DFS-tree. The *DFS-tree* T is defined on the vertices of G and includes exactly those edges traversed to number a new vertex. Formally, $v_i v_j \in E(T)$ with $i < j$ if and only if $i = \max\{k \mid v_k v_j \in E(G)$ and $k < j\}$. We say u is the *parent* of v in T if $uv \in E(T)$ and $\sigma^{-1}(u) < \sigma^{-1}(v)$. The *nontree edges* (commonly called *back edges*) are the remaining edges in G. In T, u is an *ancestor* of v if there is a path from u to v all of whose edges go from parent to child. We say two vertices are *related* if one is an ancestor of the other.

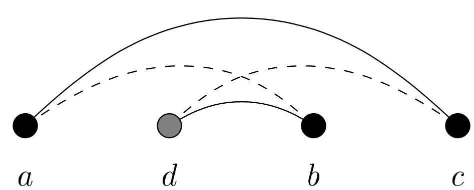
Many data structure textbooks (for example, [CLRS01]) mention the following fundamental property of DFS.

(D1): In a DFS-tree of an undirected graph, the two endpoints of each nontree edge are related.

This property is a consequence of Corollary 3.3.2: for a back edge ac with $a < c$, let b be the parent of c, and when computing the path, always take the rightmost neighbour to obtain a path in the DFS-tree. In fact, Corollary 3.3.2 is stronger than (D1) in that, if uv is a back edge in the DFS-tree with $u < v$, we know all vertices between u and v (inclusive) in σ are related to u.

3.4 Lexicographic Depth-First Search

Looking at the relationship between BFS and LexBFS, one naturally asks whether there is a "lexicographic analogue" for DFS. We have noted that LexBFS enforces the requirement of "$dc \notin E$" on top of BFS. It is natural to question whether a similar requirement makes sense for DFS. We state this restriction of property (D) below.



(LD): Given an ordering σ of V, if $a < b < c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex d with $a < d < b$ such that $db \in E$ and $dc \notin E$.

We turn to the task of devising an algorithm that realizes this property. Considering our 3-sun example in Figure 3.1, once vertices 1 and 2 are chosen, the only way to avoid violating (LD), and hence the only valid ordering, is (1,2,3,5,6,4). Clearly this is a DFS ordering, but the DFS ordering (1,2,4,5,6,3) violates (LD). The LexBFS ordering (1,2,3,5,4,6) is also disallowed.

These examples suggest that our algorithm should choose vertices that are adjacent to as many vertices we have more recently numbered as possible. As it is to be related to LexBFS, our algorithm should also be in some sense similar to our LexBFS algorithm. (We discuss the property that actually relates LexBFS and LexDFS in Section 3.5). We present Algorithm 8 as our LexDFS algorithm.

Algorithm 8: Lexicographic Depth-First Search (LexDFS)

input : a graph $G = (V, E)$

output: an ordering σ of V

1 assign the label ε to all vertices
2 **for** $i \leftarrow 1$ *to* n **do**
3   pick an unnumbered vertex v with lexicographically largest label
4     $\sigma(i) \leftarrow v$ //This assigns to v the number i
5   **foreach** *unnumbered vertex* w *adjacent to* v **do**
6   prepend i to $label(w)$

Examining Algorithm 8, our answer to the "what is LexDFS?" question becomes quite convincing. There is just one difference between this algorithm and the LexBFS algorithm. For LexDFS, we prepend the new digit i to $label(i)$ at line 6, whereas, for LexBFS, we append the new label digit $n - i$. This elementary change converts the algorithm from a breath-first search into a depth-first search. We now show that our algorithm correctly defines LexDFS.

Theorem 3.4.1. *For an arbitrary graph G, an ordering σ of V is a LexDFS ordering of G if and only if σ has property (LD).*

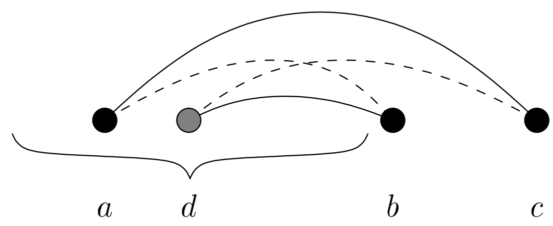
Proof. For the forward direction, suppose σ is an ordering computed by LexDFS. Let $a < b < c$ be a triple of vertices in σ with $ac \in E$ and $ab \notin E$. Since b was numbered before c, $label_b(b) \geq label_b(c)$. We note that the digits in each label appear in decreasing order. Since $\sigma^{-1}(a)$ appears in $label_b(c)$ and does not appear in $label_b(b)$, $label_b(b)$ must include some digit $j > \sigma^{-1}(a)$ which does not appear in $label_b(c)$. Let $d = \sigma(j)$. Then $d < b$, $db \in E$ and $dc \notin E$. Since $j > \sigma^{-1}(a)$, $a < d$, thus property (LD) is satisfied.

Now for the reverse direction, suppose $\sigma = (v_1, \ldots, v_n)$ is an ordering respecting property (LD). Suppose $(v_1, \ldots, v_{i-1})$ can be obtained by LexDFS, but $(v_1, \ldots, v_i)$ cannot. That is, v_i is the first vertex in σ that cannot be chosen next by LexDFS. Let u be a vertex that LexDFS may choose next. Since u must be chosen over v_i, $label(u)$ must be lexicographically larger than $label(v_i)$: let j be the largest digit in $label(u)$ that is not in $label(v_i)$ (i.e., v_j is the rightmost vertex in σ_{i-1} with $v_j u \in E$ and $v_j v_i \notin E$). By applying property (LD) to the vertex triple (v_j, v_i, u), we know there exists a vertex d between v_j and v_i such that $dv_i \in E$ and $du \notin E$. Then $\sigma^{-1}(d)$ is in $label(v_i)$ but not in $label(u)$. Since $v_j < d$, $label(v_i)$ is in fact larger than $label(u)$, a contradiction. $\square$

As a new search algorithm, we should concern ourselves with its implementation. Algorithm 8 does not immediately have a linear-time implementation using the ideas used for LexBFS. It seems that the partition refinement idea exploited for LexBFS cannot efficiently be applied to LexDFS, as its depth-first properties can force any remaining vertex to be the next to be numbered. We investigate this question in Chapter 5.

3.5 Maximal Neighbourhood Search

We now consider the question of "what fact relates LexBFS and LexDFS?" What useful property does this nonedge dc give us? Equivalently, we ask what type of search we get when adding the restriction $dc \notin E$ to the generic search property (S). We formalize our discussion around this property.



(M): Given an ordering σ of V, if $a < b < c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex d with $d < b$ and $db \in E$ and $dc \notin E$.

Surprisingly, this property has already been used in the literature. Tarjan and Yannakakis [TY84] mentioned that both maximum cardinality search and LexBFS respect property (M), and showed that any ordering with property (M) is the reverse of a perfect elimination ordering of a chordal graph. But the paper does not consider whether there is a natural search algorithm that is characterized by this property. We answer that question here by introducing *maximal neighbourhood search* (MNS) as Algorithm 9. The labels assigned to vertices are sets, whose elements correspond to its numbered neighbours.

The idea behind MNS is simple: at each step, pick a vertex whose neighbourhood in the part of the graph already explored is maximal (with respect to set inclusion). This is a straightforward generalization of the LexBFS, LexDFS and MCS algorithms. We obtain each of these searches from Algorithm 9 by further specifying how to order incomparable neighbourhoods. For LexBFS, we choose a vertex adjacent to as many *earliest* chosen vertices as possible. For LexDFS, we choose a vertex adjacent to as many *most recently* chosen vertices as possible. For MCS, we choose a vertex whose neighbourhood has *maximum* cardinality.

We now show that the vertex orderings produced by Algorithm 9 are characterized by property (M).

Theorem 3.5.1. *For an arbitrary graph G, an ordering σ of V is a MNS ordering if and only if σ has property (M).*

Algorithm 9: Maximal Neighbourhood Search (MNS)

```
input  : a graph G = (V, E)
output: an ordering σ of V

assign the label ∅ to all vertices
for i ← 1 to n do
    pick an unnumbered vertex v with maximal label (under set inclusion)
    σ(i) ← v      //This assigns to v the number i
    foreach unnumbered vertex w adjacent to v do
        add i to label(w)
```

Proof. For notation, let $G_i = G[\sigma_i] = G[\{\sigma(1), \ldots, \sigma(i)\}]$ for $1 \leq i \leq n$. For the forward direction, suppose σ is an MNS ordering. Suppose $a < b < c$ is a triple in σ that violates (M). That is, $ac \in E$ and $ab \notin E$, yet there is no vertex $d < b$ with $db \in E$ and $dc \notin E$. Consider the point in execution when $b = v_i$ is chosen. Then $N_{G_{i-1}}(b) \subsetneq N_{G_{i-1}}(c)$, so b could not have been chosen next by MNS, a contradiction. Hence (M) must hold for σ.

For the converse, suppose $\sigma = (v_1, \ldots, v_n)$ is an ordering respecting property (M). Suppose $(v_1, \ldots, v_{i-1})$ can be obtained by MNS, but $(v_1, \ldots, v_i)$ cannot; that is, v_i is the first vertex in σ that cannot be chosen next by MNS. Then there must be another vertex u with $N_{G_{i-1}}(u) \supsetneq N_{G_{i-1}}(v_i)$. Let $x \in N_{G_{i-1}}(u) - N_{G_{i-1}}(v_i)$. We apply (M) to the triple $x < v_i < u$, giving a vertex $d < v_i$ with $dv_i \in E$ and $du \notin E$. This contradicts $N_{G_{i-1}}(u) \supsetneq N_{G_{i-1}}(v_i)$, hence our assumption that v_i could not be chosen by MNS was incorrect. □

3.6 Summary

We have examined six different search algorithms from the perspective of our key question: if a search visits a vertex a before b before c, and there is an edge ac but no

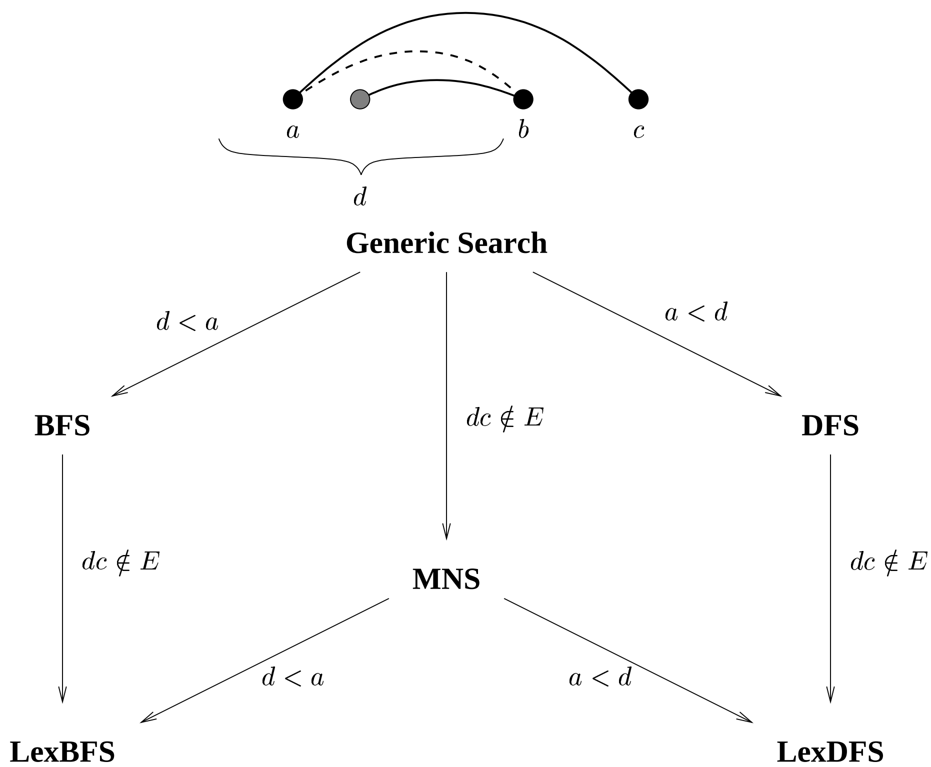


Figure 3.2: Summary of 4-vertex search characterizations. For example, vertex ordering σ is a LexDFS ordering iff σ satisfies the condition for generic search together with $dc \notin E$ and $a < d$.

edge ab, how could the search have visited b before c? In this examination, we have discovered characterizing properties for the classic breadth-first and depth-first searches, and introduced a new class of maximal neighbourhood searches. We summarize the characterizations of the six search paradigms and the hierarchy they form in Figure 3.2.

Though we have concentrated on simple graphs in this chapter, these ideas can easily be extended to more general environments. Our characterizations make no mention of multiple edges, but are equally applicable to searches on multigraphs. A search ordering implicitly imposes an orientation on the edges of a simple graph: it is straightforward to apply the same ideas to directed graphs to obtain similar characterizations. We can also extend such characterizations beyond graphs: these ideas also hold for searches on hypergraphs. We consider these extensions in Chapter 8.

Such characterizing properties can be a very powerful and useful tool for studying graphs. Beyond giving us a better understanding of how a search reveals the structure of a graph, it allows us to employ (and analyze) multiple sweeps of a search to gather additional information about a graph. Multi-sweep LexBFS has recently been applied to a variety of problems [Cor04a], resulting in very simple, and in many cases intuitive, algorithms. It is our hope that these characterizations and the insight they bring will lead to the development of new algorithms, quite possibly multi-sweep algorithms involving other searches, or even combinations of different searches.

Chapter 4

An Algorithm for Graph Search

In this chapter we investigate the common algorithmic thread linking the graph search paradigms we explored in the previous chapter.[1] Our goal is to derive a common algorithm that is sufficient to express each of the discussed search algorithms simply by inserting an appropriate kind of structure. Core to this work is the formalization of a vertex labelling algorithm, which we call *maximal label search* (MLS). Though it is not a search algorithm in the sense of graph search, we find it satisfies our stated goals, and it can be extended to address related issues.

We employ labelling schemes as the structures that specialize our generic algorithm to obtain each search paradigm. A labelling scheme describes how labels should be assigned and updated while processing a graph, and determines how vertices should be chosen. We present characterizations of labelling schemes that correspond to each of the main graph search paradigms, results that will be extended in Chapter 6. We will find in Chapter 5 that our algorithm is convenient for deriving implementation time and space bounds.

[1]The results of this chapter are the subject of [BKS].

4.1 A vertex labelling algorithm

All basic graph search algorithms proceed in a standard manner: iteratively visiting a new vertex that is adjacent to a previously visited vertex whenever possible. Once a vertex is visited, any of its unvisited neighbours become eligible for being visited next, though the exact selection criteria vary among different types of search.

When performing a search of some graph, we consider each unvisited vertex to have associated with it some label indicating its "priority" for being visited next. For example, a vertex with no visited neighbours would have lower priority than a vertex with a visited neighbour, and hence have a lower-valued label. As additional neighbours of a vertex are visited, its priority may be increased (or even decreased) for some search paradigms. When the next vertex is chosen to be visited, there should be no other unvisited vertex with higher priority.

We proceed to formally codify this idea as a generic algorithm. First we specify that the "label" of a vertex, $label(v)$, must be a member of some set L. A vertex's label may change many times during the execution of the search. The "priority" of a vertex for being visited next is expressed mathematically as an ordering relation on the labels associated with the vertices. This may be a partial ordering, since there may be many different labels which may be chosen at any point in the search.

Once a vertex is chosen, the unvisited vertices in its neighbourhood may become more desirable to choose next. Algorithmically, we need to define a function $INC(l, i)$ that will update the label $l = label(v)$ of an unvisited vertex v when one of its neighbours has been chosen as the ith vertex. The $INC(l, i)$ function will be used to update $label(v)$ to a new element in L (which may depend on i) to reflect the new priority of choosing v in our search. We note that, for a particular graph G of order n, $i \in \{1, \ldots, n-1\}$, but the value of n cannot be known a priori. Both our label set L and INC function must consider n as unknown.

As our INC function requires a previous label, each vertex must initially have some label associated with it. Since a search may begin at any vertex in the graph and each vertex should initially be treated identically, it is convenient to select a single element l_0 of L to serve as the initial label for all vertices. We assume no restriction on the location of l_0 in the partial order, though for graph search it is typical that l_0 is a minimum element.

We combine these four concepts into a labelling scheme that, once specified, can be used to develop an algorithm for various types of search.

Definition 4.1.1. *We say structure $(L, \preceq, l_0, INC)$ is a labelling scheme if*

- *L is a set (the set of labels, possibly infinite in general),*
- *$\preceq$ is a partial order on the label set L (the priority order),*
- *l_0 is a distinguished element of L (used as the initial label for all vertices), and*
- *INC is a function from $L \times \mathbb{Z}^+$ to L.*

As a labelling scheme is independent of the input graph, the number of vertices n in the input graph is unknown. The set of labels L is often infinite, though only a finite number of labels will ever be used for any given input graph (since the graph itself is assumed to be finite).

With this notation, the algorithm for performing a graph search is straightforward. A graph search algorithm needs only iteratively choose a vertex with label maximal over all unvisited vertices, and update the labels of its neighbours with the INC function. We name this algorithm Maximal Label Search (MLS) and describe this as Algorithm 10.

If σ is an ordering of G found by Algorithm MLS using a labelling scheme S, we say that σ is an S-MLS ordering of G, or, equivalently, an S-MLS(G) ordering. When we restrict our attention to a particular labelling scheme S, we will call the algorithm S-MLS and if, moreover, we consider a particular graph G, we call the algorithm S-MLS(G).

Algorithm 10: Maximal Label Search (MLS)

input : a graph $G = (V, E)$ and a labelling scheme $S = (L, \preceq, l_0, INC)$

output: an ordering σ of V

assign the label l_0 to all vertices
for $i \leftarrow 1$ *to* n **do**
  pick an unnumbered vertex v whose label is maximal under $\preceq$
  $\sigma(i) \leftarrow v$ //This assigns to v the number i
  foreach *unnumbered vertex* w *adjacent to* v **do**
    $label(w) \leftarrow INC(label(w), i)$

Again, we use $label_i(v)$ to denote the label assigned to vertex v at the *start* of iteration i of the algorithm. The following properties follow directly from the algorithm.

Lemma 4.1.2. *Let G be a graph and S be a labelling scheme. Then in any execution of S-MLS on G resulting in ordering $\sigma = (v_1, \ldots, v_n)$, the following statements hold for all integers $i, j \geq 1$ and vertices $a, b \in V$:*

i. If $i < j$, then $label_i(v_i) \not\prec label_i(v_j)$.

ii. If $v_i a \notin E$, then $label_{i+1}(a) = label_i(a)$.

iii. If $label_{i+1}(a) \neq label_i(a)$, then $v_i a \in E$ and $label_{i+1}(a) = INC(label_i(a), i)$.

iv. If $v_i < a < b$ in σ and, for all $j < i$, $v_j a \in E$ precisely when $v_j b \in E$, then $label_i(a) = label_i(b)$.

We note that during the execution of the search, the label of each vertex is either l_0 or the result of some application of the INC function. Additionally, the vertices are visited in order from 1 to n, so there are a limited number of labels that could possibly be assigned to vertices at each iteration of the algorithm. We now define those labels that could be assigned to a vertex at each iteration of the search.

Definition 4.1.3. *Let $S = (L, \preceq, l_0, INC)$ be a labelling scheme. For $i \geq 1$, let $L_i \subseteq L$ be defined recursively by*

- $L_1 = \{l_0\}$, *and*
- $L_{i+1} = L_i \cup \{l \in L : l = INC(l', i)$ *for some* $l' \in L_i\}$ *for* $i \geq 1$.

We say $l \in L$ is a *legal label* (with respect to labelling scheme S) *at step* i, for $i \geq 1$, if we have $l \in L_i$. The set L_i corresponds to those labels that could possibly be assigned to a vertex at the *start* of iteration i of the MLS algorithm. If we are considering a graph on n vertices and have $l \in L_n$, then we simply say that l is a *legal label*. From this definition, we can derive a bound on how many different legal labels we can have for each iteration of MLS.

Lemma 4.1.4. *Let $(L, \preceq, l_0, INC)$ be a labelling scheme. Then $|L_i| \leq 2^{i-1}$ for $i \geq 0$.*

Proof. From Definition 4.1.3, $|L_i| \leq 2 \cdot |L_{i-1}|$ for $i > 1$. Since $|L_1| = 1$, we have the desired bound. □

We stress that MLS need not yield a graph search algorithm when given an arbitrary labelling scheme. Though many graph search algorithms can be expressed as a type of MLS algorithm (as we will see shortly), there are labelling schemes S such that the orderings that S-MLS finds need not be graph search orderings. It is important to note that the use of the word "Search" in the name Maximal Label Search refers to fact that it iteratively finds a vertex whose label is maximal, and should not imply any a priori relationship with graph search.

4.1.1 Example labelling schemes for well-known graph searches

To illustrate how these labelling schemes describe various graph searches, we now describe labelling schemes that correspond to each of the well-known graph search paradigms within the framework of Algorithm MLS. We will prove these schemes yield appropriate search orderings in Section 4.2.

Generic graph search (labelling scheme S_{GS}): $L = \{0, 1\}$, $\preceq$ is the natural order $\leq$ on integers, $l_0 = 0$, and $INC(l, i) = 1$. The label corresponds to a Boolean indicating whether a vertex is adjacent to a visited vertex.

BFS (labelling scheme S_{BFS}): $L = \{1, 2, \dots\} \cup \{\infty\}$, $\preceq$ is the natural order $\geq$ on integers, $l_0 = \infty$, and $INC(l, i)$ equals i if $l = \infty$ and equals l otherwise. The label corresponds to the vertex's parent in the BFS-tree, or ∞ if none of its neighbours has been visited.

DFS (labelling scheme S_{DFS}): $L = \{0, 1, \dots\}$, $\preceq$ is the natural order $\leq$ on integers, $l_0 = 0$, and $INC(l, i) = i$. The label corresponds to the vertex's most recently visited neighbour, or 0 if no neighbour has been visited.

MNS (labelling scheme S_{MNS}): L is the set of subsets of $\{1, 2, \dots\}$, $\preceq$ is set inclusion $\subseteq$ (not a total order), l_0 is the empty set, and $INC(l, i) = l \cup \{i\}$. The label corresponds to the set of already visited neighbours of a vertex.

LexBFS (labelling scheme S_{LexBFS}): L is the set of strings over the alphabet $\{1, 2, \dots\}$, $\preceq$ is the lexicographic order in which digit i is considered larger than digit j when $i < j$, l_0 is the empty string ε, and $INC(l, i)$ is digit i appended to string l. For convenience, we term this order $\preceq$ "inverted $\leq_L$."

LexDFS (labelling scheme S_{LexDFS}): L is the set of strings over the alphabet $\{1, 2, \dots\}$, $\preceq$ is the usual lexicographic order $\leq_L$, l_0 is the empty string ε, and $INC(l, i)$ is digit i prepended to string l.

MCS (labelling scheme S_{MCS}): $L = \{0, 1, \dots\}$, $\preceq$ is the natural order $\leq$ on integers, $l_0 = 0$, and $INC(l, i) = l + 1$. The label corresponds to the number of already visited neighbours of a vertex.

We note that the last four labelling schemes all compute perfect elimination orderings of chordal graphs. Berry, Krueger and Simonet study this aspect in detail in [BKS05].

4.2 Graph search labelling schemes

We note that not all labelling schemes yield a graph search algorithm. The labelling scheme S and its INC function must satisfy a further property to ensure that the vertex orderings produced by S-MLS are valid graph search orderings.

Definition 4.2.1. *We say labelling scheme $(L, \preceq, l_0, INC)$ is a graph search labelling scheme if the following property holds:*

(L1): minimum initial label: $l_0 \prec INC(l, i)$, for all $i \geq 1$ and $l \in L_i$.

We now show that any scheme that forms a graph search labelling scheme corresponds to some graph search algorithm.

Lemma 4.2.2. *Let G be a graph, let S be a graph search labelling scheme, and let σ be an ordering of V. If σ can be computed by S-MLS(G), then σ is a graph search ordering.*

Proof. Suppose σ is an ordering found by MLS using some graph search labelling scheme S. Suppose for contradiction that σ is not a graph search ordering. Then, by Theorem 3.1.2, (S) is violated. So there exists a triple of vertices $a < b < c$ with $ac \in E$, $ab \notin E$ such that for all $d < b$, $db \notin E$. Then $label_b(b) = l_0$. But since $ac \in E$, we have $label_b(c) = INC(label_i(c), i)$ for some $i < \sigma^{-1}(b)$. By property (L1), $label_b(c) \succ l_0 = label_b(b)$, contradicting the selection of b by the algorithm. □

We note that $S_{GS} = (\{0, 1\}, \leq, 0, INC(l, i) = 1)$, the labelling scheme presented for generic graph search in Section 4.1.1, satisfies (L1), and thus is a graph search labelling scheme.

Lemma 4.2.3. *Let G be a graph and let σ be a graph search ordering of V. Then σ can be computed by S_{GS}-MLS(G).*

Proof. Suppose $\sigma = (v_1, \ldots, v_n)$ is a graph search ordering of G. Suppose $v_1, \ldots, v_{i-1}$ can be found by S_{GS}, but $v_1, \ldots, v_i$ cannot, for some $1 < i \leq n$. Then $label_i(v_i)$ is not maximal at iteration i. Let $w > v_i$ be an unnumbered vertex of maximal label at iteration i with $label_i(w) \succ label_i(v_i)$. Then $label_i(w) = 1$ and $label_i(v_i) = 0$. Thus v_i has no neighbour before it in σ. Since the label of w was updated, there is a $j < i$ with $v_j w \in E$ and $v_j v_i \notin E$. By Theorem 3.1.2, and applying (S) to the triple $v_j < v_i < w$, there is a $k < i$ with $v_k v_i \in E$, a contradiction. □

The following theorem follows directly.

Theorem 4.2.4. *Let G be a graph and let σ be an ordering of V. Then σ is a graph search ordering of G if and only if there exists a graph search labelling scheme S such that σ can be computed by S-MLS(G).*

Proof. By Lemmas 4.2.2 and 4.2.3 and the fact that S_{GS} is a graph search labelling scheme. □

4.2.1 Characterizing well-known graph searches

Additional properties can be specified such that, when satisfied by a specified labelling scheme S, we are assured that S-MLS will compute a particular type of graph search. The following properties will be useful to characterize such schemes, and we present a few facts regarding the execution of MLS using such schemes.

Property 4.2.5. *Let $S = (L, \preceq, l_0, INC)$ be a labelling scheme. We define the following properties for S:*

(L2): monotonic nondecreasing: $l \preceq INC(l, i)$, for all $i \geq 1$ and $l \in L_i$.

(L3): monotonic increasing: $l \prec INC(l, i)$, for all $i \geq 1$ and $l \in L_i$.

(L4): stability: if $l \prec l'$ then $INC(l, i) \prec INC(l', i)$, for all $i \geq 1$ and $l, l' \in L_i$.

Lemma 4.2.6. *Let G be a graph and let S be a labelling scheme. Then in any execution of S-MLS on G resulting in ordering $\sigma = (v_1, \ldots, v_n)$, the following statements hold for all integers $i, j \geq 1$ and vertices $a, b \in V$:*

i. If S is monotonic nondecreasing, then, for $i < j$, $label_i(a) \preceq label_j(a)$.

ii. If S is monotonic nondecreasing and stable, and if $label_i(a) \prec label_i(b)$ for some $v_i \leq a, b$, then there exists $j < i$ such that $v_j b \in E$ and $v_j a \notin E$.

Proof. i. Direct from transitivity of $\preceq$ and (L2).

ii. We prove a strengthening of the contrapositive. Suppose for all $j < i$, $v_j b \in E$ implies $v_j a \in E$. We prove $label_i(b) \preceq label_i(a)$, which implies $label_i(a) \not\prec label_i(b)$. To do this, we show by induction that $label_{j+1}(b) \preceq label_{j+1}(a)$ for all $j < i$. For $j = 0$, $label_1(b) = label_1(a) = l_0$, and our claim holds. Now let $0 < j < i$ and assume $label_j(b) \preceq label_j(a)$. If $v_j b \in E$, then $v_j a \in E$, and so $label_{j+1}(b) = INC(label_j(b), j) \preceq INC(label_j(a), j) = label_{j+1}(a)$ by stability (L4). Otherwise, $v_j b \notin E$, so $label_{j+1}(b) = label_j(b)$. By (L2) and part (i), $label_j(a) \preceq label_{j+1}(a)$. So $label_{j+1}(b) \preceq label_{j+1}(a)$ holds by transitivity. Thus our claim is proven for all $j < i$. □

We now proceed to state and prove characterizations of labelling schemes that produce each of the major graph search paradigms. We note that in each case, the minimum initial label property (L1) is either a condition of the characterization, or is subsumed by the characterizing conditions.

Proposition 4.2.7 ([BKS05]). *Let G be a graph and let σ be an ordering of V. Then σ is an MNS ordering if and only if there is a labelling scheme S such that σ can be computed by S-MLS, where S satisfies the properties*

(L3): monotonic increasing: $l \prec INC(l, i)$, for all $i \geq 1$ and $l \in L_i$,[2] and

(L4): stability: if $l \prec l'$, then $INC(l, i) \prec INC(l', i)$, for all $i \geq 1$ and $l, l' \in L_i$.

Proof. Suppose σ is an ordering found by S-MLS for some labelling scheme S satisfying the properties above. If σ is not an MNS ordering, then, by Theorem 3.5.1,

[2]This property subsumes (L1) and (L2).

(M) is violated. So there are vertices $a < b < c$ with $ac \in E$, $ab \notin E$ and for every $d < b$, if $db \in E$ then $dc \in E$. Let $a < b < c$ be such vertices with a leftmost in σ. Then $label_a(b) = label_a(c)$ by Lemma 4.1.2(iv). By (L3), $label_{a+1}(c) = INC(label_a(c), \sigma^{-1}(a)) \succ label_a(c) = label_a(b) = label_{a+1}(b)$. By (L4), $label_b(b) \prec label_b(c)$, contradicting the choice of b by MLS.

For the converse, suppose $\sigma = (v_1, \dots, v_n)$ is an MNS ordering. Let S be the labelling scheme $S_{MNS} = (\{\text{subsets of } \{1, 2, \dots\}\}, \subseteq, \emptyset, INC(l, i) = l \cup \{i\})$ described in Section 4.1.1. We note S_{MNS} satisfies (L3) and (L4). Suppose $v_1, \dots, v_{i-1}$ can be found by S-MLS, but $v_1, \dots, v_i$ cannot, for some $1 < i \leq n$. Then $label_i(v_i)$ is not maximal at iteration i. Let $w > v_i$ be an unnumbered vertex of maximal label at iteration i with $label_i(v_i) \subset label_i(w)$. Then there is a $j < i$ with $j \in label_i(w)$ and $j \notin label_i(v_i)$, hence $v_j w \in E$ and $v_j v_i \notin E$. By Theorem 3.5.1 and applying (M) to the triple $v_j < v_i < w$, there is a $k < i$ such that $v_k v_i \in E$ and $v_k w \notin E$. Thus $label_i(v_i) \not\subset label_i(w)$, a contradiction. □

Proposition 4.2.8. *Let G be a graph and let σ be an ordering of V. Then σ is a DFS ordering of G if and only if there is a labelling scheme S such that σ can be computed by S-MLS, where S satisfies the property*

(L5): depth-firstness: $l \prec INC(l', i)$, for all $i \geq 1$ and $l, l' \in L_i$.[3]

Proof. Suppose σ is an ordering found by S-MLS for some labelling scheme S satisfying (L5). If σ is not a DFS ordering, then, by Theorem 3.3.1, (D) is violated. So there exist vertices $a < b < c$ with $ac \in E$, $ab \notin E$ such that for all d, $a < d < b$, $db \notin E$. Then $label_a(b) = label_b(b)$ by induction on Lemma 4.1.2(ii). By (L5), $label_a(b) \prec INC(label_a(c), \sigma^{-1}(a)) = label_{a+1}(c) \preceq label_b(c)$ by Lemma 4.2.6(i) and the fact that (L5) subsumes (L2). Thus $label_b(b) \prec label_b(c)$, contradicting the selection of b by MLS.

For the converse, suppose $\sigma = (v_1, \dots, v_n)$ is a DFS ordering. Let S be the labelling

[3] This property subsumes (L1), (L2) and (L3).

scheme $S_{DFS} = (\{0, 1, \dots\}, \leq, 0, INC(l, i) = i)$ described in Section 4.1.1. We note S_{DFS} satisfies (L5). Suppose $v_1, \dots, v_{i-1}$ can be found by S-MLS, but $v_1, \dots, v_i$ cannot, for some $1 < i \leq n$. Then $label_i(v_i)$ is not maximal at iteration i. Let $w > v_i$ be an unnumbered vertex of maximal label at iteration i with $label_i(v_i) \prec label_i(w)$. Thus $label_i(w) = INC(label_j(w), j) = j$ for some $j < i$. Then $v_j w \in E$ and $v_j v_i \notin E$. By Theorem 3.3.1, (D) holds on the triple $v_j < v_i < w$, so there is some k, $j < k < i$, with $v_k v_i \in E$; choose the largest such k. Then $label_i(v_i) = k > j$, contradicting our assumption. □

Proposition 4.2.9. *Let G be a graph and let σ be an ordering of V. Then σ is a BFS ordering of G if and only if there is a labelling scheme S such that σ can be computed by S-MLS, where S satisfies the properties*

(L1): minimum initial label: $l_0 \prec INC(l, i)$, for all $i \geq 1$ and $l \in L_i$,

(L2): monotonic nondecreasing: $l \preceq INC(l, i)$, for all $i \geq 1$ and $l \in L_i$, and

(L6): breadth-firstness: if $l \prec l'$ then $INC(l, i) \prec l'$, for all $i \geq 1$ and $l, l' \in L_i$.

Proof. Suppose σ is an ordering found by S-MLS for some labelling scheme S satisfying the properties above. If σ is not a BFS ordering, then, by Theorem 3.2.1, (B) is violated. So there exists vertices $a < b < c$ with $ac \in E$, $ab \notin E$ such that for all $d < a$, $db \notin E$. Then $label_{a+1}(b) = l_0$ by repeated applications of Lemma 4.1.2(ii) and, by (L1), $label_{a+1}(c) = INC(label_a(c), \sigma^{-1}(a)) \succ l_0$. By induction on (L6), $label_b(b) \prec label_{a+1}(c)$. By (L2), $label_{a+1}(c) \preceq label_b(c)$. Thus $label_b(b) \prec label_b(c)$, contradicting the choice of b by MLS.

For the converse, suppose $\sigma = (v_1, \dots, v_n)$ is a BFS ordering. Let S be the labelling scheme $S_{BFS} = (\{1, 2, \dots\} \cup \{\infty\}, \geq, \infty, INC)$ described in Section 4.1.1, where $INC(l, i) = i$ if $l = \infty$ and $INC(l, i) = l$ otherwise. We note S_{BFS} satisfies (L1), (L2) and (L6). Suppose $v_1, \dots, v_{i-1}$ can be found by S-MLS, but $v_1, \dots, v_i$ cannot, for some

$1 < i \leq n$. Then $label_i(v_i)$ is not maximal. Let $w > v_i$ be an unnumbered vertex of maximal label at iteration i with $label_i(v_i) \prec label_i(w)$. Then $label_i(w) = INC(label_j(w), j)$ for some $j < i$ with $v_j w \in E$; choose the smallest such j. Then $label_i(w) = j$. By (L2) and Lemma 4.2.6(i), $label_{j+1}(v_i) \preceq label_i(v_i)$. If $v_j v_i \in E$, then $label_{j+1}(v_i) = INC(label_j(v_i), j) \succeq j = label_i(w)$, which contradicts our choice of w. Hence $v_j v_i \notin E$. By Theorem 3.2.1, (B) holds on the triple $v_j < v_i < w$, so there is a $k < j$ such that $v_k v_i \in E$. Thus $label_i(v_i) \succeq label_{k+1}(v_i) = INC(label_k(v_i), k) \succeq k \succ j = label_i(w)$, which contradicts our choice of w. □

Proposition 4.2.10. *Let G be a graph and let σ be an ordering of V. Then σ is a LexBFS ordering of G if and only if there is a labelling scheme S such that σ can be computed by S-MLS, where S satisfies the properties*

> *(L3): monotonic increasing: $l \prec INC(l, i)$, for all $i \geq 1$ and $l \in L_i$,*[4] *and*

> *(L6): breadth-firstness: if $l \prec l'$ then $INC(l, i) \prec l'$, for all $i \geq 1$ and $l, l' \in L_i$.*[5]

Proof. Suppose σ is an ordering found by S-MLS for some labelling scheme S satisfying the properties above. If σ is not a LexBFS ordering, then, by Theorem 2.3.1, (LB) is violated. So there exists vertices $a < b < c$ with $ac \in E$, $ab \notin E$ such that for all $d < a$, $db \in E$ implies $dc \in E$; choose a triple with a leftmost in σ. Then for all $d < a$, $db \in E$ if and only if $dc \in E$, so $label_a(b) = label_a(c)$ by Lemma 4.1.2(iv). Then $label_a(c) \prec label_{a+1}(c)$ by (L3) and $label_{a+1}(b) = label_a(b)$. By induction on (L6), $label_b(b) \prec label_{a+1}(c)$. Since (L3) subsumes (L2), $label_{a+1}(c) \preceq label_b(c)$ by Lemma 4.2.6(i). Thus $label_b(b) \prec label_b(c)$, contradicting the choice of b by the algorithm.

For the converse, suppose $\sigma = (v_1, \ldots, v_n)$ is a LexBFS ordering. Let S be the labelling scheme $S_{LexBFS} = (\{\text{strings over } \{1, 2, \ldots\}\}, \text{inverted } \leq_L, \varepsilon, INC)$, where $INC(l, i) =$

---

[4]This property subsumes (L1) and (L2).

[5]These two properties imply (L4) also holds.

$l \circ i$, described in Section 4.1.1. We note S_{LexBFS} satisfies (L3) and (L6). Suppose $v_1, \ldots, v_{i-1}$ can be found by S-MLS, but $v_1, \ldots, v_i$ cannot, for some $1 < i \leq n$. Then $label_i(v_i)$ is not maximal. Let $w > v_i$ be an unnumbered vertex of maximal label at iteration i with $label_i(v_i) \prec label_i(w)$. Since (L3) and (L6) imply that (L2) and (L4) hold, by Lemma 4.2.6(ii) there is a $j < i$ with $v_j w \in E$ and $v_j v_i \notin E$; pick j smallest. By Theorem 2.3.1, (LB) holds on the triple $v_j < v_i < w$, so there is a $k < j$ such that $v_k v_i \in E$ and $v_k w \notin E$; pick k smallest. Then, for all $u < v_k$, $uv_i \in E$ and $uw \in E$, so $label_k(v_i) = label_k(w)$ by Lemma 4.1.2(iv), and $label_{k+1}(v_i) = INC(label_k(v_i), k) \succ label_k(v_i) = label_k(w) = label_{k+1}(w)$ by (L3). By (L6) and induction, $label_i(w) \prec label_{k+1}(v_i) \preceq label_i(v_i)$, which contradicts our choice of w. □

Proposition 4.2.11. *Let G be a graph and σ be an ordering of V. Then σ is a LexDFS ordering of G if and only if there is a labelling scheme S such that σ can be computed by S-MLS, where S satisfies the properties*

(L5): depth-firstness: $l \prec INC(l', i)$, for all $i \geq 1$ and $l, l' \in L_i$,[6] *and*

(L4): stability: if $l \prec l'$ then $INC(l, i) \prec INC(l', i)$, for all $i \geq 1$ and $l, l' \in L_i$.

Proof. Suppose σ is an ordering found by S-MLS for some labelling scheme S satisfying the properties above. If σ is not a LexDFS ordering, then, by Theorem 3.4.1, (LD) is violated. So there exists vertices $a < b < c$ with $ac \in E$, $ab \notin E$ such that for all d, $a < d < b$, $db \in E$ implies $dc \in E$; choose a triple with a rightmost. By (L5), $label_{a+1}(c) \succ label_a(b) = label_{a+1}(b)$. By induction on (L4), $label_b(b) \prec label_b(c)$, contradicting the choice of b by the algorithm.

For the converse, suppose $\sigma = (v_1, \ldots, v_n)$ is a LexDFS ordering. Let S be the labelling scheme $S_{LexDFS} = (\{\text{strings over } \{1, 2, \ldots\}\}, \leq_L, \varepsilon, INC)$, where $INC(l, i) = i \circ l$, described in Section 4.1.1. We note S_{LexDFS} satisfies (L5) and (L4). Suppose $v_1, \ldots, v_{i-1}$ can be found by S-MLS, but $v_1, \ldots, v_i$ cannot, for some $1 < i \leq n$. Then

[6] This property subsumes (L1), (L2) and (L3).

$label_i(v_i)$ is not maximal. Let $w > v_i$ be an unnumbered vertex of maximal label at iteration i with $label_i(v_i) \prec label_i(w)$. Since (L5) subsumes (L2), by Lemma 4.2.6(ii) there is some $j < i$ such that $v_j w \in E$ and $v_j v_i \notin E$ and for all k, $j < k < i$, $v_k w \in E$ if and only if $v_k v_i \in E$. By Theorem 3.4.1, (LD) holds on triple $v_j < v_i < w$, so there is a vertex d, $v_j < d < v_i$ with $dv_i \in E$ but $dw \notin E$, a contradiction. □

Proposition 4.2.12. *Let G be a graph and let σ be an ordering of V. Then σ is an MCS ordering of G if and only if there is a labelling scheme S such that σ can be computed by S-MLS, where S satisfies the properties*

(L3): monotonic increasing: $l \prec INC(l, i)$, for all $i \geq 1$ and $l \in L_i$,[7] and

(L7): independence: $INC(l, i) = INC(l, i')$, for all $i, i' \geq 1$ and $l \in L_i$.[8]

Proof. Suppose σ is an ordering found by S-MLS for some labelling scheme S satisfying the properties above. If σ is not an MCS ordering, then by Theorem 2.4.2, (MC) is violated. So there exists vertices $a_1 < \cdots < a_k < b < c$ with $a_i c \in E$, $a_i b \notin E$ and for all choices $d_1 < \cdots < d_k < b$, there is some i, $1 \leq i \leq k$, such that either $d_i b \notin E$ or $d_i c \in E$; choose such a set of vertices with k maximum. Let $A = \{v < b : vc \in E, vb \notin E\}$, $D = \{v < b : vb \in E, vc \notin E\}$, and $B = \{v < b : vb \in E, vc \in E\}$. Then $A = \{a_1, \ldots, a_k\}$ and $|D| < k$. Define function $Inc^r(l) = INC(Inc^{r-1}(l), 1)$ for $r > 0$ and $Inc^r(l) = l$ for $r = 0$. By (L7), the value of $Inc^r(l)$ is precisely INC applied to label l exactly r times. By (L3), $Inc^r(l) \prec Inc^{r'}(l)$ if and only if $r < r'$. Now, $label_b(c) = Inc^{|A \cup B|}(l_0)$ and $label_b(b) = Inc^{|D \cup B|}(l_0)$. Since $|D| < |A|$, $label_b(b) \prec label_b(c)$, contradicting the choice of b by MLS.

For the converse, suppose $\sigma = (v_1, \ldots, v_n)$ is an MCS ordering. Let S be the labelling scheme $S_{MCS} = (\{0, 1, \ldots\}, \leq, 0, INC(l, i) = l + 1)$ described in Section 4.1.1. We note S_{MCS} satisfies (L3) and (L7). Suppose $v_1, \ldots, v_{i-1}$ can be found by S-MLS, but

[7] This property subsumes (L1) and (L2).

[8] These two properties, and the fact that all vertices have the same initial label l_0, imply (L4) also holds.

$v_1, \ldots, v_i$ cannot, for some $1 < i \leq n$. Then $label_i(v_i)$ is not maximal. Let $w > v_i$ be an unnumbered vertex of maximal label at iteration i. Then $label_i(v_i) < label_i(w)$. Let $W = \{v < v_i : vw \in E\}$. Then $|W| = label_i(w)$. Let $X = \{v \in W : vv_i \in E\}$. Let $k = |W - X|$. Then $W - X = \{a_1, \ldots, a_k\}$. By Theorem 2.4.2, (MC) holds on $a_1 < \cdots < a_k < v_i < w$, so there are vertices $d_1 < \cdots < d_k < v_i$ such that $d_j v_i \in E$ and $d_j w \notin E$ for $1 \leq j \leq k$. So $label_i(v_i) \geq |X| + k = |W| = label_i(w)$, a contradiction. □

4.2.2 Other schemes

We note that not all search-like algorithms can be expressed using a labelling scheme. MLS does not "look ahead" into the unexplored portion of the graph, and thus cannot make decisions based on any information that can be gathered, such as determining connected components in the unexplored subgraph. Thus the MEC and MCC algorithms of Shier [Shi84], or the comp-MLS generalizations of Berry, Krueger and Simonet [BKS05] (which we discuss in Section 7.1.3) require information that MLS itself cannot obtain.

MLS does, however, express some interesting algorithms that are not graph searches. For example, we can construct labelling schemes that allow MLS to compute a LexBFS or LexDFS ordering of the complement of a graph without computing that complement. We do this by reversing the partial order, in effect instructing MLS to choose vertices with minimal label. We investigate such schemes in Section 6.1, and find which characterizing properties of labelling schemes can and cannot find particular graph search orderings of the complement of a graph.

4.3 Extended labelling schemes

For many applications a single sweep of a graph search algorithm is insufficient to reveal enough structure of a graph to recognize membership is a specific graph class or to prove desired properties about the graph. The information collected from an initial sweep can however be used to direct subsequent sweeps through the graph. As a simple example, the ending vertex of an initial BFS sweep can be used as the start vertex of a second BFS sweep to find the diameter of a tree [Han73]. As a more involved example, Corneil, Olariu and Stewart [COS95] present an algorithm that basically performs two enhanced BFS sweeps of an AT-free graph, with the second starting where the first ended, to deduce a dominating path.

This idea of multiple sweeps can be extended to use the entire vertex ordering to help break ties in a subsequent sweep. This idea has successfully been applied to LexBFS to create a LexBFS$^+$ algorithm [Ma, Sim91], which has been used in a number of contexts [Cor04a]. Corneil, Olariu and Stewart [COS98] further extend this idea by using the orderings computed by two previous sweeps to compute a LexBFS* ordering, which they use to recognize interval graphs. Bretscher et al. [BCHP03] introduce a related LexBFS$^-$ algorithm that, using a previously computed ordering, computes a LexBFS ordering of the complement of a graph.

We extend our definition of a labelling scheme to include such types of algorithms.

Definition 4.3.1. *We say structure $(L, \preceq, L_0, INC)$ is an extended labelling scheme if*

- *L is a set (the set of labels),*
- *$\preceq$ is a partial order on the label set L (the priority order),*
- *$L_0 = (l_1, l_2, \dots)$ is a sequence of distinguished elements of L (used as the initial labels), possibly with repeated elements, such that $l_1 \preceq l_2 \preceq \dots$ or $l_1 \succeq l_2 \succeq \dots$, and*
- *INC is a function from $L \times \mathbb{Z}^+$ to L.*

We extend MLS by, instead of initializing all vertices to the same label l_0, initializing the vertices to L_0 based on a given vertex ordering. We call this extension MLS$^+$, and present it as Algorithm 11.

Algorithm 11: Extended Maximal Label Search (MLS$^+$)

input : an extended labelling scheme $(L, \preceq, L_0, INC)$, a graph $G = (V, E)$ and an ordering τ of V

output: an ordering σ of V

for $i \leftarrow 1$ *to* n **do**
  $label(\tau(i)) \leftarrow l_i \ (\in L_0)$

for $i \leftarrow 1$ *to* n **do**
  pick an unnumbered vertex v whose label is maximal under $\preceq$
  $\sigma(i) \leftarrow v$ //This assigns to v the number i
  foreach *unnumbered vertex* w *adjacent to* v **do**
    $label(w) \leftarrow INC(label(w), i)$

We say σ is an S-MLS$^+(G, \tau)$ ordering if Algorithm 11 can find σ given G and an ordering τ using the extended labelling scheme S. To bound the number of possible labels, we must know n the number of vertices in the input graph, as the sequence L_0 itself may contain an unbounded number of different labels. For $n \geq 1$, we define $L_i(n) \subseteq L$ for $1 \leq i \leq n$ by

- $L_1(n) = \{l_1, \ldots, l_n\}$, and
- $L_{i+1}(n) = L_i(n) \cup \{INC(l, i) : l \in L_i(n)\}$ for $1 \leq i < n$.

Every label that could possibly be assigned to a vertex at the *start* of step i for a graph on n vertices is in $L_i(n)$, though some labels in $L_i(n)$ might never be possible to assign to a vertex (for example, in the case when l_1 is maximal and $l_1 \succ l_2$, the label $l = INC(l_1, 1)$ is in $L_2(n)$ even though it could never be assigned as a vertex label unless $l \in L_0$ or $l = INC(l', i)$ for some suitable $l' \neq l_1$).

Extended labelling schemes generalize labelling schemes in that, given any labelling scheme S, one can devise an extended labelling scheme S' such that any ordering found by S-MLS can be found by S'-MLS$^+$.

Lemma 4.3.2. *Let σ be an S-MLS ordering of G for some labelling scheme S. Then σ is an S'-MLS$^+(G, \tau)$ ordering for some extended labelling scheme S', where τ is an arbitrary ordering of V.*

Proof. Suppose $S = (L, \preceq, l_0, INC)$. Let τ be any ordering of V. Let $S' = (L, \preceq, L_0, INC)$ where $L_0 = (l_0, l_0, \ldots)$. Then each vertex will have the same initial label in both algorithms. Since the remaining lines of the two algorithms are identical, MLS$^+$ can compute σ. □

Thus computing an S-MLS ordering is asymptotically no worse than running S'-MLS$^+$. We also note that we can obtain a bound on the number of labels the MLS$^+$ algorithm may possibly use (for a particular extended labelling scheme and number of vertices in a graph) that is similar to our bound on the number of legal labels for MLS.

Lemma 4.3.3. *Let $(L, \preceq, L_0, INC)$ be an extended labelling scheme and let G be a graph. Then $|L_n(n)| \leq n2^n$.*

Proof. We have $|L_0(n)| \leq n$, and within n iterations of the MLS$^+$ algorithm each of these initial labels can be updated in up to 2^n ways. Thus $|L_n(n)| \leq n2^n$. □

Thus both MLS and MLS$^+$ require an asymtotically equivalent number of bits to express the labels that may be used. These two lemmas suggest that MLS and MLS$^+$ have in some sense equivalent time and space bounds for their implementation, which we will see in the next chapter.

4.4 Summary

We have presented two algorithms, MLS and MLS$^+$ that generalize graph search by assigning labels to vertices and choosing a vertex with maximal label at each iteration.

Table 4.1: Labelling schemes for each of the main types of graph search.

Search	Set of Labels	l_0	Order	$INC(l, i)$
generic	$\{0, 1\}$	0	$\leq$	1
BFS	$\{1, 2, \ldots\} \cup \{\infty\}$	∞	$\geq$	i if $l = \infty$, l o.w.
DFS	$\{0, 1, \ldots\}$	0	$\leq$	i
MNS	$P(\{1, 2, \ldots\})$	$\emptyset$	$\subseteq$	$l \cup \{i\}$
LexBFS	$\{1, 2, \ldots\}^*$	ε	inverted $\leq_L$	$l \circ i$
LexDFS	$\{1, 2, \ldots\}^*$	ε	$\leq_L$	$i \circ l$
MCS	$\{0, 1, \ldots\}$	0	$\leq$	$l + 1$

We found that the major search paradigms can be expressed in this phrasing simply by specifying how labels are assigned to vertices and which labels should be considered larger then others. Example labelling schemes for each of the main search paradigms are summarized in Table 4.1. We note that labelling schemes are oblivious to the number of vertices in or the size of the input graph.

Furthermore, we found that by imposing restrictions on the labelling schemes we used with the MLS algorithm, we could characterize labelling algorithms that computed particular search orderings. Each of the main graph search paradigms have simple such characterizations that provide insight into how the search explores a graph. These characterizations, and a quick reference of the characterizing properties, are listed in Table 4.2.

We also saw how this labelling scheme approach can easily be extended to capture LexBFS$^+$ type searches, which are useful for devising multiple sweep algorithms. Though we leave implementation to the next chapter, we have noted that both MLS and MLS$^+$ will have somewhat equivalent asymptotic time bounds on their implementations.

Table 4.2: Labelling scheme characterizations for graph search. Checkmarks indicate properties used in characterization, asterisks indicate properties implied by checkmark properties.

Graph Search	Characterizing Properties (L1)	(L2)	(L3)	(L4)	(L5)	(L6)	(L7)
generic	✓						
BFS	✓	✓				✓	
DFS	*	*	*		✓		
MNS	*	*	✓	✓			
LexBFS	*	*	✓	*		✓	
LexDFS	*	*	*	✓	✓		
MCS	*	*	✓	*			✓

List of characterizing properties:

(L1) $l_0 \prec INC(l, i)$ (minimum initial label)

(L2) $l \preceq INC(l, i)$ (monotonic nondecreasing)

(L3) $l \prec INC(l, i)$ (monotonic increasing)

(L4) if $l \prec l'$ then $INC(l, i) \prec INC(l', i)$ (stability)

(L5) $l \prec INC(l', i)$ (depth-firstness)

(L6) if $l \prec l'$ then $INC(l, i) \prec l'$ (breadth-firstness)

(L7) $INC(l, i) = INC(l, i')$ (independence)

for all $i, i' \geq 1$ and $l, l' \in L_i$.

Chapter 5

Implementing Graph Search

In this chapter we consider the question of implementing graph search algorithms.[1] The existing major search paradigms have many well-known implementations, most of which are optimal in time and space complexity. We concentrate on implementing the new search paradigms, such as LexDFS, where efficient implementation is less obvious, and on the implementation of MLS, as a generalization of many search algorithms. We obtain bounds on the complexity of MLS in both general and restricted cases.

5.1 Implementation and complexity of MLS

We begin by considering the question of implementing Algorithm MLS. We observe the following fact directly from the algorithm.

Lemma 5.1.1. *Let S be a labelling scheme. In any execution of S-MLS on a graph G, there are n selections of a vertex whose label is maximal and m updates of vertex labels.*

MLS takes two input parameters. We will assume that an input graph is provided in the adjacency list format, with the neighbour lists provided in arbitrary order. A labelling scheme implies the existence of two important procedures: computing $INC(l, i)$

[1]Some of the results of this chapter are reported in [BKS].

and comparing two labels under the order $\preceq$. It is conceivable that these procedures require significant time to compute, a fact that must be considered when considering an implementation of S-MLS. The labels themselves may also require significant space to store, which should also be considered.

Suppose $S = (L, \preceq, l_0, INC)$ is a labelling scheme. Let $t_{comp}(n)$ denote the worst case time required to compare two legal labels under $\preceq$, let $t_{INC}(n)$ denote the worst case time required to compute $INC(l, i)$ for a legal label $l \in L_{i-1}$, and let $s(n)$ denote the space required to store a legal label, for an input graph on n vertices.

5.1.1 A straight-forward implementation

We now consider a straight-forward implementation of S-MLS.

Implementation A. Let l be an array that holds n labels, one for each vertex. To find a maximal element, execute the following procedure:

1. Choose an arbitrary unnumbered vertex v and assign to variable x.

2. For each unnumbered vertex $w \neq v$, if $l(x) \prec l(w)$, assign w to variable x.

3. Return x as a vertex having maximal label.

This simple implementation yields the following bounds.

Lemma 5.1.2. *Suppose $S = (L, \preceq, l_0, INC)$ is a labelling scheme. Then S-MLS can be implemented in $O(n^2\, t_{comp}(n) + m\, t_{INC}(n))$ time and $O(n\, s(n))$ additional space.*

Proof. Suppose S, G and a suitable implementation of the comparison and INC functions are given. Using Implementation A of MLS, a label update for vertex w requires finding $l(w)$, computing $INC(l(w), i)$ and storing the new label, which requires $O(t_{INC}(n))$ time. Finding a maximal label in Implementation A requires $O(n)$ label comparisons. Thus by Lemma 5.1.1, such an implementation requires $O(n^2\, t_{comp}(n) + m\, t_{INC}(n))$ time. Since only the labels are stored, one per vertex, $O(n\, s(n))$ space is required in addition to the input graph G. $\square$

5.1.2 Using efficient priority queues

We can view graph search as interacting with a priority queue data structure whose domain of elements is the set of vertices of the given graph. A *priority queue* P is an abstract data type that, given an unordered domain of elements and a totally ordered domain of keys, supports a number of operations, including the following:

- $Insert(e, k)$ — insert an element e with key k into P
- $ExtractMax()$ — return and remove from P an element with a maximum key
- $UpdateKey(e, k)$ — update to k the key associated with element e in P

The key associated with each element in P can be considered its priority.

For our purposes, the elements we store in the priority queue will be the unvisited vertices of the graph, and the priority key for each such vertex v will be its label $l(v)$. We note that a priority queue requires a total order of the label set: an arbitrary linear extension of our partial order $\preceq$ suffices.

We note that by Lemma 4.1.4 there is a finite upper bound on the number of labels that can be used which depends only on the size of the input graph. This fact can be exploited to devise an efficient priority queue. Van Emde Boas [vEB77, vEBKZ77] presents a priority queue data structure using stratified trees that can be implemented in $O(u)$ space and $O(\log\log u)$ time per operation, where u is the number of possible key values. Willard [Wil83] later improved the space complexity to $O(k)$ using y-fast tries, where k is the number of elements stored in the priority queue.

With such a priority queue implementation in mind, we present another implementation of S-MLS.

Implementation B. Let $\leq$ be an arbitrary linear extension of $\preceq$. Let P be a priority queue that can store up to n elements with not necessarily distinct keys (the labels of the unvisited vertices) selected from a universe of size $u = |L_n|$. Let $l(v)$ denote the key (label) associated with element (vertex) v in P. To initialize, use $P.Insert(v, l_0)$. To update a vertex's label, use $P.UpdateKey(v, INC(l(v), i))$. To find a vertex with maximal label, use $P.ExtractMax()$.

Lemma 5.1.3. *Suppose $S = (L, \preceq, l_0, INC)$ is a labelling scheme. Then S-MLS can be implemented in $O((n+m)(\log n)\, t_{comp}(n) + m\, t_{INC}(n))$ time using $O(n\, s(n))$ additional space.*

Proof. Suppose S, G and a suitable implementation of the comparison and INC functions are given. We consider Implementation B of MLS using van Emde Boas priority queues. We let $u = |L_n|$. Then $u \leq 2^n$ by Lemma 4.1.4, so each $UpdateKey$ and $ExtractMax$ operation of the queue requires $O(\log\log u) = O(\log n)$ label comparisons.

Computing $INC(l(v), i)$ and updating $l(v)$ in the data structure P requires $O(t_{INC}(n) + \log n \cdot t_{comp}(n))$ time. To find a vertex of maximal label requires a single call to $ExtractMax$, which requires $O(\log n \cdot t_{comp}(n))$ time. By Lemma 5.1.1, total time is $O(n \log n \cdot t_{comp}(n) + m(t_{INC}(n) + \log n \cdot t_{comp}(n)))$. Since no more than n items are stored in the data structure, the van Emde Boas priority queue can be implemented using $O(n\, s(n))$ space. $\square$

In the previous proof, we rely only on the bound on the number of legal labels from Lemma 4.1.4. Lemma 4.3.3 allows us to extend this technique to obtain identical time and space bounds for MLS$^+$.

Corollary 5.1.4. *Suppose $S = (L, \preceq, L_0, INC)$ is an extended labelling scheme. Then S-MLS$^+$ can be implemented in $O((n+m)(\log n)\, t_{comp}(n) + m\, t_{INC}(n))$ time using $O(n\, s(n))$ additional space.*

One notes that the time bounds obtained for MLS by the two implementations are incomparable. Depending on the edge density of G, either implementation may yield the better time bound.

For many interesting graph search algorithms, label comparisons and computing INC require constant time, and input graphs are small enough so that a label can be stored within a constant number of machine words. In such a case, we obtain the following bound.

Corollary 5.1.5. *Suppose $S = (L, \preceq, l_0, INC)$ is a labelling scheme such that, given a graph on n vertices, two labels can be compared in constant time, $INC(l, i)$ can be computed in constant time, and a legal label can be stored in constant space. Then S-MLS can be implemented in $O(\min\{(n + m)\log n, n^2\})$ time and $O(n)$ additional space.*

5.1.3 Optimal implementations for restricted schemes

If a labelling scheme S respects additional properties, we can sometimes exploit them to devise a more efficient implementation of S-MLS. We demonstrate this by presenting a partition refinement implementation for some special cases of S-MLS. This implementation is similar to the work of Habib et al. [HMPV00].

Implementation C. (partition refinement)

Let $\leq$ be an arbitrary linear extension of $\preceq$. The unnumbered vertices are partitioned into classes, where each class consists of all vertices with a given label. Each class C_l is stored as a list of the vertices with label l. To store the partition, we maintain a doubly-linked ordered list P of the pairs (l, C_l) for each non-empty class C_l. Initially, all vertices are in class C_{l_0} and P consists of only of the pair (l_0, C_{l_0}). To choose a vertex to number, an arbitrary vertex v in the first class in P is chosen and removed from that class (if this class becomes empty, it is removed from P); suppose v is assigned number i. Then, for each class C_l such that $l \neq INC(l, i)$, a new class $C_{INC(l,i)}$ is created and each vertex in $C_l \cap N(v)$ is moved to this new class. The pair $(INC(l, i), C_{INC(l,i)})$ is placed either immediately before or immediately after the pair (l, C_l) in P, depending on whether $l \geq INC(l,i)$ or $l < INC(l,i)$ respectively. If either C_l or $C_{INC(l,i)}$ is now empty, the empty class is deleted from P.

We now define two additional properties of labelling schemes, and show that if S satisfies these properties, then Implementation C yields a valid implementation of S-MLS.

Property 5.1.6. *Let $S = (L, \preceq, l_0, INC)$ be a labelling scheme. We define the following properties for S:*

(L8): total ordering: $l \preceq l'$ or $l \succ l'$ for every $i \geq 1$ and $l, l' \in L_i$.

(L9): refinement: if $l \prec l'$, then

- *$l \prec INC(l', i)$,*
- *$INC(l,i) \prec l'$, and*
- *$INC(l,i) \prec INC(l', i)$*

for all $i \geq 1$ and $l, l' \in L_i$.

Lemma 5.1.7. *Suppose S is a labelling scheme that satisfies both the total ordering (L8) and the refinement (L9) properties. Then Implementation C correctly implements S-MLS.*

Proof. To show correctness, we must show that any vertex in the first class in P has maximal label. For notation, let $C_l < C_{l'}$ mean that the class C_l appears earlier in P than $C_{l'}$. We show by induction on i that if $l_i(v) \succ l_i(w)$, then $C_{l_i(v)} < C_{l_i(w)}$ in P. If $i = 1$, $l_i(v) = l_0$ for all vertices v, so the statement is trivially true. Suppose the statement is true for some i with $1 \leq i < n$. Suppose $l_{i+1}(v) \succ l_{i+1}(w)$. Since $\preceq$ is a total order, there are three cases for the relation of $l_i(v)$ and $l_i(w)$ under $\preceq$. If $l_i(v) \succ l_i(w)$, then by hypothesis $C_{l_i(v)} < C_{l_i(w)}$. Since $C_{l_{i+1}(v)}$ is the same as or placed adjacent to

$C_{l_i(v)}$ in P, and likewise for w, the relative order is maintained, hence $C_{l_{i+1}(v)} < C_{l_{i+1}(w)}$ in P. If $l_i(v) = l_i(w) = l$, then C_l contained both v and w at the previous step, and C_l was split into two subclasses at this step, with v and w put in different subclasses. These two subclasses are ordered in P with the class containing v first, hence $C_{l_{i+1}(v)} < C_{l_{i+1}(w)}$. Finally, if $l_i(v) \prec l_i(w)$, then (L9) implies that $l_{i+1}(v) \prec l_{i+1}(w)$, which is not the case. This completes the induction. □

Theorem 5.1.8. *Suppose $S = (L, \preceq, l_0, INC)$ is a labelling scheme such that S satisfies both the total ordering (L8) and the refinement (L9) properties. Then S-MLS can be implemented in $O(n + m(1 + t_{comp}(n) + t_{INC}(n)))$ time using $O(n\, s(n))$ additional space.*

Proof. Suppose S, G and a suitable implementation of the comparison and INC functions are given. Since S satisfies (L8) and (L9), we consider Implementation C of MLS, which correctly implements S-MLS by Lemma 5.1.7.

Initialization of P requires $O(n)$ time to create the class C_{l_0}. Choosing each vertex of maximal label requires constant time. For the update step, each edge causes at most one vertex to move to a different set (the first time one of its endpoints is numbered, the other endpoint may receive a different label and get moved), so at most m vertex moves, new class creations, or empty class deletions are performed, summed over all iterations. Each class creation requires one evaluation of the INC function and one label comparison. Thus the total time required is $O(n + m(1 + t_{comp}(n) + t_{INC}(n)))$.

Each class needs only to store one label for the entire class, and there are at most $O(n)$ classes in P at any time. Since there are at most n vertices stored in P, the total space required is $O(n\, s(n))$. □

In practice, computing label comparisons, computing INC, and storing a label are inexpensive endeavors. In many cases we can consider label comparisons and computing INC to take constant time, and a label can be stored in a constant number of machine words. Under such an assumption, we have obtained an optimal implementation, as simply reading the input graph itself requires $\Omega(n+m)$ time and space.

Corollary 5.1.9. *Suppose $S = (L, \preceq, l_0, INC)$ is a labelling scheme such that S satisfies both the total ordering (L8) and the refinement (L9) properties, and, given a graph on n vertices, two labels can be compared in constant time, $INC(l, i)$ can be computed in constant time, and a legal label can be stored in constant space. Then S-MLS can be implemented in optimal $O(n+m)$ time and $O(n)$ additional space.*

5.1.4 Label-free implementations for restricted schemes

In some cases, an S-MLS implementation need not even compute or store the labels being assigned to vertices. We call such an implementation label-free.

Definition 5.1.10. *An implementation I of algorithm S-MLS is said to be label-free if, for every graph G, in no execution of I on G is the comparison function ever evaluated, the INC function ever called, or any label ever stored.*

Implementation C computes and compares labels only to determine how to order the subclasses that resulted from splitting a single class. If S is monotonic, we know a priori which way to order the subclasses, making storing or manipulating labels unnecessary. A suitable modification to Implementation C when S is monotonic increasing or monotonic decreasing yields a label-free implementation.

Corollary 5.1.11. *Suppose S is a labelling scheme that satisfies the properties*

(L8): $l \preceq l'$ or $l \succ l'$, for every $i \geq 1$ and $l, l' \in L_n$ (total ordering),

(L9): if $l \prec l'$, then $l \prec INC(l', i)$, $INC(l, i) \prec l'$, and $INC(l, i) \prec INC(l', i)$, for all $i \geq 1$ and $l, l' \in L_{i-1}$ (refinement),

and either

(L3): $l \prec INC(l, i)$, for all $i \geq 1$ and $l \in L_{i-1}$ (monotonic increasing) or

(L11): $l \succ INC(l, i)$, for all $i \geq 1$ and $l \in L_{i-1}$ (monotonic decreasing).

Then S-MLS can be implemented in optimal $O(n + m)$ time and $O(n)$ additional space.

5.2 Label-free implementations for graph search

We now turn our attention to some specific graph search algorithms. By the results of Chapter 4, we can implement each of the major search paradigms using one of the MLS implementations discussed in the previous section. In many cases, one can obtain simpler, more intuitive, or more efficient implementations; indeed, many simple implementations are well-known for the basic graph searches.

The two new search paradigms introduced in Chapter 3, MNS and LexDFS, are not presently well-known, so we devote the rest of this section to their implementation.

5.2.1 Maximal Neighbourhood Search

Both LexBFS and MCS are restrictions of MNS. Since an implementation of MNS needs only to find a single valid ordering, a valid implementation could simply run a LexBFS or MCS algorithm on the input graph and return the ordering it computed. Both LexBFS and MCS have well-known, linear time and space (and thus optimal) implementations, and any is sufficient to serve as a MNS implementation.

5.2.2 Lexicographic Depth-First Search

We now consider implementation of the Lexicographic Depth-First Search (LexDFS) algorithm. Given the LexDFS algorithm, as introduced in Chapter 3, we attempt to construct an efficient implementation that will find a valid LexDFS ordering of any input graph. At this point we wish to remind the reader of the LexDFS algorithm details by referring to Algorithm 8 on page 31 of Chapter 3.

By Proposition 4.2.11, we can compute a LexDFS ordering using an implementation of S_{LexDFS}-MLS. By Lemmas 5.1.2 and 5.1.3, S_{LexDFS}-MLS can be implemented in $O(\min\{(n+m)\log n, n^2\}\cdot t_{comp}(n)+m\,t_{INC}(n))$ time and $O(n\,s(n))$ additional space given a graph with n vertices. As this is a labelling implementation, in which the labels for vertices are stored explicitly as the algorithm is being executed, our implementation time bounds are dependent on label operations and storage costs.

A natural question is to ask whether we can improve these bounds, perhaps via a label-free implementation. We devote the remainder of this section to addressing this question.

We make the following observation regarding Algorithm 8.

Lemma 5.2.1. *Let G be a graph. Then in any execution of Algorithm 8, there are n choices of a vertex of lexicographically maximum label and m updates of vertex labels at line 6.*

To analyze the complexity of this algorithm, we assume that the input graph is presented in adjacency list format, and that a label manipulation (or its equivalent, when speaking of a label-free implementation) can be completed in one "step." Then Algorithm 8 needs $\Theta(n)$ steps to initialize vertex labels in line 1, requires a total of $\Theta(n)$ time over all iterations of the main loop to number the chosen vertices in line 4, and will execute the body of the loop on line 5 a total of $\Theta(m)$ times over all iterations of the main loop.

The issue becomes how to perform line 6 efficiently, while still being able to execute line 3 in constant time for each vertex. We will present two alternatives.

First we establish a general framework for our implementations of LexDFS. Since our goal is a label-free implementation, we will neither compute nor store the vertex labels. Instead we keep the unnumbered vertices in lexicographic order, using sets to collect vertices whose (now implicit) labels are equal. This implementation is similar to that presented for LexBFS in [Gol80]. Our data structures are also quite similar to those

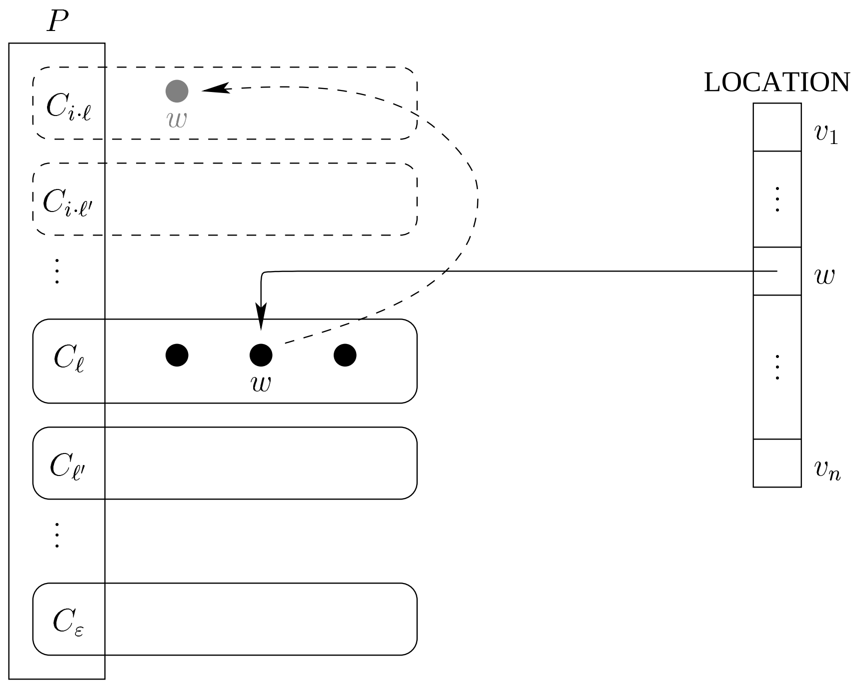


Figure 5.1: Illustration of splitting a set C_ℓ. A vertex w, being a neighbour of the vertex numbered at iteration i and whose label is updated in the LexDFS algorithm, is moved from C_ℓ to a new set $C_{i\cdot\ell}$. The auxiliary array, LOCATION, that points to the present location of w is also shown.

used for the label-free version of Implementation C earlier in this chapter, though the execution is quite different.

General implementation. The main data structure we will use is a doubly-linked ordered list P, which will contain sets of vertices. Each vertex set, C_ℓ, will consist of all the unnumbered vertices with label ℓ:

$$C_\ell = \{v \in V : l(v) = \ell \text{ and } \sigma^{-1}(v) \text{ is undefined}\}.$$

Each of these sets of vertices, C_ℓ, is implemented as a doubly-linked list, plus an indicator flag. We will maintain that every set C_ℓ appearing in our list P contains at least one vertex; if any set C_ℓ becomes empty during execution it will be deleted from P. To facilitate constant time lookup of where a vertex is located in our data structure, we also maintain an auxiliary array of pointers, LOCATION, indexed by vertex. To execute each of the lines of Algorithm 8, we do the following:

- Initially (line 1), all vertices are contained in C_ε and P contains only C_ε.
- To select a vertex of lexicographically maximum label, we select as v in line 3 any vertex in the first set in P (call it C_ℓ) and delete v from that set C_ℓ. If C_ℓ has now become empty, we delete C_ℓ from P.
- To update the label of the unnumbered neighbours w of v (that is, lines 5–6), we do the following (these neighbours w may be processed in arbitrary order):
  - For each set C_ℓ containing an unnumbered vertex $w \in N(v)$, we create a new set $C_{i \cdot \ell}$.
  - If w is an unnumbered neighbour of v in C_ℓ, we delete w from C_ℓ and add w to $C_{i \cdot \ell}$, updating the pointer for w in the LOCATION array.
  - If any set C_ℓ becomes empty (i.e., all vertices in C_ℓ were also in $N(v)$), that set C_ℓ is deleted from P.
  - Once all unnumbered neighbours of v have been processed, the new sets $C_{i \cdot \ell}$ are inserted at the front of P in the same relative order as the original C_ℓ sets (that is, $C_{i \cdot \ell}$ is placed earlier in P than $C_{i \cdot \ell'}$ only if C_ℓ is earlier in P than $C_{\ell'}$).

  Figure 5.1 shows this operation pictorially.

The new sets $C_{i \cdot \ell}$ must be placed at the front of P since each has label lexicographically greater than any previously assigned labels. Furthermore, the order of these new sets depends solely on the previous labels and their relative order.

We note that the splitting step and creation of a new $C_{i \cdot \ell}$ set can be performed "on demand" when the first neighbour vertex w in C_ℓ is encountered. We also note that, by performing the set emptiness test and deleting an empty C_ℓ set when removing a neighbour vertex w from C_ℓ, the set creation and deletion costs can be assigned to processing the edge vw in our time analysis.

We now present two approaches for ensuring these new sets are placed in the correct position in P, and compute their running times.

Proposition 5.2.2. *LexDFS Algorithm 8 is computable in $O(n^2)$ time.*

Proof. We present the "*split & collect*" technique to show this time bound.

Suppose we are processing $w \in N(v)$, and suppose $w \in C_\ell$. If C_ℓ has not yet been split (on this iteration), we create $C_{i \cdot \ell}$ and insert it in P immediately before C_ℓ (if C_ℓ has already been split on this iteration, then $C_{i \cdot \ell}$ has already been created). We set a flag on $C_{i \cdot \ell}$ to indicate it is a new set not yet placed in its proper location. This is the *split* step.

Once the entire neighbourhood of v has been processed, a post-processing *collect* step must be performed to correctly order the new sets and clean up any sets which have become empty. We traverse P starting from the end (bottom), moving any new set $C_{i \cdot \ell}$ to the beginning of P and resetting the new flag. Simultaneously we delete any set C_i we encounter which is empty.

Correctness: Since the relative order of the new sets is maintained by the *collect* step and the new sets are placed at the beginning of P, the sets are properly ordered in P by decreasing implicit label.

Time: Each edge causes one vertex to move to a new set (the first time one of its endpoints is numbered, the other endpoint is moved). Thus at most m vertex moves, set creations, or empty set deletions are performed aggregated over all iterations. Each such operation is constant time.

As there are at most n vertices, there are at most n distinct nonempty sets at any time. There can be no more than n empty sets, as each original set can only be split once in an iteration. Thus the scan through P examines $O(n)$ elements on each of n iterations.

Thus the time required for this technique is $O(m+n^2)$, totaling $O(n+m+n^2) = O(n^2)$ time for the whole algorithm. □

Proposition 5.2.3. *LexDFS Algorithm 8 is computable in $O(n + m \log n)$ time.*

Proof. We present the "*sort & split*" technique to show this time bound.

We first specify the data structures we use in this implementation. We implement the ordered list P in an array of m elements. Each element $P[j]$, $1 \leq j \leq m$, can store a predecessor pointer, a successor pointer, and a data pointer, which points to a list of vertices (i.e., the vertices in the class C_ℓ). The lists of vertices are implemented as a doubly-linked list, with each vertex also having a pointer back to the class's place in the array. We will maintain the ordering of classes stored in P by placing classes with lexicographically larger labels in larger array indexes.

Only a few entries in P are used at any particular time, possibly with gaps; the predecessor and successor pointers allow finding the previous and next entries in P efficiently. We also use an auxilary *top* pointer that points to the first element (i.e., with largest index) that is stored in the array.

Initially, our partition consists of one class C_ε, so *top* points to $P[1]$ in the array, which is set with null predecessor and successor pointers, and whose data pointer points to the list of vertices in C_ε.

To split a class C_ℓ based on the neighbourhood of a vertex v, we move the vertices in $C_\ell \cap N(v)$ to a new list $C_{i\cdot\ell}$, and insert this class at the front of P.

To insert a class $C_{i\cdot\ell}$ in (the front of) P, suppose *top* points to $P[j]$. Since we know $i \cdot \ell$ will be lexicographically larger than any other class currently in P, we will store $C_{i\cdot\ell}$ in a index larger than any other class. We set $P[j+1]$ to point to the list of vertices of $C_{i\cdot\ell}$, set the predecessor of $P[j]$ to point to $P[j+1]$, set the successor pointer of $P[j+1]$ to point to $P[j]$, set the predecessor pointer of $P[j+1]$ to null, and set *top* to point to $P[j+1]$ (i.e., we store $C_{i\cdot\ell}$ in $P[j+1]$ and insert it at the front of the list).

To delete a class C_ℓ, suppose the class is stored in $P[j]$. We simply update the pointers of the predecessor and successor elements to point to each other (i.e., remove C_ℓ from the list). If *top* points to C_ℓ, it is updated to point to the successor; if C_ℓ is last in P, its predecessor's successor pointer is set to null.

We note that we can determine the relative order of two classes simply by examining their indexes in the array.

We now decribe how to perform the update step of the algorithm. Suppose vertex v has just been numbered as $\sigma(i)$, and let $N'(v)$ be the set of vertices in the neighbourhood of v that have not yet been numbered. For $w \in N'(v)$, let $f(w)$ be the index of the class containing w in the array P. We first sort the vertices of $N'(v)$ in increasing order of their $f(w)$ values; thus the vertices of $N'(v)$ are sorted in increasing order of their implicit labels. This is the *sort* step. We then process each vertex in $N'(v)$ in this order.

Suppose a vertex $w \in C_\ell$ is the next vertex to be processed. If C_ℓ has not yet been split this iteration, we create a new set $C_{i\cdot\ell}$ and insert it at the top of P (if C_ℓ has been split, we use the already created $C_{i\cdot\ell}$). We delete w from C_ℓ and insert w in $C_{i\cdot\ell}$. If C_ℓ becomes empty, delete C_ℓ. This is the *split* step. There is no need for a clean up step.

Correctness: As i is greater than any labels assigned in previous iterations, $C_{i\cdot\ell}$ should be earlier in P than any sets with labels not including i. As all new sets created in an iteration contain i, the relative order of new sets depends solely on the relative order of the old labels ℓ. Since the vertices are processed from smallest label to largest due to the sort, always inserting the new sets at the top of P maintains this relative order.

Time: Suppose d_i is the degree of the i-th numbered vertex, v_i. The time for the sorting step is bounded by $c \cdot d_i \log d_i$ for some constant c using standard comparison-based sorting algorithms. The total time spent sorting is thus bounded by

$$\sum_{i=1}^{n} c \cdot d_i \log d_i < c \cdot \left(\sum_{i=1}^{n} d_i \right) \log n = 2c \cdot m \log n \in O(m \log n)$$

since $\log d_i < \log n$ and $\sum_{i=1}^{n} d_i = 2m$.

Each edge causes one vertex to move to a new set (the first time one of its endpoints is numbered, the other endpoint is moved). Thus at most m vertex moves, set creations, or empty set deletions are performed aggregated over all iterations. Each such operation takes constant time.

Thus the total time required for this technique is $O(n + m \log n + m) = O(n + m \log n)$. □

We combine these two results into the following theorem about LexDFS.

Theorem 5.2.4. *Let G be a graph. Then LexDFS can be implemented in $O(\min\{n^2, n + m \log n\})$ time.*

In some restricted cases, we can improve this bound. In the case that the degree of a chosen vertex is bounded by some constant, the *sort* step sorts a bounded number of elements each iteration. Such a sort can be computed in constant time, hence the "sort & split" implementation runs in $O(n + m)$ time. Similarly, if we can bound the number of distinct labels at any point in the algorithm by a constant, scanning P takes constant time each iteration, reducing the complexity of our "split & collect" implementation to $O(n + m)$ time. It is an open question to characterize a class of graphs with such a property.

In general, we know of no implementation of LexDFS that runs in linear time on all graphs. One may note that a fundamental obstacle is efficiently maintaining order in a list. Dietz and Sleator [DS87] provide an efficient solution to the Order-Maintenance Problem, which supports insertion, deletion, and finding the relative order of elements in a list. Unfortunately, we must also find the first element in our ordered list, an operation that is not supported by their data structure. If their data structure (or that of Bender et al. [BCD$^+$02]) could be extended to efficiently support this operation, we may be able to improve our LexDFS time bounds. It remains an open problem to devise a linear time LexDFS implementation, or show that no such implementation can exist.

Chapter 6

Graph Search and Derived Graphs

All searches give a certain amount of insight into the structure of a graph, including the insight gained from our four-vertex characterizations or the MLS conditions. At times we wish to gain such insight into some graph arising from another graph via some function, such as a graph's complement or some power of a graph. Often, we want to obtain insight into the structure of such a derived graph while avoiding the possibly large cost of computing the function.

Graph search is often a convenient and efficient tool to accomplish this. For example, it may be that a LexBFS ordering of the complement of some graph is desired. In some cases, such as when G has $O(n)$ edges, computing the complement of a graph, which in this case would have $O(n^2)$ edges, takes more time than running a LexBFS algorithm. One would want a better approach. It turns out that such an ordering can be found without even computing the complement, given a slight modification of the standard LexBFS algorithm [HMPV00]. This leads directly to an efficient method for recognizing cochordal graphs (those graphs whose complements are chordal).

In this chapter we turn our attention to such problems of computing a search ordering of some derived graph more efficiently than the brute force method of computing the derived graph and running the search algorithm.[1] We exemplify two types of derived graphs, the complements and powers of graphs, and consider when we can find particular search orderings of the derived graphs without explicitly constructing the derived graph. We emphasize that in this chapter n and m denote the number of vertices and number of edges, respectively, of the original graph and not of the derived graph.

6.1 Complements of graphs

We previously touched on the question of computing search orderings of complements of graphs in Section 4.2.2, where we mentioned that some non-graph search labelling schemes yielded graph search orderings of the complement of a graph. It turns out that labelling schemes and the MLS algorithm achieve our goal of computing an ordering without computing the graph's complement since only the edges of the graph G are used to update labels.

We present labelling schemes for computing LexBFS and LexDFS orderings of complements below.

<u>LexBFS of $\overline{G}$</u> (labelling scheme $S_{\overline{LexBFS}}$): L is the set of strings over the alphabet $\{1, 2, \ldots\}$, $\preceq$ is the reverse of the order used for S_{LexBFS}, l_0 is the empty string ε, and $INC(l, i)$ is digit i appended to string l.

<u>LexDFS of $\overline{G}$</u> (labelling scheme $S_{\overline{LexDFS}}$): L is the set of strings over the alphabet $\{1, 2, \ldots\}$; $\preceq$ is $\geq_L$, the reverse of the usual lexicographic order; l_0 is the empty string ε; and $INC(l, i)$ is digit i prepended to string l.

Before we prove that the labelling schemes $S_{\overline{LexBFS}}$ and $S_{\overline{LexDFS}}$ correctly compute LexBFS and LexDFS orderings of the complements of the input graphs, and before we characterize the labelling algorithms that find such orderings, we first require a few more facts and properties regarding labelling schemes.

---

[1]Some of the results of Section 6.2 appear in [CK]. The results of Section 6.1 appear in [BKS].

Property 6.1.1. *Let $S = (L, \preceq, l_0, INC)$ be a labelling scheme. We define the following properties for S:*

(L10): monotonic nonincreasing: $l \succeq INC(l, i)$, for all $i \geq 1$ and $l \in L_i$.

(L11): monotonic decreasing: $l \succ INC(l, i)$, for all $i \geq 1$ and $l \in L_i$.

Lemma 6.1.2. *Let G be a graph and let S be a labelling scheme. Then in any execution of S-MLS on G resulting in ordering σ, the following statements hold:*

i. if S is monotonic nonincreasing, then $label_i(v) \succeq label_j(v)$ for all $1 \leq i < j$ and $v \in V$.

ii. if S is monotonic nonincreasing and stable, and if $label_i(a) \prec label_i(b)$ for some $v_i \leq a, b$, then there exists an integer $j < i$ such that $v_j a \in E$ and $v_j b \notin E$.

Proof. For (i), the statement holds by transitivity and (L10). For (ii), the proof is very similar to that of Lemma 4.2.6(ii). □

We are now ready to prove our characterization of labelling schemes that allow MLS to find a LexBFS ordering of the complement of a graph. We prove the two halves of the characterization separately to contrast with later results.

Lemma 6.1.3. *Let G be a graph and let σ be a vertex ordering that is computed by S-MLS(G), for some labelling scheme S that satisfies the properties*

(L11): $l \succ INC(l, i)$ (monotonic decreasing),[2] *and*

(L12): if $l \prec l'$ then $l \prec INC(l', i)$ [3]

for all $i \geq 1$ and legal labels $l, l' \in L_i$. Then σ is a LexBFS ordering of $\overline{G}$.

[2]This property subsumes (L10).

[3]These two properties imply (L4) holds.

Proof. Suppose σ is an ordering found by S-MLS for some labelling scheme S satisfying the properties above. If σ is not a LexBFS ordering of $\overline{G}$, then, by Theorem 2.3.1, (LB) is violated in $\overline{G}$. So there exists vertices $a < b < c$ with $ac \notin E$, $ab \in E$ such that for all $d < a$, $db \notin E$ implies $dc \notin E$; choose such a triple with a leftmost. Then $label_a(b) = label_a(c)$ by Lemma 4.1.2(iv) and $label_{a+1}(b) \prec label_a(c) = label_{a+1}(c)$ by (L11). Since (L11) subsumes (L10), $label_b(b) \preceq label_{a+1}(b)$ by Lemma 6.1.2(i). By induction using (L12), $label_{a+1}(b) \prec label_b(c)$. Thus $label_b(b) \prec label_b(c)$, contradicting the choice of b by MLS. □

We note that $S_{\overline{LexBFS}} = (\{\text{strings over } \{1, 2, \dots\}\}, \text{ inverted } \geq_L, \varepsilon, INC)$ where $INC(l, i) = l \circ i$, as described above, satisfies (L11) and (L12).

Lemma 6.1.4. *Let G be a graph and let σ be a LexBFS ordering of $\overline{G}$. Then σ can be computed by $S_{\overline{LexBFS}}$-MLS(G).*

Proof. Suppose $\sigma = (v_1, \dots, v_n)$ is a LexBFS ordering of $\overline{G}$. Suppose $v_1, \dots, v_{i-1}$ can be found by $S_{\overline{LexBFS}}$-MLS, but $v_1, \dots, v_i$ cannot, for some $1 < i \leq n$. Then $label_i(v_i)$ is not maximal. Let $w > v_i$ be an unnumbered vertex of maximal label at iteration i with $label_i(v_i) \prec label_i(w)$. By Lemma 6.1.2(ii) there is a $j < i$ with $v_j v_i \in E$ and $v_j w \notin E$; pick j smallest. By Theorem 2.3.1, (LB) holds on the triple $v_j < v_i < w$ in $\overline{G}$, so there is a $k < j$ such that $v_k v_i \notin E$ and $v_k w \in E$; pick k smallest. Then $label_k(v_i) = label_k(w)$ by Lemma 4.1.2(iv), and $label_{k+1}(v_i) \succ INC(label_k(w), k) = label_{k+1}(w)$ by (L11). By induction using (L12), $label_i(v_i) \succ label_{k+1}(w) \succeq label_i(w)$ by Lemma 6.1.2(i), contradicting our choice of w. □

Proposition 6.1.5. *Let G be a graph and let σ be an ordering of V. Then σ is a LexBFS ordering of $\overline{G}$ if and only if there is a labelling scheme S such that σ can be computed by S-MLS(G), where S satisfies the properties*

(L11): $l \succ INC(l, i)$ (monotonic decreasing), and

(L12): if $l \prec l'$ then $l \prec INC(l', i)$

for all $i \geq 1$ and legal labels $l, l' \in L_i$.

Proof. By Lemmas 6.1.3 and 6.1.4, and that $S_{\overline{LexBFS}}$ satisfies (L11) and (L12). □

As any labelling scheme S that satisfies those conditions will allow S-MLS to compute a LexBFS ordering of the complement of a graph, we know that S-MLS has also computed a BFS ordering and a search ordering. So not only do we have an algorithm to compute LexBFS orderings of $\overline{G}$ without computing the complement, we also have an algorithm that does the same for MNS, BFS and generic graph search orderings.

We now use the same approach to characterize which labelling schemes allow MLS to find a LexDFS ordering (and hence a DFS ordering) of the complement of a graph.

Lemma 6.1.6. *Let G be a graph and let σ be a vertex ordering that can be computed by S-MLS(G) for some labelling scheme S that satisfies the properties*

(L13): $l \succ INC(l', i)$,[4] and

(L4): if $l \prec l'$ then $INC(l, i) \prec INC(l', i)$ (stability)

for all $i \geq 1$ and legal labels $l, l' \in L_i$. Then σ is a LexDFS ordering of $\overline{G}$.

Proof. Suppose σ is an ordering found by S-MLS for some labelling scheme S satisfying the properties above. If σ is not a LexDFS ordering of $\overline{G}$, then, by Theorem 3.4.1, (LD) is violated in $\overline{G}$. So there exists vertices $a < b < c$ with $ac \notin E$, $ab \in E$ such that for all d, $a < d < b$, $dc \in E$ implies $db \in E$. By (L13), $label_{a+1}(c) = label_a(c) \succ label_{a+1}(b)$. By induction on (L4), $label_b(c) \succ label_b(b)$, contradicting the choice of b by the algorithm. □

We note that $S_{\overline{LexDFS}} = (\{\text{strings over } \{1, 2, \ldots\}\}, \geq_L, \varepsilon, INC)$ where $INC(l, i) = i \circ l$, as described above, satisfies (L13) and (L4).

[4] This property subsumes (L10) and (L11).

Lemma 6.1.7. *Let G be a graph and let σ be a LexDFS ordering of $\overline{G}$. Then σ can be computed by $S_{\overline{LexDFS}}$-MLS(G).*

Proof. Suppose $\sigma = (v_1, \ldots, v_n)$ is a LexDFS ordering of $\overline{G}$. Suppose $v_1, \ldots, v_{i-1}$ can be found by $S_{\overline{LexDFS}}$-MLS, but $v_1, \ldots, v_i$ cannot, for some $1 < i \leq n$. Then $label_i(v_i)$ is not maximal. Let $w > v_i$ be an unnumbered vertex of maximal label at iteration i with $label_i(v_i) \prec label_i(w)$. Then by Lemma 6.1.2(ii) there is some (rightmost) $j < i$ such that $v_jv_i \in E$ and $v_jw \notin E$ and for all k, $j < k < i$, $v_kv_i \in E$ iff $v_kw \in E$. By Theorem 3.4.1, (LD) holds for triple $v_j < v_i < w$ in $\overline{G}$, so there is a vertex d, $v_j < d < v_i$ with $dv_i \notin E$ but $dw \in E$, a contradiction. □

Proposition 6.1.8. *Let G be a graph and let σ be an ordering of V. Then σ is a LexDFS ordering of $\overline{G}$ if and only if there is a labelling scheme S such that σ can be computed by S-MLS(G), where S satisfies the properties*

(L13): $l \succ INC(l', i)$, and

(L4): if $l \prec l'$ then $INC(l, i) \prec INC(l', i)$ (stability)

for all $i \geq 1$ and legal labels $l, l' \in L_i$.

Proof. By Lemmas 6.1.6 and 6.1.7, and that $S_{\overline{LexDFS}}$ satisfies (L13) and (L4). □

Implementing MLS with the labelling schemes $S_{\overline{LexBFS}}$ and $S_{\overline{LexDFS}}$ allows us to find LexBFS, MNS, BFS, generic search, and LexDFS and DFS orderings of the complement of G in the same asymptotic time as finding LexBFS or LexDFS orderings of G. From Chapter 5, we have a linear (in G) implementation of $S_{\overline{LexBFS}}$-MLS, hence we have a time-optimal algorithm for finding LexBFS, MNS, BFS, and generic search orderings of $\overline{G}$, a fact that was of course previously known [HMPV00].

Our approach using $S_{\overline{LexDFS}}$ is not quite so successful. Though our approach still avoids computing complements, $S_{\overline{LexDFS}}$-MLS does not appear to immediately admit a linear time implementation. By the results of Chapter 5 and parallel to the LexBFS

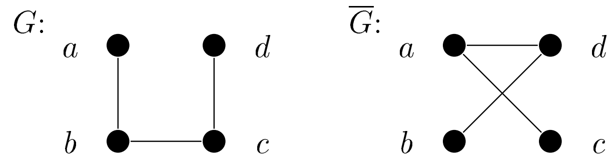


Figure 6.1: A graph and its complement. Ordering *acbd* is not a search ordering of the complement.

case, we do, however, obtain a LexDFS ordering of $\overline{G}$ in many cases more efficiently than computing $\overline{G}$. We note that for DFS, a linear (in G) algorithm is known for finding a DFS ordering of $\overline{G}$ using a different technique. Dahlhaus, Gustedt and McConnell [DGM02] use a technique they call complement stacks to find a DFS ordering of the complement of a graph, and show by amortized analysis that the procedure runs in time linear in G.

After observing the results for LexBFS and LexDFS, one may naturally ask whether a similar approach would work for other, simpler searches. Can we just take a labelling scheme that computes any search and always just reverse the partial order to obtain a corresponding search of the complement of a graph? This approach works for LexBFS and LexDFS because of their stability, but, unfortunately, other searches may not have this property. We can see how this approach fails for generic search. Suppose we take the labelling scheme example for generic graph search and modify it so that $0 \succ 1$. Suppose the MLS algorithm with this scheme is applied to the graph G in Figure 6.1. Vertex a could be chosen first, followed by vertex c (whose label is currently 0 and maximal). Both b and d now have label 1, so b could be chosen first followed by d. But *acbd* is not a search ordering of $\overline{G}$ as $a < b < d$ violates (S) in $\overline{G}$. It turns out that we cannot fully capture graph search of complements with any labelling scheme.

Theorem 6.1.9. *Let S be a labelling scheme such that for every graph G, every generic search ordering of $\overline{G}$ can be found by S-MLS(G). Then there exists a graph $G = (V, E)$ and an ordering σ of V such that σ can be found by S-MLS(G) but σ is not a search ordering of $\overline{G}$.*

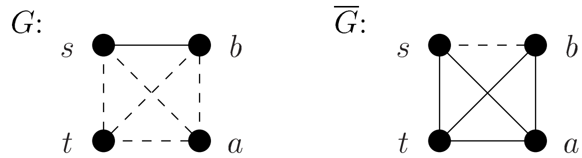


Figure 6.2: A graph for which MLS cannot find exactly the search orderings of the complement. Dashed lines indicate nonedges.

Proof. Let $G = (\{s, t, a, b\}, \{sb\})$ be the graph on four vertices and one edge sb, as shown in Figure 6.2. Let $l_1 = INC(l_0, 1)$. Consider ordering $\sigma = stba$. Since σ is a graph search of $\overline{G}$, σ can be found by S-MLS(G). By Lemma 4.1.2(i), $label_2(t) = l_0 \not\prec label_2(b) = INC(l_0, 1) = l_1$ and $label_3(b) = l_1 \not\prec label_3(a) = l_0$. We now consider ordering $\tau = sbta$. Since $l_1 \not\prec l_0$ and $l_0 \not\prec l_1$, ordering τ may be chosen by S-MLS(G). But τ is not a graph search ordering of $\overline{G}$ since $s < b < t$ violates (S) in $\overline{G}$. $\square$

Regardless of the labelling scheme S one may devise, there are graphs upon which either some search ordering of $\overline{G}$ cannot be found by S-MLS, or S-MLS may compute an ordering that is not a search ordering of $\overline{G}$. Thus no labelling scheme conditions can characterize graph search orderings of the complement.

6.2 Powers of graphs

In this section, we consider one aspect of the problem of computing some type of search ordering of a power of a graph without resorting to computing that power. We ask whether a search ordering of the original graph is a search ordering of the power. We will show that our search characterizations are invaluable in obtaining straightforward proofs of both positive and negative results for this question.

We start with the simple question of whether a BFS ordering of a graph remains a BFS ordering for all powers of a graph.

Theorem 6.2.1. *For any graph G, every BFS ordering of G is a BFS ordering of G^k, for all $k \geq 1$.*

Proof. Let σ be a BFS ordering of G. The theorem is trivially satisfied when $k = 1$. For $k > 1$, we assume for induction that σ is a BFS ordering for all powers of G strictly less than k and consider σ on G^k. Suppose $a < b < c$ is a triple in σ on G^k with $ac \in E^k$ and $ab \notin E^k$.

If (a, b, c) is such a triple in σ on G^{k-1}, then by (B) and our induction hypothesis, there exists a vertex $d < a$ with $db \in E^{k-1} \subseteq E^k$, hence satisfying (B) on G^k.

Suppose instead that (a, b, c) is not such a triple in σ on G^{k-1}. Since $E^{k-1} \subseteq E^k$, $ab \notin E^{k-1}$. Thus $ac \notin E^{k-1}$. But then there must be a vertex e with $ae \in E^{k-1}$ and $ec \in E$. Since $ab \notin E^k$ but $ae \in E^{k-1}$, we must have $eb \notin E$. We have 3 cases for the position of e in σ relative to a and b.

Case 1: $e < a$. In σ on G, since $ec \in E$ and $eb \notin E$, we apply (B) to the triple $e < b < c$, giving us a vertex $d < e < a$ such that $db \in E \subseteq E^k$, satisfying (B) on G^k.

Case 2: $a < e < b$. We first consider σ on G. We know that $ec \in E$ and $eb \notin E$, so by applying (B) to the triple $e < b < c$, we get a vertex $d < e$ such that $db \in E$. If $d < a$, we're done; otherwise, we have $a < d$. Since $db \in E$ but $ab \notin E^k$, we must have $ad \notin E^{k-1}$. We apply (B) and our inductive hypothesis to the triple $a < d < e$ in σ on G^{k-1}, giving us a vertex $d_2 < a$ such that $d_2d \in E^{k-1}$. Since $d_2d \in E^{k-1}$ and $db \in E$, d_2 and b are of distance at most k in G. Thus $d_2b \in E^k$, satisfying (B) on G^k.

Case 3: $b < e$. In σ on G^{k-1}, since $ae \in E^{k-1}$ and $ab \notin E^{k-1}$, we apply (B) and our inductive hypothesis to the triple $a < b < e$, giving us a vertex $d < a$ with $db \in E^{k-1} \subseteq E^k$, satisfying (B) on G^k.

In each case, we show the existence of a vertex $d < a$ with $db \in E^k$, hence (B) is satisfied by σ on G^k. Thus σ is a BFS ordering of G^k. $\square$

It is clear that a generic search ordering of G is also a generic search ordering of G^k: if a vertex becomes eligible to be chosen in a search of G, it will also be eligible in a search

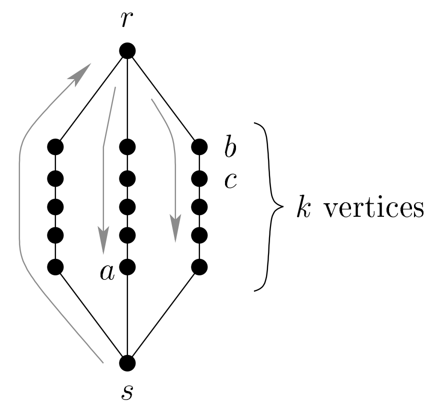


Figure 6.3: Given k, a graph G where no DFS from s is a DFS of G^k. The possible DFS path through G is shown in grey.

of G^k. Whereas both BFS and generic search orderings survive through the powers of a graph, one naturally wonders about DFS. We now show that a DFS ordering of G is not necessarily a DFS ordering in a given power of G. Again, the characterizing property of DFS from Chapter 3 will play an crucial role in our proof.

Theorem 6.2.2. *For every $k \geq 2$, there exists a graph G such that no DFS ordering of G is a DFS ordering of G^k.*

Proof. Fix $k \geq 2$. We first show the existence of a graph G such that no DFS ordering starting from a distinguished start vertex s is a DFS ordering of G^k.

Let G be the graph consisting of two vertices s and r and three s,r-paths of length $k+1$. This graph is shown in Figure 6.3.

Let σ be an arbitrary DFS ordering of G starting from s. The DFS will proceed from s along one of the three paths to reach r. From r, the DFS will follow a second path towards s to reach a adjacent to s. As s is already discovered, the DFS will recurse back to r, and travel the third path towards s. Let b be the first vertex visited on this third path, and let c be the next vertex on the path.

Thus, in σ, we have $s < r < a < b < c$. Moreover, b appears in σ immediately following a (when a is reached, all neighbours of a are already numbered, so DFS recurses to r, where b must be the next vertex to be numbered).

Now consider σ in G^k. One easily verifies that a and c are of distance k in G, and a and b are of distance $k+1$ in G. Hence $ac \in E^k$ and $ab \notin E^k$. But since a and b are next to each other in σ, there can be no vertex between a and b. Therefore, the triple $a < b < c$ violates the condition (D). Hence σ cannot be a DFS ordering of G^k.

To prove the theorem, we consider two copies of G joined at s by an edge: call this graph G'. Consider any DFS ordering σ' of G': it may start in either copy of G, possibly at s itself. We observe that if σ' starts in one copy of G, σ' restricted to the other copy of G appears as a DFS ordering of G starting at s. Following the argument above, we can identify three vertices $a < b < c$ in the second copy of G.

We note that no vertex from the other copy of G is within distance k of b, thus there can be no edge ab in $(G')^k$. The triple $a < b < c$ violates (D) as above, so σ' is not a DFS of $(G')^k$. Since σ' was chosen arbitrarily, no DFS ordering of G' can be a DFS ordering of $(G')^k$. $\square$

Brandstädt, Dragan and Nicolai [BDN97] show that the reverse of a LexBFS ordering of a chordal graph G is a common perfect elimination ordering of all odd powers of G. One may be tempted to think that a LexBFS ordering of a chordal graph is a LexBFS ordering of the odd powers, but Corneil and Krueger [CK] show this is not the case.

Theorem 6.2.3 ([CK]). *For each $k \geq 2$, there exists a chordal graph G such that no LexBFS ordering of G is a LexBFS ordering of G^k.*

The constructions used in the proof of this theorem are illustrated in Figure 6.4. We note that the construction shown in Figure 6.4(b) for the odd powers yields a graph that is strongly chordal. A graph is *strongly chordal* if it is chordal and every cycle $C = (x_1, \ldots, x_{2k})$ of even length has a chord $x_i x_j$ where the distance between the endpoints x_i and x_j in C is odd. An immediate corollary of Theorem 6.2.3 is that for each $k \geq 1$, there exists a strongly chordal graph G such that no LexBFS ordering of G is a LexBFS

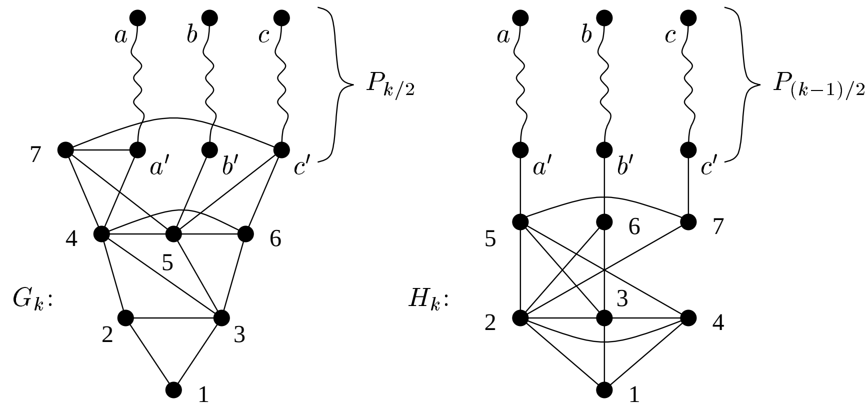


Figure 6.4: Constructions for (a) k even and (b) k odd, where no LexBFS ordering starting from 1 is a LexBFS ordering of G^k.

ordering of G^{2k+1}. We observe that, despite the negative result of Theorem 6.2.3, the result of Brandstädt, Dragan and Nicolai [BDN97] coupled with Theorem 2.4.3 (Shier [Shi84]) leads directly to the following fact.

Corollary 6.2.4. *Every LexBFS ordering of a chordal graph G is an MEC (equivalently, MCC) ordering of G^{2k+1} for all $k \geq 1$.*

We conclude this section by identifying a class of graphs where LexBFS does persist through squares. A bipartite graph is *chordal bipartite* if every cycle of length at least 6 has a chord.

Theorem 6.2.5 ([CK]). *Let G be a chordal bipartite graph. If σ is a LexBFS ordering of G, then σ is a LexBFS ordering of the graph G^2.*

We note that this theorem cannot be strengthened to bipartite graphs, as shown by the graph in Figure 6.5. Call this graph B. The ordering $\sigma = (a, b, c, d, e, f, g)$ is a LexBFS ordering of B; however, the triple of vertices $d < e < f$ violates property (LB) in B^2. We have $df \in E^2$ and $de \notin E^2$, but every vertex in σ left of d that is adjacent to e in B^2 is also adjacent to f in B^2. Attaching a second copy of B via an edge at a ensures no LexBFS ordering of the graph is a LexBFS ordering of its square.

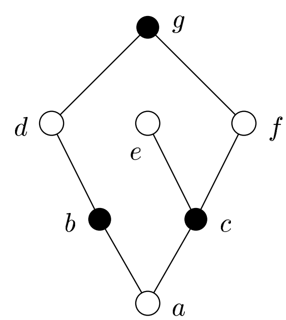


Figure 6.5: A bipartite graph with a LexBFS ordering that is not a LexBFS ordering of the square.

Chapter 7

Deriving Search-Based Algorithms

The four-vertex characterizations of the various graph searches are a tool not only for giving us a better understanding of the differences and similarities between searches, but also for exploiting the power of graph search for application to interesting problems. We have seen in the previous chapter how these characterizations can lead to simple proofs of search properties.

In this chapter, we build on this theme and devise algorithms related to graph search to solve deeper problems.[1] We recall from the graph search hierarchy in Chapter 3 that maximal neighbourhood search is a generalization of both LexBFS and MCS, two searches that have been widely used. We use MNS to better understand how these searches reveal vertex separation properties, and show how extensions of MNS can also exploit these properties to find elimination orderings and minimal triangulations of arbitrary graphs.

7.1 Finding minimal triangulations

To initiate our discussion of minimal triangulations, we step back and consider the fundamental concepts of chordal graphs. Also known as triangulated graphs, *chordal graphs*

[1]Many of the results of this chapter are reported in [BKS05].

have several structural properties that allow us to efficiently solve many problems on these graphs that are hard in general. Chordal graphs have come to be an extremely important class in structural graph theory [BLS99].

There are many different characterizations of chordal graphs. A graph G is chordal if and only if each cycle in G of length at least 4 has a chord. Dirac [Dir61] observed that a graph is chordal if and only if every minimal separator is a clique. This characterization can be extended by the fact that minimal separators are contained within the neighbourhood of a vertex [LB62].

Another useful characterization of chordal graphs captures their linear structure and uses the fact that every chordal graph contains at least two simplicial vertices [Dir61]. Since removing any vertex (and all its incident edges) from a chordal graph results in a chordal graph, we can play a game where we iteratively find a simplicial vertex and remove it. This is known as the Elimination Game, and if you succeed in removing all vertices from a graph G, we know G is chordal. Fulkerson and Gross [FG65] showed the other direction too, that you can only succeed if the original graph is chordal.

The order in which vertices are removed in the Elimination Game yields a vertex ordering of the graph. As each vertex is simplicial in the remaining graph, this ordering is called a *simplicial ordering* or a *perfect elimination ordering* (peo). We recall the definition of a peo from Chapter 2.

Definition 7.1.1. *Let $G = (V, E)$ be a graph and $\sigma = (v_1, \ldots, v_n)$ be an ordering of V. We say σ is a perfect elimination ordering (peo) if v_i is simplicial in $G[v_i, \ldots, v_n]$ for all i, $1 \leq i \leq n$.*

A graph is chordal if and only if it has a peo [FG65]. The Elimination Game is one way to compute a peo; more efficient algorithms exist, such as LexBFS [RTL76] and MCS [TY84].

When a graph G fails to be chordal, it is often useful to embed G into some chordal graph H. We can do this simply by adding enough edges to G such that we create a chordal supergraph H (this is always possible, since the complete graph is chordal). We call such a graph H a *triangulation* of G, and write $H = (V, E \cup F)$ where $E \cap F = \emptyset$. We call the set F of edges added to G the set of *fill edges* or the *fill-in* of the graph G.

Many different triangulations of a graph exist in general. We are usually interested in "small" triangulations where we add as few edges as necessary. The *minimum triangulation* problem (also known as the minimum fill-in problem) asks to find a triangulation of G with the fewest number of fill edges. This problem has application to sparse matrix computations [Ros72], relational database management [TY84], knowledge based systems and computer vision [Heg04]. Not surprisingly, the minimum triangulation problem is NP-hard in general [Yan81], so one is interested in finding a *minimal triangulation*, where the fill-in is inclusion-minimal. A triangulation $H = (V, E \cup F)$ is said to be minimal if $(V, E \cup F')$ is non-chordal for every proper subset F' of F. There are several methods for finding a minimal triangulation in polynomial time [Heg04]. One method is to find and saturate the minimal separators in a graph, thus satisfying Dirac's characterization. The approach we are interested in is generalizing the peo approach.

The Elimination Game simply iteratively selects and removes a simplicial vertex, that is, a vertex whose neighbourhood is a clique. We can do a similar operation on an arbitrary graph $G = (V, E)$: simply pick a vertex to remove and add fill edges to make its neighbourhood a clique. Suppose F is the set of edges that are added. The result is an elimination ordering of G, where this ordering forms a peo of the graph $H = (V, E \cup F)$. Thus H is a triangulation of G.

Since the ordering in which the vertices are selected does not impact whether the graph H is a triangulation, we can choose this ordering arbitrarily and provide it as input to this elimination process. Suppose $\alpha = (v_1, \ldots, v_n)$ is an ordering of V. At each step in the elimination process, choose the next unchosen vertex in α, add fill edges to make its neighbourhood in the remaining graph a clique, and remove the vertex from the graph. We formalize this idea in the following definitions.

Definition 7.1.2. *Let $G = (V, E)$ be a graph. For a vertex $v \in V$, we define the deficiency of v in G, $D_G(v)$, as the set of edges that must be added to make v simplicial in G. That is,*

$$D_G(v) = \{xy : vx, vy \in E, xy \notin E, x \neq y\}.$$

Given an ordering α of V, we define the fill-in $F = F(G, \alpha)$ as the set of edges that must be added to G for α to be a perfect elimination ordering of $(V, E \cup F)$. We formally state this below.

Definition 7.1.3. *Let $G = (V, E)$ be a graph and let $\alpha = (v_1, \ldots, v_n)$ be an ordering of V. Let $G_0 = G$. For $1 \leq i \leq n$, we let $F_i = D_{G_{i-1}}(v_i)$ and obtain G_i from G_{i-1} by adding the edges in F_i and removing vertex v_i. We define the fill-in of G for ordering α as*

$$F = \bigcup_{i=1}^{n} F_i$$

and $G^+_\alpha = (V, E \cup F)$.

If G^+_α is a minimal triangulation of G for some vertex ordering α, we say α is a *minimal elimination ordering* (meo) of G.

Rose, Tarjan and Lueker [RTL76] showed the following property of minimal triangulations, upon which many minimal triangulation algorithms are based.

Theorem 7.1.4 ([RTL76]). *Let $G = (V, E)$ be a graph and let $H = (V, E \cup F)$ be triangulated. Then H is a minimal triangulation if and only if, for each $f \in F$, $H - f = (V, E \cup F - \{f\})$ is not triangulated.*

The following characterization of minimal triangulations is fundamental to the proofs of correctness of several minimal triangulation algorithms, including our own.

Theorem 7.1.5 ([RTL76]). *Let $G = (V, E)$ be a graph and $H = (V, E \cup F)$ be triangulated. Then H is a minimal triangulation of G if and only if each $f \in F$ is a unique chord of a 4-cycle in H.*

One of the more popular (and efficient) algorithms for computing minimal triangulations is due to Rose, Tarjan and Lueker [RTL76]. They showed that an extension of LexBFS they call Lex M could efficiently compute an meo α and its associated minimal triangulation G^+_α.

Lex M works similarly to LexBFS, assigning labels to vertices, iteratively selecting a vertex with lexicographically maximum label, and appending a digit to the labels of nearby vertices. The distinction is that Lex M uses the vertex labels to deduce where the fill edges would be added, and appends appropriate labels to both real neighbours (in the original graph) and would-be neighbours had we included those fill edges. Lex M basically mimics a LexBFS of the filled graph G^+_α. For completeness, we present Lex M as Algorithm 12.

Algorithm 12: LexBFS for computing an meo (Lex M) [RTL76]

input : a graph $G = (V, E)$

output: an ordering σ of V

```
1 assign the label ε to all vertices
2 for i ← 1 to n do
3     pick an unnumbered vertex v with lexicographically maximum label
4     α(i) ← v      //This assigns to v the number i
5     foreach unnumbered vertex w such that there exists a path
              [v = v_1, v_2, ..., v_k, v_{k+1} = w] with v_j unnumbered and
              label(v_j) < label(w) for 2 ≤ j ≤ k do
6         append n − i to label(w)
```

Let us term a path that Lex M finds at line 5 a *fill path*. Every nontrivial fill path corresponds to a fill edge of G^+_α.

As an example of how Lex M works, consider the graph in Figure 7.1. Call this graph G. Initially all vertices have label ε, so s may be chosen first. Each of its neighbours,

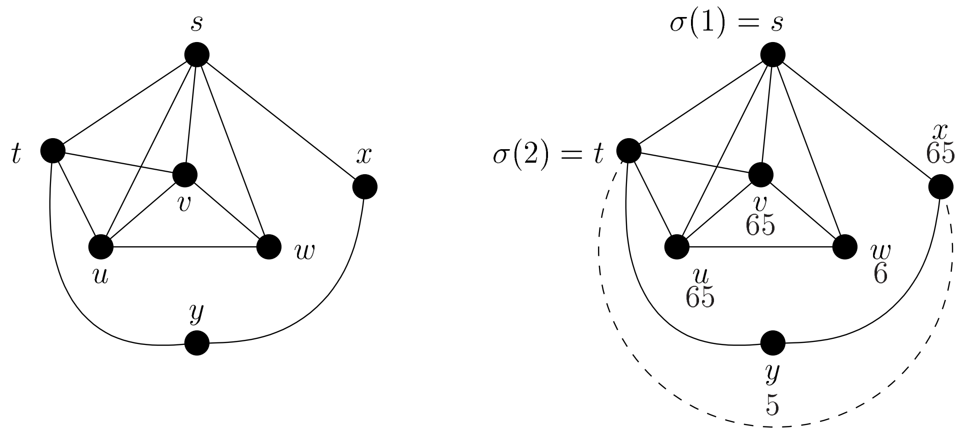


Figure 7.1: (a) A graph used to illustrate Lex M. (b) The state after the second step of Lex M. The numbers in the figure are the labels assigned to each unnumbered vertex, and the dashed line is the fill edge added for the fill path $[t, y, x]$.

t, u, v, w, x, have digit 6 appended to their labels. Since t has maximum label, we may choose t next. Each of t's unnumbered neighbours, u, v, y have digit 5 appended. In addition, $[t, y, x]$ is a fill path, so x also gets digit 5 appended to its label. Notice that there is no fill path that ends at w, so the label of w remains unchanged. The algorithm would continue, possibly selecting ordering $\alpha = stuvxwy$. Notice that this is a LexBFS ordering of the graph $G^+_\alpha = (V, E \cup \{tx\})$.

Recently Berry, Blair, Heggernes and Peyton [BBHP04] extended MCS to MCS-M in the same way as Lex M extends LexBFS, and showed that MCS-M also finds meos. Similar to how MCS and LexBFS generally yield different orderings of a graph, MCS-M and Lex M yield different meos of a graph. Villanger [Vil04] recently showed that both algorithms produce the same set of minimal triangulations, even though the meos that generate these triangulations may be different.

Observing that MNS generalizes both LexBFS and MCS, our question is whether MNS can be extended in a similar way to obtain minimal triangulations. There is no reason a priori that such a generalization would subsume similar generalizations of peo-computing searches such as LexDFS, so a second question that arises is whether LexDFS can be extended as well. In the remainder of this section we answer these questions in the affirmative, yielding insight into the relationship between such graph search algorithms and minimal elimination orderings.

7.1.1 MNS-M

Our goal is to mimic in some sense the relationship between LexBFS and Lex M to derive an MNS-M algorithm. Fundamental to the Lex M (and MCS-M) algorithm is the concept of a fill path, a path that roughly corresponds to a desired (or needed) fill edge. The following lemma was used by Rose, Tarjan and Lueker to capture this correspondence when arguing correctness of their Lex M algorithm.

Lemma 7.1.6 ([RTL76]). *Let $G = (V, E)$ be a graph and let α be an ordering of V with $G^+_\alpha = (V, E \cup F)$. Then $vw \in E \cup F$ if and only if there exists a chain $[v = v_1, v_2, \ldots, v_k, v_{k+1} = w]$ in G such that $v_j > v$ and $v_j > w$ for $2 \leq j \leq k$.*[2]

We define a reasonable analogue to this statement that is particularly suited to an MNS-type algorithm.

Definition 7.1.7. *An MNS-M **fill path** is a chain $[v = v_1, v_2, \ldots, v_k, v_{k+1} = w]$ such that $v < v_j$ and $label_v(v_j) \subset label_v(w)$ for $2 \leq j \leq k$.*

We use this concept of a fill path to devise our new MNS-M algorithm, and present it as Algorithm 13. The only change we make to the basic MNS algorithm is to add to line 5 our MNS equivalent of the Lemma 7.1.6 property.

We remind the reader that given an execution of MNS-M, we say $label(w)$ is the final label assigned to vertex w. We define $label_i(w) = label(w) \cap \{1, \ldots, i-1\}$ to be the label of w at the point during algorithm execution when $\alpha(i)$ is assigned. For shorthand, we write $label_v(w) = label_{\alpha^{-1}(v)}(w)$.

[2] We note that in the vacuous case when $vw \in E$, the chain is just $[v, w]$.

Algorithm 13: Maximal Neighbourhood Search for computing an meo (MNS-M)

input : a graph $G = (V, E)$

output: an ordering σ of V

assign the label $\emptyset$ to all vertices

for $i \leftarrow 1$ ***to*** n **do**

  pick an unnumbered vertex v with maximal label (neighbourhood)

  $\alpha(i) \leftarrow v$ //This assigns to v the number i

  foreach *unnumbered vertex w such that there exists a fill path $[v = v_1, v_2, \ldots, v_k, v_{k+1} = w]$ with v_j unnumbered and $label(v_j) \subset label(w)$ for $2 \leq j \leq k$* **do**

    add i to $label(w)$

We now proceed to prove that MNS-M correctly computes an meo. We require several lemmas regarding the algorithm and the labels it assigns.

Lemma 7.1.8. *Suppose α is an MNS-M ordering of G. Then $i \in label(w)$ if and only if $\alpha^{-1}(w) > i$ and there exists a fill path $[v = \alpha(i), v_2, \ldots, v_k, w]$ with $label_i(v_j) \subset label_i(w)$ and $\alpha^{-1}(v_j) > i$ for $2 \leq j \leq k$.*

Proof. Follows directly from algorithm 13. □

Our next lemma states that, during the execution of the MNS-M algorithm, once a vertex w is the end of some fill path P, P will always be a fill path until w is numbered. Any label digit that is assigned to an internal vertex on the fill path will also be given to w.

Lemma 7.1.9. *Let G be a graph and let $[v, v_2, \ldots, v_k, w]$ be a fill path during an execution of MNS-M. If $i \in label(v_j)$ for some $i < \alpha^{-1}(w)$ and $2 \leq j \leq k$, then $i \in label(w)$.*

Proof. Let $[v, v_2, \ldots, v_k, w]$ be a fill path at step t. Let v_j be the first vertex internal to the path to receive a label after step t, and let i be this label. Assume $i < \alpha^{-1}(w)$ (that is, w has not yet been chosen). Let $x = \alpha(i)$.

Since v_j receives a label, there must exist a fill path $[x, x_2, \ldots, x_r, v_j]$ with $label(x_l) \subset label(v_j)$ for $2 \leq l \leq r$. Since $label(v_j) \subset label(w)$, we have $label(x_l) \subset label(w)$ for $2 \leq l \leq r$. Hence $[x, x_2, \ldots, x_r, v_j, v_{j+1}, \ldots, v_k, w]$ is also a fill path, hence i is also assigned to the label of w. Thus $[v, v_2, \ldots, v_k, w]$ is still a fill path, and repeated application of this argument establishes the lemma. □

Lemma 7.1.10 (Fill Path Lemma). *Let G be a graph and consider any execution of MNS-M on G. For any fill path $[v, v_2, \ldots, v_k, w]$, we have $label_i(v_j) \subset label_i(w)$ for $\alpha^{-1}(v) \leq i < \alpha^{-1}(w)$, $2 \leq j \leq k$.*

Proof. By Lemma 7.1.9. □

Lemma 7.1.11. *If α is an MNS-M ordering of G, then $label(w) = \{\alpha^{-1}(v) \mid vw \in E \cup F$ and $\alpha^{-1}(v) < \alpha^{-1}(w)\}$.*

Proof. For the first half of the equality, suppose $i = \alpha^{-1}(v) \in label(w)$. Then by Lemma 7.1.8, $\alpha^{-1}(w) > i$ and there is a fill path $[v, v_2, \ldots, v_k, w]$ with $label_v(v_j) \subset label_v(w)$ for $2 \leq j \leq k$. By Lemma 7.1.10, w is chosen before v_j, for $2 \leq j \leq k$. Thus the fill path fulfills Lemma 7.1.6, giving $vw \in E \cup F$.

For the other half of the equality, suppose $vw \in E \cup F$. Assume wlog that $\alpha^{-1}(v) < \alpha^{-1}(w)$. By Lemma 7.1.6, there is a chain $[v, v_2, \ldots, v_k, w]$ with $\alpha^{-1}(v_j) > \alpha^{-1}(w)$. We will proceed by induction on k, the length of the chain.

Base case: Suppose $k = 1$. The chain is merely $[v, w]$ which is trivially a fill path, hence $\alpha^{-1}(v) \in label(w)$.

Induction hypothesis: Assume now for induction that the Lemma is true for chains of length $k \leq k_0$. We will show the Lemma is true for a chain of length $k = k_0 + 1$.

Let v_j be the first vertex chosen in $\{v_2, \ldots, v_k\}$. Then $[v, v_2, \ldots, v_j]$ and $[w, v_k, \ldots, v_j]$ are chains of length less than k, thus $\alpha^{-1}(v), \alpha^{-1}(w) \in label(v_j)$ by our inductive hypothesis. Hence there exist fill paths $[v, x_2, \ldots, v_j]$ and $[w, y_2, \ldots, y_r, v_j]$, with $label_w(y_i) \subset label_w(v_j)$ for $2 \leq i \leq r$.

Since $label_w(w)$ is maximal, $label_w(w) \not\subset label_w(v_j)$. If $label_w(w) = label_w(v_j)$, then $\alpha^{-1}(v) \in label(w)$ and we're done. Suppose then that $label_w(w) \parallel label_w(v_j)$. Then there exist $a \in label_w(v_j) - label_w(w)$ and $b \in label_w(w) - label_w(v_j)$: choose a and b to be the earliest such labels assigned and let $u_a = \alpha(a)$ and $u_b = \alpha(b)$. If $a < b < \alpha^{-1}(w)$, then $label_b(w) \subset label_b(v_j)$. Let y_i be the first vertex along the path $y_2, \ldots, y_{r+1} = v_j$ such that $label_b(w) \subset label_b(y_i)$. Then $[u_b, \ldots, w, \ldots, y_i]$ is a fill path, so $b \in label(y_i)$. Since $label_w(y_i) \subset label_w(v_j)$, we have $b \in label(v_j)$, a contradiction to our choice of b. If, on the other hand, $b < a$, then $label_a(w) \supset label_a(v_j)$, so $label_a(y_i) \subset label_a(w)$ and $[u_a, \ldots, v_j, y_r, \ldots, y_2, w]$ is a fill path, giving $a \in label(w)$, a contradiction. We must conclude that $label_w(w) \nparallel label_w(v_j)$

Thus we have $label_w(v_j) \subset label_w(w)$. Since $\alpha^{-1}(v) \in label(v_j)$ and $\alpha^{-1}(v) < \alpha^{-1}(w)$, then $\alpha^{-1}(v) \in label(w)$, completing the proof. $\square$

Theorem 7.1.12. *Let $G = (V, E)$ be a graph and α be an MNS-M ordering of G that computed the labels $label(v)$ for each $v \in V$. Then any edge $vw \in F = F(G, \alpha)$ is the unique chord of some 4-cycle $\mu = [p, v, q, w]$ in $G_\alpha^+ = (V, E \cup F)$.*

Proof. Let $vw \in F$. Assume wlog that $\alpha^{-1}(v) < \alpha^{-1}(w)$. By Lemma 7.1.11, $\alpha^{-1}(v) \in label(w)$. By Lemma 7.1.8, there exists a fill path $[v, v_2, \ldots, v_k, w]$ with $label_v(v_j) \subset label_v(w)$ for $2 \leq j \leq k$. Let p be the vertex chosen first in $\{v_2, \ldots, v_k\}$. Then $vp, wp \in E \cup F$ by Lemma 7.1.6.

Since $label_v(p) \subset label_v(w)$, there exists a vertex q such that $\alpha^{-1}(q) \in label_v(w) - label_v(p)$. By Lemma 7.1.11, $qw \in E \cup F$. Since α is a perfect elimination ordering of G_α^+ and $vw, qw \in E \cup F$, we have $vq \in E \cup F$.

Now, since $\alpha^{-1}(q) \notin label(p)$ and $\alpha^{-1}(q) < \alpha^{-1}(p)$, Lemma 7.1.11 says that $qp \notin E \cup F$.

Therefore, vw is the unique chord of the 4-cycle $\mu = [p, v, q, w]$. $\square$

We immediately obtain the following.

Theorem 7.1.13. *Let $G = (V, E)$ be a graph with MNS-M ordering α. Then α is a minimal elimination ordering of G.*

Proof. G^+_α is a minimal triangulation by Theorems 7.1.5 and 7.1.12. □

It is interesting to note that every Lex M or MCS-M (or even LexDFS-M, which can be defined analogously) ordering is also an MNS-M ordering. Rose, Tarjan and Lueker [RTL76] show that the Lex M labels consist precisely of the label digits corresponding to those vertices monotonely adjacent (via a fill path), exactly as Lemma 7.1.11 states. Thus Lemma 7.1.10 characterizes the relationship between the labels of vertices on the fill path for both Lex M and MCS-M, providing insight into why both algorithms work. As expected, Theorem 7.1.13 generalizes the results of [RTL76] and [BBHP04].

One may wonder about the relationship of the orderings found by the basic peo-finding graph search algorithms and their meo-finding counterparts. Can, for example, MNS-M find all orderings that MNS can find for any graph? We answer this in the negative. We note that we use the term Lex M instead of LexBFS-M for historical reasons.

Proposition 7.1.14. *There exist graphs such that the following are true:*

1. *There exists an MNS ordering that is not an MNS-M ordering.*

2. *There exists an MNS-M ordering that is not an MNS ordering.*

3. *There exists an MCS ordering that is not an MCS-M ordering.*

4. *There exists an MCS-M ordering that is not an MCS ordering.*

5. *There exists a LexBFS ordering that is not a Lex M ordering.*

6. *There exists a Lex M ordering that is not a LexBFS ordering.*

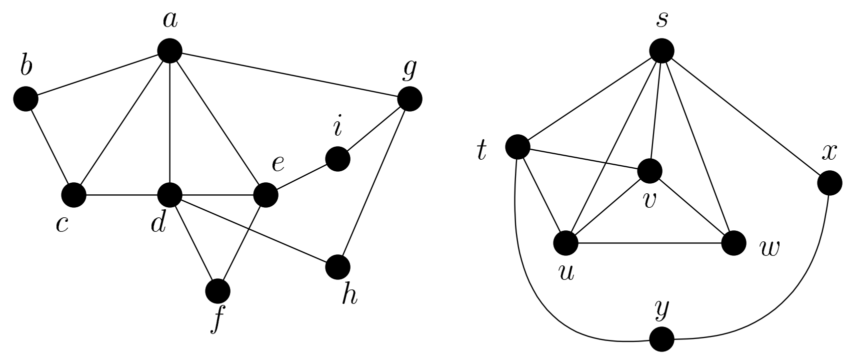


Figure 7.2: Graphs that can distinguish graph search orderings and meos.

Proof. Consider the graphs in Figure 7.2.

The ordering $acdefbghi$ is a valid MCS or MNS ordering, but it is neither an MNS-M ordering, as $label_f(f) = \{3,4\} \subset label_f(g) = \{1,3,4\}$, nor an MCS-M ordering, as $label_f(f) = 2 < label_f(g) = 3$.

The ordering $acdegbfhi$ is a valid MCS-M or MNS-M ordering, but it is neither an MNS nor an MCS ordering, since the triple $e < g < f$ violates properties (M) and (MC).

The ordering $stuvwxy$ is a valid LexBFS ordering, but it is not a Lex M ordering, since $label_w(w) < label_w(x)$.

The ordering $stuvxwy$ is a valid Lex M ordering, but it is not a LexBFS ordering, as the triple $v < x < w$ violates property (LB). □

We observe that MNS-M has the same behaviour on the input graph G as MNS has on G^+_α. If the input graph is chordal, its only minimal triangulation is itself. So when the same question relating MNS-M and MNS orderings is posed regarding chordal graphs, the answer is opposite: if G is chordal, every MNS-M ordering of G is an MNS ordering of G, and vice versa.

7.1.2 MLSM

We now recall the idea of Maximal Label Search (MLS) from Chapter 4. As one extends MNS to MNS-M, one can also extend MLS to MLSM in a similar manner. We present this as Algorithm 14.

Algorithm 14: Maximal Label Search for computing an meo (MLSM)

---

input : a graph $G = (V, E)$ and a labelling scheme $S = (L, \preceq, l_0, INC)$

output: an ordering α of V

assign the label l_0 to all vertices

for $i \leftarrow 1$ *to* n **do**

  pick an unnumbered vertex v whose label is maximal under $\preceq$

  $\alpha(i) \leftarrow v$ //This assigns to v the number i

  foreach *unnumbered vertex* w *such that there exists a chain* $[v = v_1, v_2, \ldots, v_k, v_{k+1} = w]$ *with* v_j *unnumbered and* $label_i(v_j) \prec label_i(w)$ *for* $2 \leq j \leq k$ **do**

    $label(w) \leftarrow INC(label(w), i)$

Berry, Krueger and Simonet [BKS05] investigate MLSM on labelling schemes that are monotonic increasing and stable (i.e., labelling schemes for LexBFS, MCS, LexDFS and MNS). Following the same proof structure as we illustrated used to prove the correctness of MNS in Theorem 7.1.13, the following is shown.

Theorem 7.1.15 ([BKS05]). *Let $G = (V, E)$ be a graph and let S be a labelling scheme that satisfies the properties*

(L3): monotonic increasing: $l \prec INC(l, i)$, for all $i \geq 1$ and legal labels $l \in L_i$, and

(L4): stability: if $l \prec l'$, then $INC(l, i) \prec INC(l', i)$, for all $i \geq 1$ and legal labels $l, l' \in L_i$.

If α is an S-MLSM ordering of G, then α is an meo of G.

As an immediate corollary, Lex M, MCS-M and LexDFS-M all compute an meo of an arbitrary graph.

7.1.3 CompMLS and CompMLSM

It is well known that, in general, there are peos that LexBFS is incapable of finding. The same can be said for MCS. The same result holds for MNS, and thus LexDFS. Figure 7.3 illustrates this difficulty.

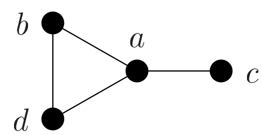


Figure 7.3: The ordering *abcd* is the reverse of a peo, but is not an MNS ordering.

The MEC and MCC algorithms of Shier [Shi84] addresses this issue by choosing vertices that are maximal within their component of the remaining graph instead of maximal in the entire remaining graph. Shier (Theorem 2.4.3 [Shi84]) shows that any peo of a graph can be computed with these algorithms.

This idea can easily be translated into the MLS context. Berry, Krueger and Simonet [BKS05] devise compMLS, a component choosing version of MLS. We present this as Algorithm 15.

The following statements are corollaries of Shier's results. We recall that, by Proposition 4.2.7, the monotonic increasing and stable properties for MLS characterize MNS orderings.

Corollary 7.1.16. *Let $G = (V, E)$ be a graph and let σ be an S-compMLS ordering of G, where S is a labelling scheme that is monotonic increasing and stable. Then σ is a peo of G.*

Algorithm 15: Component-wise Maximal Label Search (compMLS)

input : a graph $G = (V, E)$ and a labelling scheme $S = (L, \preceq, l_0, INC)$

output: an ordering σ of V

```
assign the label l_0 to all vertices
for i ← 1 to n do
    pick a component C of G − {σ(1), ..., σ(i − 1)}
    pick an unnumbered vertex v ∈ C whose label is maximal in C
    σ(i) ← v      //This assigns to v the number i
    foreach unnumbered vertex w adjacent to v do
        label(w) ← INC(label(w), i)
```

Corollary 7.1.17. *Let $G = (V, E)$ be a graph and let S be a labelling scheme that is monotonic increasing and stable. Then any peo σ of G is computable by S-compMLS(G).*

The same extension can be applied to our various meo-finding algorithms. For example, [BKS05] applies this idea to the MLSM algorithm, yielding compMLSM. When proving the correctness of our meo-finding algorithms, we only compare the labels of vertices along a fill path in G. This means the vertices we choose need only be maximal within their connected component. Hence, the following theorem follows easily from earlier results.

Theorem 7.1.18 ([BKS05]). *Consider any meo-finding algorithm Lex M, MCS-M, MNS-M, LexDFS-M or S-MLSM for any labelling scheme S that is monotonic increasing and stable. Let G be a graph. Then any ordering computed by the comp- version of the algorithm is an meo of G.*

It turns out that, for every graph, precisely the same orderings can be computed by the various comp- meo-finding algorithms.

Theorem 7.1.19 ([BKS05]). *Let S, S' be labelling schemes that are monotonic increasing and stable. Then, for every graph G, any S-compMLSM(G) ordering can be computed by S'-compMLSM(G).*

It also turns out that the component-wise approach does not allow us to find additional minimal triangulations.

Theorem 7.1.20 ([BKS05]). *Let S be a labelling scheme that is monotonic increasing and stable. Then every graph has the same S-MLSM and S-compMLSM minimal triangulations.*

These theorems lead to the following interesting result, which generalizes the result of Villanger [Vil04] that Lex M and MCS-M compute the same minimal triangulations.

Theorem 7.1.21 ([BKS05]). *For any graph G, whichever meo-computing algorithm is used (Lex M, MCS-M, MNS-M, LexDFS-M or S-MLSM for any labelling scheme S that is monotonic increasing and stable, or their comp- extensions, compLex M, compMCS-M, compMNS-M, compLexDFS-M or S-compMLSM), the set of minimal triangulations that are computable is the same.*

As a conclusion to this section, we remark that these graph search-based algorithms for finding meos cannot find every meo nor every minimal triangulation of a graph, as shown by Figure 7.4.

It has been shown experimentally [BHT01] that Lex M finds only a small number of the minimal triangulations of graphs, and often yields a triangulation that is far from minimum. Though the results of this chapter go a long way to understanding the relationship between graph search and minimal triangulation, we have failed to enrich the family of minimal triangulations that can be obtained. This appears to be a fundamental limitation of graph search-based algorithms, and many questions related to these issues remain open.

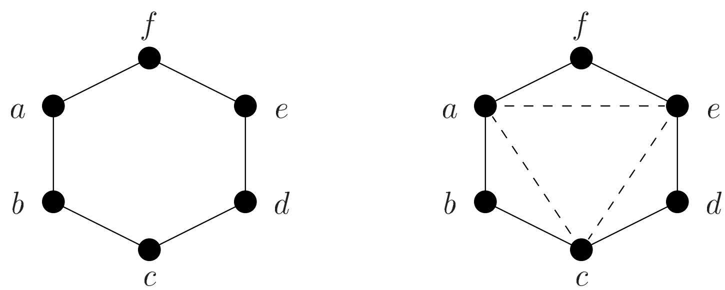


Figure 7.4: The meo $bdface$ is not an MNS-M ordering, and its corresponding minimal triangulation cannot be obtained by any of these algorithms.

Chapter 8

Concluding Remarks

We conclude this thesis with remarks about extensions of graph search to generalizations of undirected graphs. We consider the question of extending the unifying graph search characterizations presented in Chapters 3 and 4 to these generalized settings. We leave the reader with several open questions and directions for future research on these topics.

8.1 Generalizations of undirected graphs

There are several basic generalizations of undirected graphs, such as digraphs, multigraphs and hypergraphs, upon which we can run some sort of graph search algorithm. In some cases, such as for directed graphs, our graph search algorithms, if designed carefully, require little change to be applicable to the new environment and retain many of their useful properties.

8.1.1 Directed graphs

Both BFS and DFS have been widely applied to directed graphs. For example, BFS can be used to compute shortest paths in a directed graph, and DFS can be used to perform a topological sort or identify strongly connected components [Tar72]. We ask the question here whether simple characterizations, like we derived in Chapter 3, can describe search orderings of directed graphs.

In this section, we consider a directed graph $D = (V, A)$. We employ this notation to distinguish directed graphs from undirected graphs. We say $xy \in A$ is a directed edge (or arc) oriented from x to y, with x being the tail and y being the head.

Our standard algorithms for expressing graph search on undirected graphs can easily be translated to the directed case. We present as Algorithm 16 a version of MLS for digraphs.

Algorithm 16: Maximal Label Search (MLS) on directed graphs

---

input : a digraph $D = (V, A)$ and a labelling scheme $S = (L, \preceq, l_0, INC)$

output: an ordering σ of V

assign the label l_0 to all vertices

for $i \leftarrow 1$ *to* n **do**

  pick an unnumbered vertex v whose label is maximal under $\preceq$

  $\sigma(i) \leftarrow v$ //This assigns to v the number i

  foreach *unnumbered vertex* w *with* $vw \in A$ **do**

    $label(w) \leftarrow INC(label(w), i)$

Fundamentally, there is little difference between Algorithm 16 and our original MLS algorithm. Where in the undirected case we considered unnumbered neighbours w of a visited vertex v, in the directed case we follow arcs only to the out-neighbours of v (i.e., following the arcs in the appropriate direction). We observe that this is similar to how the undirected version of MLS worked, in that the edges were only used in the forward direction to update labels of vertices that were to be numbered later.

Suppose we are given a digraph $D = (V, A)$ and a vertex ordering σ. We define an undirected graph $G(D, \sigma)$ companion to D by

$$G(D, \sigma) = (V, E = \{xy : xy \in A \text{ and } x <_\sigma y\}).$$

Thus $G(D, \sigma)$ contains edges corresponding to the left-to-right arcs of D given ordering σ. Any back-arcs or loops in D are discarded by this translation.

Similarly, given an undirected graph $G = (V, E)$ and a vertex ordering σ, one can define a directed graph $D(G, \sigma)$ companion to G by

$$D(G, \sigma) = (V, A = \{xy : xy \in E \text{ and } x <_\sigma y\}).$$

Thus $D(G, \sigma)$ contains arcs corresponding to the edges of G oriented in the left-to-right direction.

These constructions capture the relationship between search on directed graphs and search on undirected graphs.

Theorem 8.1.1. *Let σ be an ordering of vertex set V and let S be a labelling scheme. Then following statements are true:*

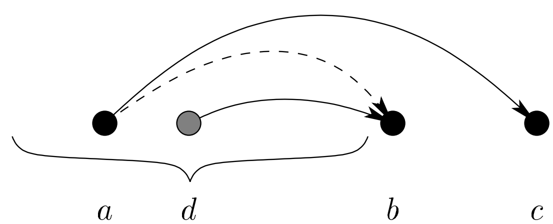
1. *For an undirected graph G, σ is an S-MLS ordering of G if and only if σ is an S-MLS ordering of $D(G, \sigma)$.*

2. *For a directed graph D, σ is an S-MLS ordering of D if and only if σ is an S-MLS ordering of $G(D, \sigma)$.*

Proof. 1. Suppose $G = (V, E)$ is a graph and $\sigma = (v_1, \ldots, v_n)$ is an ordering of V. Let $D = D(G, \sigma)$. Let $label_i^G(v)$ indicate the label of vertex v at the start of iteration i in any execution of S-MLS(G) that computes σ. We will construct a valid execution of S-MLS(D) such that the label of every unnumbered $v \in V$ at iteration i, denoted by $label_i^D(v)$, is equal to $label_i^G(v)$ for $i \geq 1$. In particular, the label of v_i is maximal at iteration i, so v_i may be chosen by S-MLS(D). When $i = 1$, $label_i^D(v) = label_i^G(v) = l_0$ for $v \in V$, so v_1 may be chosen by S-MLS(D). Suppose the statement holds for some $i \geq 1$. For unnumbered $v \in V$, if $label^G_{i+1}(v) \neq label^G_i(v)$, by Lemma 4.1.2(iii) $v_i v \in E$ and $label^G_{i+1}(v) = INC(label^G_i(v), i)$. By the construction of $D(G, \sigma)$, $v_i v \in A$, thus $label^D_{i+1}(v) = INC(label^D_i(v), i) = label^G_{i+1}(v)$. Otherwise suppose $label^G_{i+1}(v) = label^G_i(v)$. If $v_i v \notin E$, then $v_i v \notin A$ by the construction of $D(G, \sigma)$, so $label^D_{i+1}(v) = label^D_i(v)$. If $v_i v \in E$, then $INC(label^G_i(v), i) = label^G_i(v) = label^D_i(v)$, and $label^D_{i+1}(v) = label^D_i(v)$. Thus v_i may be chosen by S-MLS(D). The reverse direction holds via a similar argument.

2. Holds by a similar proof. □

Thus, if σ is an S-MLS ordering of a graph (resp. digraph), σ is an S-MLS ordering of its suitable companion digraph (resp. graph). Since we have a characterization of valid search orderings on the undirected graph from Chapter 3, we can derive characterizations of the valid search orderings of the directed graph.



(S2): For an ordering σ of the vertices of a digraph, if $a < b < c$ and $ac \in A$ and $ab \notin A$, then there exists a vertex $d < b$ such that $db \in A$.

Theorem 8.1.2. *For an arbitrary digraph D, an ordering σ of V is a search ordering of D if and only if σ has property (S2).*

Proof. Let D be a digraph and let σ be an ordering of V. Let $G = G(D, \sigma)$. Let $S = S_{\text{GS}} = (\{0, 1\}, \leq, 0, INC)$ where $INC(l, i) = 1$, as described in Section 4.1.1. From Theorem 8.1.1, σ is an S-MLS(D) ordering iff σ is an S-MLS(G) ordering. By Theorem 4.2.4, σ is an S-MLS(G) ordering iff σ is a graph search ordering of G. By Theorem 3.1.2, σ is a graph search ordering of G iff σ has property (S). Thus, by the construction of G, σ has property (S) iff $a < b < c$ with $ac \in A$ and $ab \notin A$ implies there exists a vertex $d < b$ such that $db \in A$. This is precisely property (S2), completing our proof. $\square$

Similar theorems hold for each of the other major graph search algorithms using the same argument. We can characterize the orderings that each of BFS, DFS, LexBFS, LexDFS, MNS, and MCS find on directed graphs simply by orienting the edges from left to right.

Directed versions of each of these search algorithms can also be created directly, and the characterizations we claim can easily be proven directly.

Though DFS has been applied to several problems on directed graphs, the use of the other graph search paradigms seem to be restricted to the world of undirected graphs. With the vertex ordering characterizations described in this section, it will be interesting to see whether these other searches have useful applications to digraph problems. We leave this as an open question.

8.1.2 Multigraphs and hypergraphs

Depending on how one treats multigraphs and hypergraphs, one may generalize graph search in such a way that its behaviour on these objects is much the same as graph search on undirected graphs. This is generally our approach in the case of directed graphs. However, with other interpretations, this may not be the case.

A multigraph is a generalization of a graph in that parallel edges are allowed between any pair of vertices. The question of characterization of search orderings of multigraphs must be prefaced with the question of what a reasonable concept of search would be on a multigraph. How are parallel edges to be considered? To use MCS as an example, if there are multiple edges between a numbered vertex v and an unnumbered vertex w, should w count its visited neighbourhood as increased by one vertex or should the size of the visited neighbourhood of w be incremented by one for each parallel edge? This issue becomes especially important when considering MLS, as the INC function could be called several times for a particular vertex w and we must ensure the labelling scheme is well defined.

The answers to these questions depend on the application being considered. If multiple edges are to be ignored, our search algorithms essentially treat multigraphs as simple graphs, and our characterization results hold. But if multiple edges permit a vertex label to be updated several times within a single iteration, some searches are fundamentally altered (such as for our MCS example).

We leave the topic of searches on multigraphs at this point, with the open question of characterizing graph-like searches on multigraphs where multiple edges are considered meaningful. It may be that a generalization of graph searching captures these ideas nicely, and may lead to a better understanding of our characterizations.

Applying graph search ideas to hypergraphs leads to the same issues as we raised for multigraphs: how do we deal with the generalized notion of an edge? A hypergraph consists of a set V of vertices and a set of subsets of V. In contrast to graphs, a "hyperedge" may contain an arbitrary number of endpoints. How should a graph search-like algorithm explore such a hyperedge?

Depending on the application, the answers may vary. Perhaps a hyperedge leads to all incident vertices, essentially treating that hyperedge as a representation for a clique of a graph. Perhaps a hyperedge can only be traversed if all but one incident vertex has been numbered. Each of these approaches ignores the information contained in the hyperedge; perhaps a better solution from a labelling perspective would be to somehow share the label update over all incident vertices.

Apart from the trivial extensions of the results of Chapters 3 and 4 when a hypergraph is treated much like a graph, we leave open the question of graph search-like algorithms for hypergraphs and characterizing interesting vertex orderings. As with extensions to multigraphs, a generalization of our characterization ideas to hypergraphs may enhance our understanding of graph search.

8.2 Other orderings from graph search

In this thesis we have concentrated on a particular product of graph search algorithms: the ordering of vertices by when they are first visited. Graph searches in general compute many other orderings which may be useful. The finishing time of a vertex is when its incident edges have been fully traversed by a graph search. Ordering vertices by their finishing times has been shown to give a simple method for computing a topological sort of a directed graph (see, for example, [CLRS01]). Other uses of finishing times and the vertex post-orderings appear to have received little study. It would be interesting to study the properties of such post-orderings and explore whether they have simple characterizing properties.

We can similarly consider discovery and finishing orderings of the edges of a graph. Graph search traverses every edge of a graph, and we wonder whether the edge orderings have related characterizations. It would be interesting to explore the link between edge orderings of a graph and the vertex orderings of its line graph. Line graphs themselves could be studied as a derived graph in a manner similar to our study of complements and powers in Chapter 6.

As we conclude this thesis, we see many avenues for further exploration. The work described in this thesis has led to characterization results relating the various graph search paradigms. Our goal was to create a unified framework by which we could consider graph search, and understand the vertex orderings that these algorithms can compute. In this, we hope that we have succeeded. This work is but a step along the way, illustrating just how much we have yet to learn about this basic tool of computing.

Bibliography

[BBHP04] Anne Berry, Jean R. S. Blair, Pinar Heggernes, and Barry W. Peyton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39(4):287–298, 2004.

[BCD96] Andreas Brandstädt, Victor D. Chepoi, and Feodor F. Dragan. Perfect elimination orderings of chordal powers of graphs. *Discrete Math.*, 158(1-3):273–278, 1996.

[BCD$^+$02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002)*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164, 2002.

[BCHP03] A. Bretscher, D. G. Corneil, M. Habib, and C. Paul. A simple linear time LexBFS cograph recognition algorithm. In *Proceedings of WG'03: 29th International Workshop on Graph-Theoret ic Concepts in Computer Science*, number 2880 in Lecture Notes in Computer Science, pages 119–130, 2003.

[BDN97] Andreas Brandstädt, Feodor F. Dragan, and Falk Nicolai. LexBFS-orderings and powers of chordal graphs. *Discrete Math.*, 171(1-3):27–42, 1997.

[BHT01] Jean R. S. Blair, Pinar Heggernes, and Jan Arne Telle. A practical algorithm for making filled graphs minimal. *Theoretical Computer Science*, 250(1–2):125–141, January 2001.

[BKS] A. Berry, R. Krueger, and G. Simonet. An algorithmic perspective of graph searching. Submitted.

[BKS05] A. Berry, R. Krueger, and G. Simonet. Ultimate generalizations of LexBFS and LEX M. In *Proceeding of 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2005)*, June 2005. to appear.

[BLS99] A. Brandstädt, V.B. Le, and J.P. Spinrad. *Graph classes: a survey*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, PA, 1999.

[Bre05] A. Bretscher. *LexBFS based recognition algorithms for cographs and related families*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 2005.

[CDK03] Derek G. Corneil, Feodor F. Dragan, and Ekkehard Köhler. On the power of BFS to determine a graph's diameter. *Networks*, 42(4):209–222, 2003.

[CHK99] Jou-Ming Chang, Chin-Wen Ho, and Ming-Tat Ko. LexBFS-ordering in asteroidal triple-free graphs. In *ISAAC '99: Proceedings of the 10th International Symposium on Algorithms and Computation*, pages 163–172, London, UK, 1999. Springer-Verlag.

[CIRS03] L. S. Chandran, L. Ibarra, F. Ruskey, and J. Sawada. Generating and characterizing the perfect elimination orderings of a chordal graph. *Theoretical Computer Science*, 307(2):303–317, 2003.

[CK] D.G. Corneil and R. Krueger. A unified view of graph searching. *Submitted.* Expanded abstract will appear as: Simple vertex ordering characterizations for graph search, *ICGT'05: Proceedings of 7th French International Colloquium on Graph Theory*, Hyeres, France, 2005.

[CKN+95] Derek G. Corneil, Hiryoung Kim, Sridhar Natarajan, Stephan Olariu, and Alan P. Sprague. Simple linear time recognition of unit interval graphs. *Information Processing Letters*, 55(2):99–104, 21 July 1995.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[Cor04a] D. G. Corneil. Lexicographic breadth first search — a survey. In *Proceedings of WG2004*, LNCS 3353, pages 1–19, 2004.

[Cor04b] Derek G. Corneil. A simple 3-sweep LBFS algorithm for the recognition of unit interval graphs. *Discrete Appl. Math.*, 138(3):371–379, 2004.

[COS95] Derek G. Corneil, Stephan Olariu, and Lorna Stewart. A linear time algorithm to compute a dominating path in an AT-free graph. *Information Processing Letters*, 54(5):253–257, 9 June 1995.

[COS98] Derek G. Corneil, Stephan Olariu, and Lorna Stewart. The ultimate interval graph recognition algorithm? (extended abstract). In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (San Francisco, CA, 1998)*, pages 175–180, New York, 1998. ACM.

[COS99] Derek G. Corneil, Stephan Olariu, and Lorna Stewart. LBFS orderings and cocomparability graphs. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 883–884. Society for Industrial and Applied Mathematics, 1999.

[DGM02] Elias Dahlhaus, Jens Gustedt, and Ross M. McConnell. Partially complemented representations of digraphs. *Discrete Math. Theor. Comput. Sci.*, 5(1):147–168, 2002.

[Dir61] G.A. Dirac. On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg*, 25:71–76, 1961.

[DS87] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of 9th ACM Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[Duc84] P. Duchet. Classical perfect graphs. *Ann. Discrete Math.*, 21:67–96, 1984.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[Eve79] Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.

[FG65] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific J. Math.*, 15:835–855, 1965.

[Gol80] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

[Han73] G. Y. Handler. Minimax location of a facility in an undirected tree graph. *Transportation Science*, 7:287–293, 1973.

[Heg04] Pinar Heggernes. Minimal triangulations of graphs: a survey. *Disc. Math.*, 2004. submitted.

[HMPV00] Michel Habib, Ross McConnell, Christophe Paul, and Laurent Viennot. Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234(1–2):59–84, March 2000.

[HT73] John E. Hopcroft and Robert Endre Tarjan. ACM Algorithm 447: Efficient algorithms for graph manipulation [H]. *Communications of the ACM*, 16(6):372–378, June 1973.

[LB62] C.G. Lekkerkerker and J.C. Boland. Representation of a finite graph by a set of intervals on the real line. *Fund. Math.*, 51:45–64, 1962.

[Lee61] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961.

[Luc82] E. Lucas. *Récréations Mathématiques.* Paris, 1882.

[Ma] T. Ma. unpublished manuscipt.

[Moo59] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[Ros72] D.J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R.C. Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.

[RTL76] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, June 1976.

[Shi84] D. R. Shier. Some aspects of perfect elimination orderings in chordal graphs. *Discrete Applied Mathematics*, 7:325–331, 1984.

[Sim91] Klaus Simon. A new simple linear algorithm to recognize interval graphs. In Hanspeter Bieri and Hartmut Noltemeier, editors, *Workshop on Computational Geometry*, volume 553 of *Lecture Notes in Computer Science*, pages 289–308. Springer, 1991.

[Tar95] G. Tarry. Le probléme des labyrinthes. *Nouvelles Annales de Math.*, 14:187, 1895.

[Tar72] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[Tar76] Robert Tarjan. Maximum cardinality search and chordal graphs. Unpublished lecture notes for CS 259, Stanford University, 1976.

[TY84] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, August 1984.

[TY85] Robert E. Tarjan and Mihalis Yannakakis. Addendum: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 14(1):254–255, February 1985.

[vEB77] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[vEBKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[Vil04] Y. Villanger. Lex M versus MCS-M. *to appear in a Special Issue of Discrete Mathematics*, 2004.

[Wil83] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.

[Yan81] M. Yannakakis. Computing the minimum fill-in is NP-comptete. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):77–79, 1981.