Cache Scheduling

by

Torrey Luke Hoffman
B.Sc., University of Victoria, 1995

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. V. King, Supervisor (Department of Computer Science)

[redacted]

Dr. J. Ellis, Departmental Member (Department of Computer Science)

[redacted]

Dr. W. Myrvold, Departmental Member (Department of Computer Science)

[redacted]

Dr. S. Irani, External Examiner (Department of Information and Computer Science, University of California, Irvine)

© Torrey Luke Hoffman, 1997
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Supervisor: Dr. Valerie King

Abstract

The Cache Scheduling problem is to find an optimal permutation of processes in a computer system to minimize the number of cache misses. This increases the effectiveness of cache memory and improves system performance. We define this problem, evaluate its complexity, and consider algorithms to solve it. Cache Scheduling generalizes known problems such as Pathwidth and Gate Matrix Layout. We show that a generalization of Pathwidth, which we call Almost Pathwidth, is a special case of Cache Scheduling and is a graph problem that is closed under taking graph minors.

[redacted]

Dr. V. King, Supervisor (Department of Computer Science)

[redacted]

Dr. J. Ellis, Departmental Member (Department of Computer Science)

[redacted]

Dr. W. Myrvold, Departmental Member (Department of Computer Science)

[redacted]

Dr. S. Irani, External Examiner (Department of Information and Computer Science, University of California, Irvine)

Table of Contents

List of Figures

1. Introduction

In this thesis we discuss a problem we call Cache Scheduling. Cache Scheduling is a theoretical combinatorial problem, based on a practical optimization problem that may have many applications.

Cache Scheduling models the problem of finding an optimal ordering of a set of processes in order to maximize the effectiveness of cache memory. This problem arose from research in operating systems – the work presented in this thesis was inspired by a question raised by Dr. Li of Princeton University [30]. However, it is also applicable to compilers, databases, or other systems which involve the interaction of a set of processes and a storage hierarchy consisting of a fast cache memory and a slower main memory.

Few results have been published on this topic. Recent work by Irani, Dillencourt, and Halem [21] considers the theoretical complexity of a special case of Cache Scheduling. Kimbrel considers similar problems in [25], largely from the perspective of practical implementations. Philbin, Edler, Anthus, Douglas, and Li in [37] consider some special cases of Cache Scheduling, again from the perspective of finding efficient implementations. Other than these developments, Cache Scheduling does not appear to have been studied, or even defined as a distinct problem before now.

A primary goal of the research presented here has been to develop effective algorithms for Cache Scheduling. We give approximation algorithms and an algorithm for a fixed parameter variation, and describe some other approaches that have been less successful.

We also consider the theoretical complexity of Cache Scheduling. It is easy to show that special cases of Cache Scheduling are equivalent to known NP-complete problems. However, we are especially interested in other special cases of Cache Scheduling which are *not* equivalent to previously studied problems. We conjecture that all non-trivial cases of Cache Scheduling are NP-complete, except fixed parameter variations of the problem.

We begin in Chapter 2 with a brief summary of some of the relevant definitions and background material in graph theory, algorithms, and complexity. We introduce the Cache Scheduling problem in Chapter 3. Section 3.2 contains the formal definition of Cache Scheduling, and shows how the informal problem of processes and a memory hierarchy can be formally defined as a combinatorial problem on the rows of a matrix. In 3.3, we briefly discuss the Paging Problem, an on-line problem with some similarity to Cache Scheduling.

Recent research by Kimbrel [25] generalizes the Paging Problem using a model that closely corresponds to real systems. We discuss several problems introduced by Kimbrel in Section 3.4, and describe the differences between them and Cache Scheduling.

In Chapter 4 we show that given a fixed ordering of processes, finding an optimal cache allocation is easy, and in Chapter 5, we prove a special case of Cache Scheduling is NP-complete with a reduction from Hamiltonian Path. We also discuss an NP-completeness proof given in [25], and show that despite some similarity, it does not apply to Cache Scheduling.

Chapter 6 shows that the Consecutive Ones, Pathwidth, and Gate Matrix Layout problems are all special cases of Cache Scheduling. We discuss these problems and examine the application of known results for these problems to Cache Scheduling.

In Chapter 7, we give a dynamic programming algorithm for Cache Scheduling. Although it has exponential running time, is a significant improvement over the obvious method.

In Chapter 8, we consider a fixed parameter variation of Cache Scheduling, and show that it is closely related to a generalization of Pathwidth. We show that this generalization, which we call Almost Pathwidth, is closed under taking graph minors. This provides a non-constructive proof that a polynomial time algorithm for the problem exists. We also observe that the corresponding "Almost Treewidth" generalization of Treewidth is not closed under minors.

A different view of the Cache Scheduling problem in Chapter 9 highlights relationships with the Errand Scheduling and Travelling Salesman with Neighborhoods problems. Chapter 10 introduces two simple heuristics for Cache Scheduling.

Chapter 11 describes a variation of the Cache Scheduling problem, and Chapter 12 contains conclusions, a summary of results, and suggested avenues of future research.

2. Preliminaries

We assume the reader has at least an introductory knowledge of graph theory, algorithms, and computational complexity. However, we will begin with an overview of terminology and definitions that are central to this thesis.

For graph theory definitions not given here, see Harary [19]. Background information on algorithms and complexity can be found in Cormen, Leiserson, and Rivest [11] and Papadimitriou [36].

2.1 Graph Theory

A *graph* $G = (V, E)$ consists of a *vertex set* V and an *edge set* E. We will consider only *undirected* graphs, in which each edge is an unordered pair of vertices in V. We denote an edge between vertices v and w as $\{v, w\}$. The vertex set of a graph G is denoted by $V(G)$, and the edge set is denoted by $E(G)$.

A *subgraph* G of a graph H is a graph with $V(G) \subseteq V(H)$ and $E(G) \subseteq E(H)$. If S is a subset of $V(H)$, the subgraph G of H *induced* by S has vertex set $V(G) = S$ and edge set $E(G) = \{(u, v) \mid (u, v) \in E(H)$ and both u and v are in $S\}$.

If two vertices v_1 and v_2 are joined by an edge e, we say v_1 and v_2 are *adjacent*, and that e is *incident* to both v_1 and v_2.

An edge (v, v) is called a *loop*. Except as noted otherwise, we do not allow loops. Note that our definition of graphs precludes multiple edges.

An *edge-weighted graph* is a graph G together with a function ω which assigns to each edge $e \in E(G)$ an *edge weight* $\omega(e) \in \Re$.

Two graphs G_1 and G_2 are *isomorphic* if there is a bijection ϕ from $V(G_1)$ to $V(G_2)$ such that $(u, v) \in E(G_1)$ if and only if $(\phi(u), \phi(v)) \in E(G_2)$.

Given a graph G containing an edge (u, v), we may construct a graph G' by the *edge contraction* of (u, v) in G as follows. We make G' a copy of G with vertices u and v removed, along with all their incident edges. We add a new vertex w to G', and add edges to make w adjacent to each vertex adjacent to either u or v in G'. The following diagram shows the edge contraction of $e = (v_1, v_2)$ in G, creating a new vertex v_6.

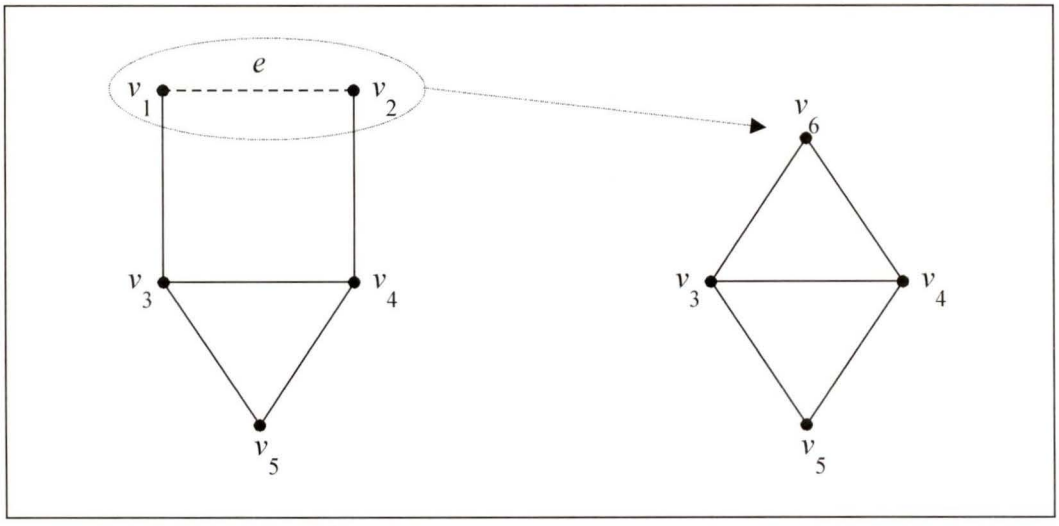


Figure 1: Edge contraction

A *vertex coloring* of a graph G is an assignment of colors to the vertices of G, one color to each vertex, so adjacent vertices are assigned different colors. If c colors are used, then the coloring is referred to as a *c-coloring*. If a c-coloring of G exists, then we say G is *c-colorable*.

A *Hamiltonian path* in a graph G with n vertices is an ordering $(v_1, v_2, \ldots v_n)$ of the vertices such that $(v_i, v_{i+1}) \in E(G)$ for all $i \in (1 \ldots n - 1)$, and each vertex in the ordering is distinct.

The *degree* of a vertex v, denoted $\deg(v)$, is the number of vertices which are adjacent to v. We say a graph G is *d-regular* if every vertex in G has degree d.

2.2 Definitions

Given a matrix M, we will denote the element in row i, column j as $m_{i,j}$. Throughout this thesis, all matrices we consider are Boolean.

Given a Boolean matrix M, with rows $r_1 \ldots r_n$, we define the *Hamming distance* $H(r_a, r_b)$ between any two rows as the number of columns in which the two rows are different. The concept of Hamming distance comes from communication theory, where it denotes the number of bits that must be changed to transform one string of bits into another.

2.3 Algorithms and Complexity

This thesis is concerned with finding algorithms for, and determining the complexity of Cache Scheduling and related problems. We will define Cache Scheduling in several forms for the purpose of studying its' complexity. We illustrate these different forms with the VERTEX COLORING problem. (We will use SMALL CAPS for formal problems throughout the thesis.) VERTEX COLORING [17] is a *decision problem*: Given a graph G and a positive integer k, does a k-coloring of G exist?

We also consider *optimization problems.* The optimization problem corresponding to vertex coloring is denoted MIN VERTEX COLORING: Given a graph G, what is the *smallest* k such that G has a k-coloring?

When searching for efficient algorithms, we usually wish to solve a *constructive* version of the problem. For example, a constructive algorithm for MIN VERTEX COLORING not only returns the integer k, but also provides a k-coloring of the input graph.

Finally, we will consider *fixed parameter* versions of these problems, in which the value k is not part of the input, but may be specific to the algorithm. For example, 2-VERTEX COLORING: Given a graph G, does a 2-coloring of G exist? Fixing a parameter in this way may change the complexity of the problem. For example, VERTEX COLORING is NP-complete, but 2-VERTEX COLORING is solvable in polynomial time [17]. Similar results can be obtained for many problems.

2.4 Treewidth, Pathwidth, and Minor Orderings

Robertson and Seymour, in their series of papers [38]-[42] have discovered deep results in the theory of graph minors. New techniques based on these results have made it possible to obtain non-constructive proofs of the polynomial-time decidability of many problems. The well-known problems of Treewidth and Pathwidth are closely related to graph minor theory, and in fact were introduced in [41] as a central component of the theory. For more information, see the survey articles by Robertson and Seymour [40], and Bodlaender [4]. In this section we give basic definitions and an outline of some primary results used throughout the thesis.

2.4.1 Treewidth and Pathwidth

The graph metrics of Treewidth and Pathwidth can be defined in two equivalent ways. The usual approach is to begin with definitions of *k-trees* and *k-paths*, and then define graphs with treewidth and pathwidth at most k to be *partial k-trees* and *k-paths*. For more information and definitions, refer to any of the standard works; for example, [27], [40].

However, we will define Treewidth and Pathwidth in terms of *decompositions*, as that method is more useful in relation to Cache Scheduling.

A *tree-decomposition* of a graph $G = (V, E)$ is a pair $D = (S, T)$, with $S = \{X_i \mid i \in I\}$ a collection of subsets X_i of the vertices of G, and $T = (I, F)$ a tree with vertices $i \in I$ and edges $\{i, j\} \in F$, with one vertex of T for each subset in the collection S, such that the following three conditions are satisfied:

1. The union of all the X_i's = $V(G)$.
2. For all edges $\{v, w\}$ in $E(G)$, there is a subset $X_i \in S$ such that both v and w are contained in X_i.
3. For each vertex $x \in V(G)$ the set of nodes $\{i \mid x \in X_i\}$ forms a subtree of T.

The *width* of a tree-decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ is the maximum over $i \in I$ of $|X_i| - 1$. If a graph G has a tree-decomposition of minimum width k, we say G has

treewidth k. Using this definition of tree-decompositions, any tree has treewidth equal to one.

Given a graph G and a tree-decomposition $D = (S, T)$, the vertices of T are often called nodes. However, we find it more convenient to refer to the sets X_i associated with vertices of T as *nodes*.

Given the definition of tree-decomposition above, we can define a *path-decomposition* as a tree-decomposition in which T is a path. However, for easier reference we give a simpler, equivalent definition:

Definition: A *path-decomposition* of a graph $G = (V, E)$ is a list D of subsets X_i of $V(G)$, $1 \leq i \leq l$, which satisfies the following three conditions:

1. The union of all the X_i's = $V(G)$.
2. For all edges $\{v, w\}$ in $E(G)$, there is a subset X_i in D such that both v and w are contained in X_i.
3. For each vertex $x \in V(G)$, if $x \in X_i$ and $x \in X_k$ with $i \leq k$, then $x \in X_j$ for $i \leq j \leq k$.

The *width* of a path-decomposition D is the maximum over $1 \leq i \leq l$ of $|X_i| - 1$. If a minimum width path-decomposition of G has width k, we say that G has *pathwidth k*. Again, we will call the sets X_i nodes. We can now formally define the decision problems Treewidth and Pathwidth:

TREEWIDTH

Instance: A graph G and an integer k.

Question: Does G have a tree-decomposition D with width k?

PATHWIDTH

Instance: A graph G and an integer k.

Question: Does G have a path-decomposition D with width k?

We can also define optimization versions and fixed parameter versions of Treewidth and Pathwidth:

MIN PATHWIDTH

Instance: A graph G

Question: What is the pathwidth of G?

k-PATHWIDTH

Instance: A graph G

Question: Does G have pathwidth at most k?

MIN TREEWIDTH and k-TREEWIDTH are similarly defined. Practical applications require constructive algorithms to solve or approximate MIN TREEWIDTH and MIN PATHWIDTH, returning minimum width (or approximately minimum) tree- or path-decompositions of the input graph.

These are interesting problems, in part because there are polynomial or even linear time algorithms for many NP-complete problems when the input is restricted to graphs of treewidth or pathwidth k, for some constant k. For example, see [8] and [3]. These algorithms require that a tree-decomposition of the input graph is given. Consequently, determining the treewidth of graphs (both constructively and non-constructively) are important problems and have been extensively studied.

As shown by Arnborg, Corneil, and Proskurowski in [2], TREEWIDTH for arbitrary graphs is NP-complete. However, a great deal of work has been done on finding approximate solutions, algorithms for special cases, and solutions for k-TREEWIDTH and k-PATHWIDTH.

For example, Bodlaender, Gilbert, Hafsteinsson, and Kloks [7] give polynomial time approximation algorithms for MIN TREEWIDTH and MIN PATHWIDTH. These algorithms find tree decompositions of width within a factor of $\log n$, and path decompositions of width within a factor of $\log^2 n$. Arnborg et al show in [2] that for fixed k, k-TREEWIDTH

can be solved in polynomial time, giving a constructive algorithm which returns tree-decompositions in time[1] $O(n^{k+3})$.

The best results for k-TREEWIDTH and k-PATHWIDTH to date are by Bodlaender in [5] which gives $O(n)$ constructive algorithms[2] based on an algorithm by Bodlaender and Kloks [6]. Unfortunately, these algorithms have large constant terms (exponential in polynomials of k) which prevent useful implementations for all but small values of k. More practical algorithms for $k \leq 3$ are provided by Matoušek and Thomas [34], and for $k \leq 4$ by Sanders [43].

2.4.2 Minor Orderings

A graph H is a *minor* of a graph G, written $H \leq G$, if a graph isomorphic to H can be obtained from a subgraph of G by contracting edges. The relation $\leq$ defines a partial ordering on finite graphs. The example on the next page shows that graph B is a minor of graph A. (The edge to be contracted at each step is indicated by a dashed line.)

---

[1] The referenced paper uses an alternate definition of treewidth. Using their definition, the algorithm is $O(n^{k+2})$.

[2] The $O(n)$ constructive algorithm for k-PATHWIDTH is not described but is a consequence of the given results.

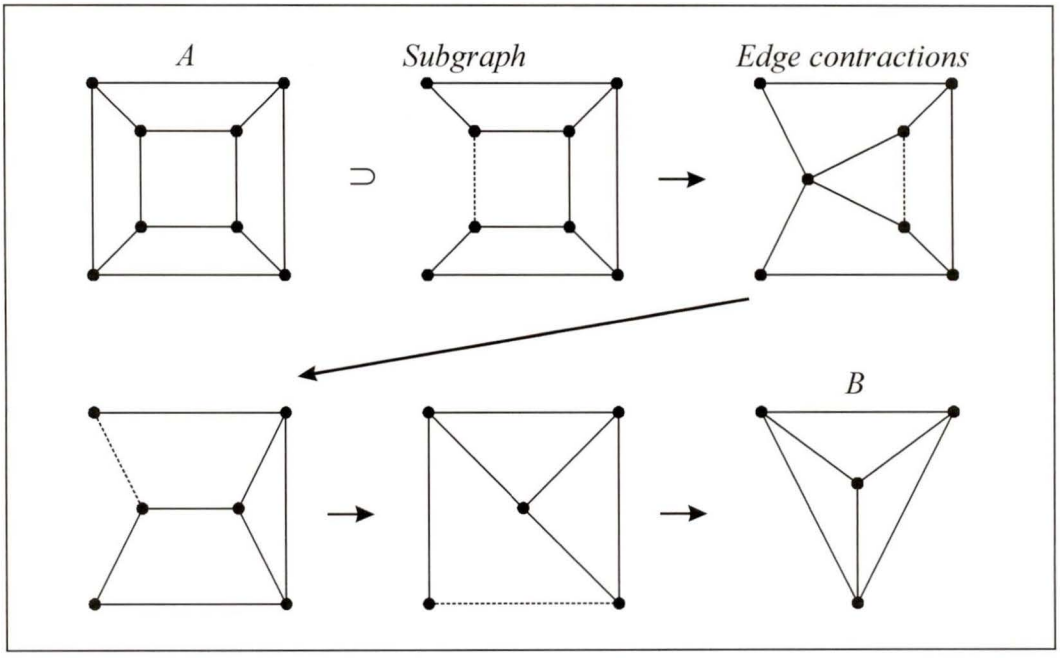


Figure 2: Graph Minor Ordering

A class $\boldsymbol{C}$ of graphs is *closed under the minor ordering* if the two conditions $G \in \boldsymbol{C}$ and $H \leq G$ together imply that H is in $\boldsymbol{C}$. One of the most significant results in graph theory is given by the following theorem:

Theorem 2.1 (Robertson-Seymour) [39]

> Any family of finite graphs which is closed under the minor ordering and which excludes some planar graph has a polynomial-time algorithm for testing membership.

The algorithm that accompanies this theorem is in fact cubic. A recent result by Bodlaender gives an improvement of the Robertson-Seymour theorem:

Theorem 2.2 (Bodlaender) [5]

> Any family of graphs that is closed under the minor ordering and excludes some planar graph has a linear time algorithm for testing membership.

Both the Robertson-Seymour theorem and Bodlaender's theorem are non-constructive. Furthermore, large constant factors preclude efficient implementations of these algorithms. For example, Bodlaender's algorithm in its current form is probably not practical for $k \geq 3$, where k is the treewidth of the graphs under consideration.

3. Problem Introduction

In this section we define Cache Scheduling and explain the origins of the problem in the interaction of software and hardware.

Cache Scheduling arises from a common problem: we want computers to run faster. More precisely, we want to optimize the performance of a set of computations which all must run on a single processor, or CPU. Each computation, which we call a process, requires access to some of the computer's memory.

Cache Scheduling is a problem of optimizing the interaction between a set of processes and a *storage hierarchy*. Storage hierarchies are a central feature of modern computer architectures, and can be divided into two main levels:

- *Primary storage* is high-speed memory which can be operated on directly by the CPU, but has limited capacity and high cost.
- *Secondary storage* is slower and cheaper storage which cannot be operated on directly, but must be copied into primary storage for processing. Secondary storage may be several orders of magnitude larger than primary storage.

Memory is addressed as a consecutive sequence of bytes. All storage is logically organized into equally sized *pages* of consecutive bytes.

It is common for primary storage to be used as a *cache*. A cache holds copies of pages of memory from secondary storage while they are needed by the CPU. When a page is no longer needed, it may be *discarded* (removed) from the cache. If the copy in the cache has been modified, it will first be copied back to secondary storage. By using caches, we hope to gain the advantages of both primary and secondary storage – high speed and high capacity – at the cost of some difficulty in managing the cache.

Caching gives increased performance for two reasons. First, secondary storage devices, such as hard drives, are often structured so that accessing an entire page of memory takes

only a small increase in time over accessing a single byte. Second, there is a high probability that if a process accesses a byte of memory once, it will access it several times, and also access many nearby bytes of memory.

Consequently, when memory is copied from secondary to primary storage for processing, it is done in whole pages in the expectation that many bytes from that page will be used.

When a process accesses a page which is not in primary storage, it must be copied from secondary storage. This is called a *page fault*, or a *cache miss*, and may take a significant amount of time. In fact, were it not for the cost of the cache misses (and some overhead for organizing the cache), caching would provide the capacity of secondary storage together with the speed of primary storage. Therefore, we place a high priority on minimizing the number of cache misses that occur. The cache scheduling problem, informally, is to minimize the occurrence of cache misses by finding an optimal ordering of processes.

3.1 Problem model and assumptions

In this section, we discuss the problem of ordering a fixed set of processes and cache load operations to minimize cache misses. Several assumptions are given which are important to our formal model of the problem. We give an example situation where the assumptions hold, and show how changing the order of processes can change the number of cache misses.

As described in the previous section, the primary storage, or cache, holds a relatively small number of pages that are copied from a larger secondary storage for access and modification as processes run.

In our problem model, we make the following assumptions:

- No process requires access to more pages than will fit in the cache simultaneously.
- Every page required by a process must be in the cache before the process begins to execute.

- Each process runs to completion once it has been started.
- We know in advance which memory pages are needed by each process.
- There are no constraints on the possible ordering of the processes.
- We control which pages are loaded and discarded from the cache.
- All pages are the same size and may be loaded into any position in the cache.
- The cache begins empty.

None of these assumptions apply to some real systems, such as hardware-controlled memory caches. In such circumstances, there is usually insufficient information available to determine an optimal caching strategy. Section 3.3 has more information on such situations.

An example problem where all of these assumptions may hold is in a database server which processes a set of queries. The cache is part of the computer's memory, and secondary storage is the hard drive holding the database files. Each query solution is computed by a single process, which requires access to several pages from the database. We know what pages from the database must be loaded for each query. None of the queries modify the database, so there are no constraints on their ordering. The database server software controls the scheduling of the processes as well as loading and discarding pages from the database files. If the database software uses an algorithm for cache scheduling, performance may be significantly improved.

Changing the ordering of processes may change the number of cache misses that occur. Suppose two processes P_1 and P_2 both use some page g. It is clear that if P_1 and P_2 are run consecutively, then g need only be copied from secondary storage once. Furthermore, even if some process P_3 is run between P_1 and P_2, if P_3 does not access too many pages of memory, g can be loaded once, used by P_1, remain unused in the cache while P_3 runs, and then used again by P_2. On the other hand, if one or more processes are run between P_1 and P_2, and page g is discarded to make room for some other page, then g will have to be loaded a second time before P_2 can run.

Given a situation for which our assumptions hold, it is clear that an optimal ordering of processes and allocation of pages in the cache exists which minimizes the number of cache misses. Intuitively, we hope to find an ordering of processes so that most memory pages required by each process will already be in the cache when that process begins to run, minimizing the total number of cache misses.

This problem was posed by Kai Li, a professor of parallel systems at Princeton University [30]. We will state this problem formally in terms of a matrix in which rows represent processes and the columns represent the memory pages in secondary storage.

3.2 Problem Definition

In this section we give the formal definition of the Cache Scheduling problem, which models the problem of processes interacting with a cache as described in the previous section.

Let M be a Boolean matrix with n rows and m columns. Unless stated otherwise, we will always assume every row and every column of M contains at least one 1. The *weight* of row i is the number of 1s in row i. The *maximum row weight* of M, denoted $w(M)$, is the maximum of all the row weights of M.

Let π be a permutation $\pi(1)\ldots\pi(n)$ of the rows of M. Denote the permuted matrix as M_π. A *block* b is a triple $b = (h, i, j)$, defined with respect to an $m \times n$ matrix M_π, such that:

- $1 \le h \le m$. (h is a column of M_π)
- $1 \le i \le j \le n$. (i and j are rows of M_π)

A block $b = (h, i, j)$ is said to *cover* all the elements of M_π in column h from rows i to j. A *covering* of M_π is a set C of blocks such that every 1 in M_π is covered by some block in C. Each block may also cover some 0's as well. If two or more blocks occur in the same column, they cannot overlap, and must be separated by at least one row; we do not allow a covering to contain blocks (h, i, j) and $(h, j + 1, k)$. A *k-covering* of M_π is a covering in

which no more than k elements on any row are covered, and the *cost* of a covering C is the number of blocks in C.

We find it convenient to refer to a *row permutation* of a matrix M as any matrix with the same set of rows as M arranged in some fixed permutation. We can now give a formal definition of the problem:

CACHE SCHEDULING (CS)

Instance: A Boolean matrix M, an integer k with $w(M) \leq k$, and an integer c.

Question: Does there exist M^* and C such that M^* is a row permutation of M, and C is a k-covering of M^* with cost at most c?

If a solution exists, we say M has a width k cache scheduling of cost c. We will concisely state this as $CS(M, k, c) = true$; otherwise $CS(M, k, c) = false$. Unless stated otherwise, we will assume that the matrices we consider have uniform row weights, so that every row of M contains exactly $w(M)$ ones. For convenience, we will refer to $w(M)$ simply as w when no ambiguity would result. Similarly, we will use n and m to refer to the number of rows and columns of M, respectively.

This combinatorial decision problem corresponds exactly to the problem of cache management that we want to optimize. As stated earlier, matrix rows represent processes, and matrix columns represent pages of secondary storage. A 1 at row i, column j, indicates that process i uses memory page j. All other matrix entries are 0s. The parameter k is the cache size, in pages. A permutation of rows of the matrix represents a permutation of processes to be run sequentially.

Each block (h, i, j) in the covering represents the loading of page h into the cache when process i begins. Page h remains in the cache until process j ends, when it is discarded. Any set of at most k pages is a potential *cache position*. In particular, if we have a cache scheduling of a matrix, the set of elements on row i that are covered by blocks is the cache position at row i. Since every 1 is covered by a block, every page required by each

process is present in the cache when the process begins. Since every row has at most k elements covered, no more than k elements are loaded into the cache at any time.

Therefore, if a row permutation of M with an associated k-covering of cost c exists, it corresponds to a permutation of processes which can be run in a cache of size k with c cache misses.

3.2.1 Cache Scheduling optimization problem

We have defined Cache Scheduling as a decision problem. We will also find it convenient to define Cache Scheduling as an optimization problem. Cache Scheduling is somewhat unusual in that it has two parameters: the cache size and the cost of the covering. We specify a cache size parameter k, and seek to minimize the cost c of the k-covering:

MIN CACHE SCHEDULING

Instance: A Boolean matrix M, an integer k with $w(M) \leq k$

Question: What is the smallest c such that M has a row permutation with a k-covering of size c?

In practice, we wish to solve the MIN CACHE SCHEDULING problem constructively, obtaining an optimal row permutation and k-covering. We also find it useful to remove the parameter k from the input of the problem, defining Cache Scheduling as a parameterized decision problem. This may allow us to develop more efficient algorithms for fixed values of k.

k-CACHE SCHEDULING

Instance: A Boolean matrix M, an integer c

Question: Can we compute M^* and C such that M^* is a row permutation of M, and C is a k-covering of M^* with cost at most c?

Further variations such as MIN k-CACHE SCHEDULING follow the same pattern and are introduced as necessary.

3.2.2 Matrix Bounds and Representation

In this section, we note some simple bounds on the relative number of rows and columns in the matrix of instances of Cache Scheduling. We also briefly discuss the representation of instances of Cache Scheduling. We will later assume that certain operations on the matrix can be performed efficiently, and this is only true if an appropriate representation of the matrix is used.

As described earlier, problem instances for Cache Scheduling include a Boolean matrix of zeros and ones, with n columns, m rows, and w 1s per row. The width of the covering, k, is between one and m, and the row weight is between one and k.

As every column must contain at least one 1, we know that $m \leq nw$. We assume that all rows are unique, and therefore $n \leq \binom{m}{w}$. There are $\binom{m}{k}$ possible cache positions, each of which covers at most $\binom{k}{w}$ rows. Conversely, each row is covered by $\binom{m-w}{k-w}$ cache positions. Each column contains a 1, so the cost c_o of an optimal covering is always greater than or equal to m.

We note that if variable row weights are allowed, a given row a may in fact contain 1s on a proper subset of the columns containing 1s on another row b. Following the terminology of [12], we say row b *dominates* row a, and a is *dominated* by b. All dominated rows can be identified and removed in $O(n^2m)$ time with a simple algorithm. Given a cache scheduling of the resulting matrix, the dominated rows can be reinserted in time $O(n)$ without any increase in the cost of the k-covering. Therefore, if we relax the assumption that all rows have weight w, we may find it convenient to assume that there are no dominated rows. Of course, when all rows are unique and have row weight exactly w, no row can be dominated.

An obvious way to represent such a matrix is to store each element $m_{i,j}$ in a separate memory location, laying out all of the elements of the matrix as a table in the computer's

memory. However, this is not very efficient, as practical instances of Cache Scheduling will typically involve very large matrices that consist mostly of zeros. A better representation of a $n \times m$ matrix is as n lists of integers, one list for each row of the matrix, where each list holds the column numbers of all the 1s in that row. This representation makes it possible to interchange rows of a matrix in constant time, and allows calculation of the Hamming distance between rows in time $O(w)$.

3.3 The Paging Problem

Cache Scheduling is similar to the *paging problem*, an important online problem [20]. An *online problem* is one in which data is supplied in incremental steps, and an *online algorithm* computes a solution to an online problem. The solution must be updated at each step as more data is provided.

The paging problem, like Cache Scheduling, considers a cache of k pages, and a secondary storage of m pages. The paging problem differs from Cache Scheduling in that a sequence of page requests is be presented one at a time to the algorithm. A *paging algorithm* must decide at each step which one of the k pages currently in the cache to discard in order to make room for each new page. The algorithm is not required or permitted to change the order of page requests.

A commonly used general algorithm for the paging problem is to discard the least recently used page. More sophisticated algorithms may use more elaborate statistical evaluations of page use, or take advantage of limited knowledge of the future which may be available in special cases. For more information, see [25] and the next section.

In the offline version of the paging problem, the entire sequence of page requests is supplied in advance. It has long been known that an optimal solution can be computed for the offline paging problem with a greedy algorithm which discards the page whose next request is the furthest in the future [20].

In Chapter 4, we describe the application of this greedy algorithm to instances of Cache Scheduling in which the permutation of rows is fixed. The problem of finding an optimal

k-covering of a fixed row permutation is a generalization of the offline version of the paging problem. Instead of a sequence of single page requests, we have a sequence of sets of w page requests.

3.4 Parallel and Multi-Process Prefetching and Caching

In recent work by Kimbrel [25], the Parallel Prefetching and Caching, and Multi-process Prefetching and Caching problems are introduced. These problems are similar to the Cache Scheduling problem we have defined in Section 3.2. In this section, we briefly describe these problems and the differences between them and Cache Scheduling.

The Parallel Prefetching and Caching (PPC) problem is a generalization of the online paging problem to more accurately model real systems. The paging problem assumes that memory pages will be loaded immediately before their first use. For each cache miss, some constant time is needed to load the page. Algorithms for the paging problem thus attempt to reduce the number of cache misses in an effort to reduce the time required for the set of processes to complete.

Algorithms for the PPC problem require at least limited knowledge of future page requests. Given such knowledge, pages may be loaded, or *fetched*, before they are first used. This is *prefetching*, and can reduce or eliminate the time delay incurred by loading a page from the cache. The drawback is that prefetching a page takes a space in the cache, overwriting some page already there. Algorithms for the PPC problem must therefore consider several factors to optimize performance, deciding which pages may be profitably replaced, and when.

When all pages are located on the same secondary storage device, or disk, only one page load may be in progress at any time. This single-disk problem is the *integrated prefetching and caching problem*, a special case of the PPC problem. If, as is the case on many real systems, pages are distributed among many such disks, then many prefetches may be ongoing simultaneously, with one from each disk. This introduces parallelism,

giving the *parallel prefetching and caching problem*. In [25], near-optimal approximation algorithms for the PPC problem are described and analyzed.

Cache Scheduling and the PPC problem are both generalizations of paging problems. However, Cache Scheduling generalizes the offline version of the paging problem, while PPC generalizes the online paging problem. Furthermore, they generalize these problems in different ways. Some significant differences between Cache Scheduling and the Parallel Prefetching and Caching problem are:

- Cache Scheduling considers a *set of processes*, each accessing at most w pages. In contrast, the PPC problem considers a *sequence of page requests*.
- For each process in Cache Scheduling, *all* the pages required by the process must be in the cache before the process may run. The PPC problem considers pages one at a time.
- Cache Scheduling does not make use of prefetching.
- The PPC problem does not consider reordering the sequence of page requests to improve performance.

The Multi-process Prefetching and Caching (MPPC) problem [25] is more like Cache Scheduling, as it considers multiple processes and finds an optimal ordering of a set of requests. The MPPC problem is defined as follows:

Let $B = B^1 \cup B^2 \cup \ldots \cup B^m$ be a collection of disjoint sets of pages. A *request sequence* is an ordered sequence of requests $R^k = r_1^k, r_2^k, \ldots r_{|R^k|}^k$, where each $r_i^k \in B^k$. There are m separate request sequences $R^1, \ldots R^m$, and there is a cache that contains at most k pages from B at any time. Fetching a page from a disk into the cache takes F time units. Each request sequence represents the requests of a single process. As such, the requests in each request sequence must be satisfied in order. A single request can be satisfied in one unit of time once the requested page is present in the cache.

We imagine that for each request sequence there is a *cursor* that points to the next request in that sequence to be satisfied. If this request is for a page that is in the cache, the cursor may advance to the next request during the next time unit. However, if several cursors point to pages that are present in the cache, only one cursor may advance in a single time unit. If all cursors point to requests that are not in the cache, processing *stalls* until one of the missing pages arrives in the cache. This will happen when the fetch for that page completes. Prefetches can overlap processing to the extent that requests to pages in the cache may be satisfied while a prefetch is occurring.

There are two constraints on prefetching:

1. If a fetch of page b_1 is initiated at time t and the cache contains k pages at that time, some page b_2 must be discarded to make room for the incoming page. Neither b_1 nor b_2 is available during the F time units between t and $t + F$ while the fetch is occuring.
2. Fetches are performed sequentially: If a fetch is initiated at time t, no other fetch can be initiated until time $t' \geq t + F$.

The goal of an algorithm for MPPC is, when given the collection of request sets, to construct a schedule for prefetching and satisfying requests that minimizes the total time required to satisfy all requests in each R^k. This total time is $\sum_{k=1}^{m} \left| R^k \right|$. The schedule specifies which pages to fetch, when to fetch them, which pages to evict, and when to satisfy each request.

This is more like Cache Scheduling than the PPC problem, but is still quite different. The important differences are:

- All the pages requested by any process in Cache Scheduling can fit in the cache *simultaneously*, and in fact are required to do so. MPPC does not have this limitation or this requirement, and instead only requires that pages requests be satisfied *sequentially* for each process.

- Each process in MPPC requests a sequence of pages which is disjoint from the sequence of pages requested by other processes. This is not the case in Cache Scheduling. More precisely, if such a situation arises in Cache Scheduling, it is trivial to find a solution.
- Cache Scheduling does not consider the issues of prefetching and the time to load pages as MPPC does.

A problem which combined some key features of Cache Scheduling and MPPC could be developed by considering a variation of MPPC in which the request sequences are not disjoint. Alternatively, an online version of Cache Scheduling could be defined that would consider the issue of prefetching.

The Thread Scheduling problem is also a generalization of the paging problem. Let B denote a set of blocks in secondary storage. A *request sequence* is an ordered sequence of requests $R^k = r_1^k, r_2^k, \ldots r_{|R^k|}^k$, where each $r_i^k \in B^k$. There are m separate request sequences $R^1, \ldots R^m$. There is a cache that contains at most k blocks from B at any time.

The requests in each sequence R^i must be served in order and consecutively, but there is no restriction on the ordering of the sequences.

The goal is to construct an ordering of the sequences such that the number of cache misses is minimized. (Given any fixed order of the sequences, it is easy to determine the number of cache misses required.)

The Thread Scheduling problem is very similar to Cache Scheduling. Unlike MPPC, the request sequences are not required to be disjoint. Thread Scheduling, like Cache Scheduling simply tries to minimize the number of cache misses and ignores prefetching.

The difference is that the request sequences for each process in Thread Scheduling are not required to have length less than or equal to k. The page set for each process in Cache Scheduling must have size k or less to fit in the cache.

It seems that a good approximation algorithm for Thread Scheduling could easily be modified to give good approximate solutions to Cache Scheduling. Apparently, finding such an approximation algorithm remains one of many interesting open problems in this area.

In Section 5.1 we briefly return to this problem and discuss the NP-completeness proof of Thread Scheduling, and show that it does not apply to Cache Scheduling.

4. *k*-Covering a Fixed Permutation

Upon first consideration of the Cache Scheduling problem, it may appear that there are two interconnected subproblems: first to find an optimal permutation of the rows, and second, to compute an optimal *k*-covering for that permutation. However, it is easy to show that given a *fixed* permutation M^* of a matrix M, a simple greedy strategy suffices to find an optimal *k*-covering in linear time.

We find it convenient to use terminology from the informal problem of processes and pages to describe the formal problem of *k*-coverings and row permutations. For example, we will speak of "discarding a page" rather than "ending a block in the *k*-covering". Thus, we describe the greedy strategy as follows:

Greedy Strategy:

> The greedy strategy which results in optimal *k*-coverings is to discard the page which will not be needed for the longest time.

An implementation of this greedy strategy is essentially the optimal offline algorithm for the paging problem described in Section 3.3, except that each row in an instance of Cache Scheduling may require several pages to be loaded at each step. This implies that the only difficult part of Cache Scheduling is finding the optimal permutation of rows.

An algorithm which computes an optimal *k*-covering for a fixed permutation must only decide what page or pages to discard from the cache when room must be made for new pages. There is never any advantage, under our model, to loading a page before it is needed, or to unnecessarily discarding pages from the cache.

Recall that a block (h, i, j) covers the matrix elements in column h from rows i through j. A *k*-covering is a set of blocks which covers every one in the matrix and covers at most k elements on every row.

Given a matrix representation of the problem, the greedy algorithm starts on the first row and steps through the rows one at a time. New blocks are started at each row as necessary, ensuring that every 1 in the matrix is included in a block. Blocks are ended as necessary, using the greedy strategy, to ensure that no more than k elements are covered on any row.

Theorem 4.1:

The greedy algorithm constructs a k-covering of M^* of minimum cost.

Proof: Clearly, the greedy algorithm constructs a k-covering of M^*, as no more than k elements are covered on any row, and every 1 in the matrix is covered. Now, we will show that the constructed k-covering has minimum cost.

Let C be the k-covering constructed by the greedy algorithm, and let C' be a minimum cost k-covering. We prove that C' has the same cost as C by showing that C' can be repeatedly modified to become the same as C, with no increase in cost. We may assume, without loss of generality, that the first and last rows of any block contain a 1.

Let i be the first row for which C' and C *disagree* on which block to end, that is, C' and C end blocks in different columns. Then at row i, C ends a block (g, b, i) which continues through row i in C', while C' ends a block (h, a, i) which continues through row i in C.

Figure 3 on the next page shows the relevant 1s in the matrix. Rows a and b are shown on the same line as their relative order is irrelevant in the diagram, all that matters is that they precede row i. Blocks are shown as rounded rectangles.

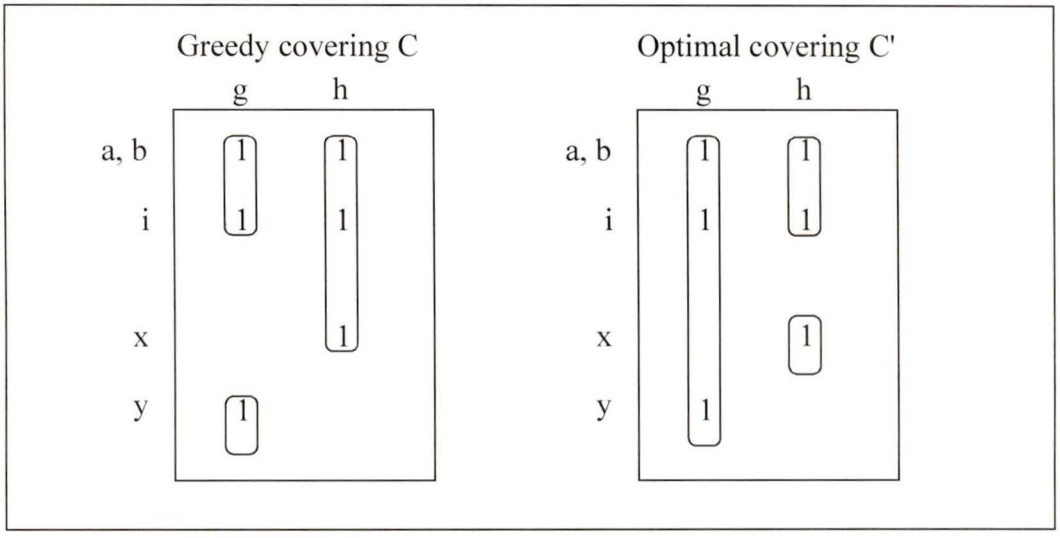


Figure 3: Greedy and Optimal k-coverings

If the last 1 in column g is in fact at row i, then C' may be modified to be identical to C without any increase in cost, because no new block will have to be created in column g. And, if the last 1 in column h is at row i, the block in C need not (and, by our assumption, would not) continue past row i either.

Therefore, we assume there are 1s in both columns g and h after row i. Let x be the row number of the next 1 in column h after row i, and y be the row number of the first 1 in column g after row i. Then C contains another block in column g, beginning at row y, and C' contains another block (h, x, z) in column h beginning at row x and extending to some row z, with $z \geq x$, not shown in the diagram. C was produced by the greedy algorithm, so the next 1 in column g after row i must occur on the same row or later than the first 1 in column h after row i. Therefore $x \leq y$.

Now suppose C' is modified to make the same choice as C at row i. By this we mean that the two blocks (h, a, i) and (h, x, z) are replaced by a single block (h, a, z), while the block in column g which begins at or before row i, and continues to at least row y is split into two blocks, one ending at row i and the other beginning at row y.

This modification of C' decreases the number of blocks in column h by one, and increases the number of blocks in column g by one. The number of elements covered in rows i to x–1 remains the same, and, an extra position in the cache will become available in rows x through y–1. The modified C' is still a k-covering with cost c. If we repeat this procedure, after each step the modified C' will have least one more row in common with C. Thus, the process will terminate and we can make C' identical to C without increasing its cost. Therefore, C is also an optimal covering.

□

The greedy algorithm does not require that the row weights of M be uniform, or that every row and every column contain at least one 1, but is applicable for any M and k. We observe that this implies that Cache Scheduling is in NP: we can nondeterministically guess an optimal permutation of the rows of the matrix, and then check it in linear time with the greedy algorithm. This also gives us a (very slow) algorithm for Cache Scheduling, as we can generate the $n!$ permutations and check them in time $O(mnn!)$. In later chapters we will obtain better algorithms.

5. Cache Scheduling is NP-Complete

In this section, we prove Cache Scheduling is NP-complete using a reduction from Hamiltonian Path. As noted in the last section, Cache Scheduling is in NP.

Suppose we have an instance of Hamiltonian Path in a graph G with n vertices and e edges. Let d be the maximum degree of any vertex in G. We assume that G contains no vertices of degree zero.

We represent G as an $n \times m$ Boolean matrix M using the following construction: We create a row r_i for each vertex i, $1 \leq i \leq n$, and two sets of columns. In the first set we have a column c_j for each of the e edges $j \in E(G)$. We call these *edge columns*. Each edge column contains two 1's. We place a 1 at row r_i, column c_j if edge j is incident to vertex i and a 0 otherwise. The number of ones on a row r_v contributed by this first set of columns is exactly equal to the degree of v in G.

We call the second set of columns *extra columns*. Each extra column contains exactly one 1. We add extra columns, with 1s in appropriate rows, so that each row will contain exactly d 1s. This will require at most $d - 1$ extra columns for each row. The following diagram shows an example graph and corresponding matrix.

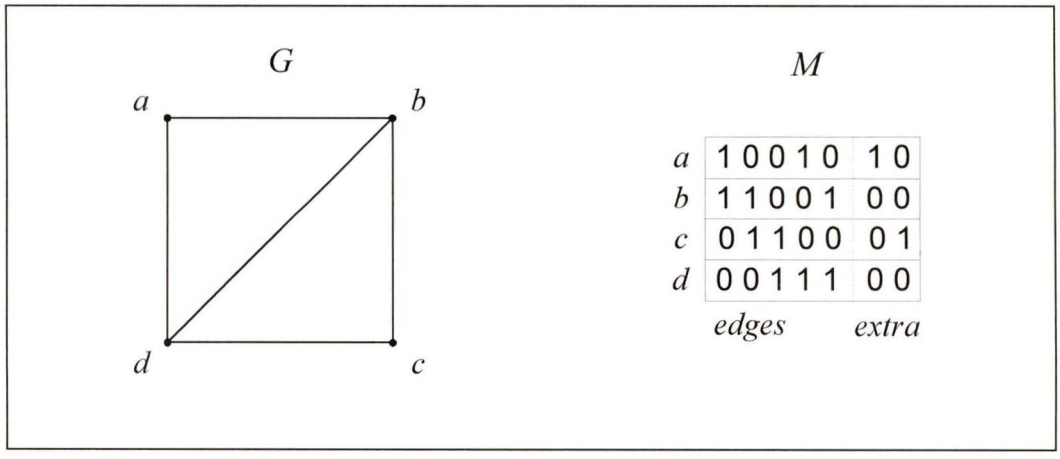


Figure 4: Hamiltonian Path Matrix Construction

The number of edge columns is e. Let the number of extra columns be called x. Then m, the total number of columns, equals $e + x$ and is $O(n^2)$. This matrix construction takes $O(n^3)$ time.

Theorem 5.1:

> There is a Cache Scheduling of M with width d and cost $2e + x - n + 1$ if and only if G contains a Hamiltonian Path.

Proof: Suppose G contains a Hamiltonian Path. If we permute the rows of M in the same order as the vertices in the Hamiltonian path, then for each of the $(n - 1)$ edges $\{u, v\}$ in the Hamiltonian path, rows r_v and r_u will be consecutive, and the two ones in the edge column $c_{u,v}$ will appear consecutively. The other $e - (n - 1)$ edges in the graph must correspond to edge columns in which the two ones are separated by at least one zero.

Now we can construct a d-covering for this row permutation of M. No 0 can appear in any block of a d-covering because there are d ones on each row, and at most d elements may be covered. Each of the $(n - 1)$ columns representing edges in the Hamiltonian path is covered with exactly one block. The other $e - (n - 1)$ edge columns in M must be covered with two blocks each. The x extra columns, each containing a single 1, require 1 block each.

The total cost is $(n - 1) + 2(e - (n - 1)) + x$ which simplifies to $2e + x - n + 1$.

Conversely, suppose M has such a cache scheduling: a row permutation and d-covering with cost $2e + x - n + 1$. The extra columns will require x blocks to cover regardless of the row permutation. Therefore the edge columns must be covered using exactly $2e - n + 1$ blocks.

There is at least one, and at most two blocks in each of the e edge columns, so $e - n + 1$ columns must have a second block, and therefore exactly $(n - 1)$ columns in the first set are covered with one block.

Each of these $(n-1)$ blocks must begin on a different row as all edges are unique. Therefore for each row of the matrix, except the last, there is a block containing two ones which starts on that row and ends on the next. The edges in G corresponding to these $n-1$ columns form a Hamiltonian path in G.

□

This reduction establishes that Cache Scheduling is NP-hard. In combination with the observation of Chapter 4 that Cache Scheduling is in NP, we have proved the following theorem:

Theorem 5.2:

Cache Scheduling is NP-complete.

In particular, the special case of Cache Scheduling with $k = w$, where w is the uniform row weight of the matrix is NP-complete. In Section 6.4, a different proof of this theorem is given, showing that Cache Scheduling is also NP-complete when the number of columns in the matrix m equals c, the cost of the k-covering.

We pose the following conjecture:

Conjecture 5.3:

Cache Scheduling is NP-complete when $k > w$ and $c > m$.

The NP-completeness result from [25] mentioned in Section 3.4 does not prove this conjecture, as described below. However, a step towards a proof of this conjecture is provided by Irani et al [21], who show that Cache Scheduling with $k = 3$ and $w = 2$ is NP-complete.

5.1 NP-completeness of Thread Scheduling

In Section 3.4 we described the Thread Scheduling problem, which is very similar to Cache Scheduling. In [25], Thread Scheduling is shown to be NP-hard. In this section

we briefly consider this NP-hardness proof and explain why it does not apply to Cache Scheduling.

If an NP-completeness reduction for Thread Scheduling existed which, given an instance of a known NP-hard problem, constructed Thread Scheduling problems in which each request sequence had length $\leq k$, such a reduction might with some small modification suffice to show that Cache Scheduling is NP-hard for $k > w$ and $c > m$.

This would give the desired proof of conjecture 5.3 above. Unfortunately, the proof in [25], which is a reduction from the Directed Hamiltonian Path problem, explicitly constructs request sequences which contain subsequences of $> k$ "private blocks", i.e. pages not requested by any other request sequence. This is a key feature of the reduction as it forces all blocks shared between different request sequences to be discarded from the cache during each request sequence.

Thus the proof does not apply to Cache Scheduling.

6. Matrix Row Permutation Problems

In this section we examine several well-known problems and show that they are special cases of Cache Scheduling. We also consider the application of known results for these problems to Cache Scheduling.

6.1 Consecutive Ones

Consider Cache Scheduling in an $n \times m$ matrix M with uniform row weights w. If we restrict our attention to special cases with cache size $k = w$ and total cost $c = m$, we are searching for a permutation of the rows with no 0s included in any block, and exactly one block per column. (Recall that we assume that every row and every column of M contains at least one 1). This special case of Cache Scheduling is known as the Consecutive Ones property.[1] [16].

A matrix M has the *consecutive ones property for columns* if and only if its rows can be permuted so that in each column all of the 1's are consecutive [16]. The consecutive ones problem and related problems are of great practical interest, as they frequently arise in several disciplines. An example that has some similarity to Cache Scheduling first appeared in [18]:

If the rows of a matrix represent records in a database, and the columns represent fixed queries, then a 1 in row i, column j denotes the fact that record i is relevant to query j. If the matrix has the consecutive ones property, then the records of the database can be ordered on a secondary storage device so that each query can be processed by reading a single consecutive series of records. This optimizes performance.

There is a linear time algorithm based on PQ-trees for testing if a matrix has the Consecutive Ones property. PQ-trees were introduced by Booth and Lueker in [9].

---

[1] Actually, this is a special case of the consecutive ones property, with uniform row weights.

6.1.1 PQ-trees

A PQ-tree is a data structure that represents a set of possible permutations of the n elements in a set U. Any PQ-tree is either empty, representing all $n!$ possible permutations, or is non-empty and represents all permutations in which the elements of each subset in a family of subsets of U occur as a consecutive subsequence.

PQ-trees can be used to find such a permutation, if one exists, by beginning with all permutations and successively removing permutations which fail to contain the specified subsets as consecutive subsequences. A general framework for this reduction is as follows:

```
Boolean Procedure Reduce( Set U, List S of subsets of U );
begin
   Π := { π | π is a permutation of the elements of U };
   for each S ∈ S do begin
      Π := Π ∩ { π | all objects of S are consecutive within π };
   end;
   return (Π = ∅);
end;
```

The set U has n objects, and $\boldsymbol{S}$ is a list of m sets[1]. Clearly, the work done outside the set intersection step in the inner loop is $O(m + n)$. PQ-trees make it possible to perform the m set intersection steps in the inner loop in time $O(m + n + \text{SIZE}(\boldsymbol{S}))$, where $\text{SIZE}(\boldsymbol{S})$ is the sum of the sizes of the sets S in $\boldsymbol{S}$. This gives a total running time for the reduction algorithm of $O(m + n + \text{SIZE}(\boldsymbol{S}))$ [9].

The application of PQ-trees to the Consecutive Ones problem is simple. The n rows of the matrix form the set U. For each of the m columns, all of the rows with 1's form a constraining subset S of U in $\boldsymbol{S}$. If the reduction algorithm returns true, the matrix has the consecutive ones property. Suppose M is specified as a set of lists, one for each row. Each list contains the column numbers of all the 1's in that row. Then the running time is

---

[1] We reverse the meanings of m and n given in [9] to be consistent with the definition of Cache Scheduling.

$O(m + n + f)$, where f is the total number of 1's in the matrix. Therefore, the special case of Cache Scheduling with $k = w$ and $c = m$ can be solved in linear time.

6.2 Pathwidth

A second special case of Cache Scheduling we will examine is Pathwidth, which we first introduced in Section 2.4. In this section we introduce a definition of Pathwidth in matrices, and prove that it is equivalent to the standard definition of Pathwidth. We then show that Pathwidth is a special case of Cache Scheduling.

6.2.1 Pathwidth in Matrices

We define a version of Pathwidth for Boolean matrices:

MATRIX PATHWIDTH

Instance: A Boolean 0/1 matrix M with one or two 1s per row, and an integer k.

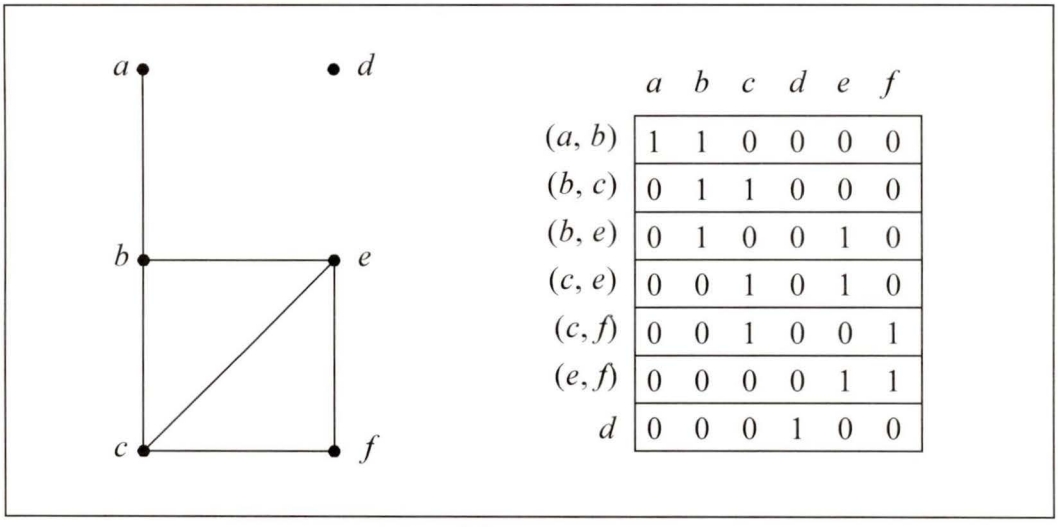
Question: Can we permute the rows of M so that, if in each column we change to * every 0 lying between the column's topmost and bottommost 1, then no row contains more than $k + 1$ 1s and *s?

We say that M has *pathwidth* k when k is the smallest integer for which such a permutation of rows exists.

Now we will show that PATHWIDTH and MATRIX PATHWIDTH are equivalent. We first define an invertible method of converting from a graph to a Boolean matrix with one or two 1s on each row. This method is based on a construction given in [15].

Given a graph G, we construct a Boolean matrix $M(G)$ with one column for every vertex of G and one row for every edge. The ordering of the rows and columns is not significant. On every row we place two 1s in the columns representing vertices connected by that edge, and 0s in every other position. If G has a vertex v with no incident edges, we create a row in $M(G)$ with exactly one 1 in the column corresponding to that vertex. That column will contain just that one 1, of course.

The following diagram shows a graph and the corresponding matrix using this construction.



	a	b	c	d	e	f
(a, b)	1	1	0	0	0	0
(b, c)	0	1	1	0	0	0
(b, e)	0	1	0	0	1	0
(c, e)	0	0	1	0	1	0
(c, f)	0	0	1	0	0	1
(e, f)	0	0	0	0	1	1
d	0	0	0	1	0	0

Figure 5: Pathwidth Graph / Matrix conversion

This construction is invertible: For any Boolean matrix with one or two ones per row, we obtain a graph by creating a vertex for each column, and edges for each row with two ones. Rows with exactly one 1 are represented by isolated vertices. However, we cannot use this method to create graphs from arbitrary Boolean matrices.

Lemma 6.1:

If G has pathwidth k, then $M(G)$ must have pathwidth k.

Proof: If G has pathwidth k, G has a path-decomposition $D = (X_1, X_2, \ldots X_l)$ with width k, and each X_i has size at most $k + 1$. From this path decomposition of G we construct a permutation of the rows of $M(G)$ as follows:

Each row of $M(G)$ has one or two 1s representing either an edge of G, or an isolated vertex in G. Therefore, each row r of $M(G)$ is associated with a set V_r of either one or two vertices.

We partition the rows of $M(G)$ into a list of disjoint groups S_i, $1 \leq i \leq l$, corresponding to the sets X_i in D. Each row r is placed into some group S_i such that V_r is a subset of X_i.

We know there must be such a X_i because of conditions 1 and 2 of the definition of path-decomposition and the construction of the matrix.

We obtain M', a row permutation of $M(G)$, by taking the groups S_i in order and ordering the rows within each group arbitrarily. Let M^* denote the matrix obtained by taking M' and changing to * every 0 between the topmost and bottommost 1 in each column.

If row r in group S_i contains a 1 in column c, then X_i must contain the vertex represented by column c. If row r in group S_i contains a * in column c, there are two possibilities. First, some other row in group S_i may contain a 1 in column c, implying that X_i contains the vertex represented by column c. Or, it may be that every row in group S_i has a * in column c. In that case, there must be 1's in column c in rows in other groups S_a and S_b with $a < i < b$. That implies that X_a and X_b must both contain the vertex represented by column c. By condition 3 of the definition of path-decomposition, X_i must also contain the vertex represented by column c.

Therefore, if row r in group S_i contains either a 1 or a * in column c, X_i must contain the vertex represented by column c. The sets X_i have size at most $k + 1$, and therefore no row of M^* can contain more than $k + 1$ 1s and *s. Therefore, M has pathwidth k.

□

Lemma 6.2:

If $M(G)$ has pathwidth k, then G must have pathwidth k.

Proof: If $M(G)$ has pathwidth k, there exists some row permutation M' of M such that if in each column of M' we change to * each zero lying between the columns topmost and bottommost 1, no row contains more than k 1's and *'s. For convenience, we denote this row permutation of M with the appropriate zeros changed to *s as M^*. We construct a path-decomposition of G as follows:

For each row i of M^*, we create a subset X_i of $V(G)$ such that X_i contains every vertex corresponding to a column for which row i has either a 1 or a *, and no other vertices. We claim that $D = (X_1, X_2, \ldots X_n)$ is a path-decomposition of G with width k.

The first condition D must satisfy is that the union of all the X_i 's = $V(G)$. This is true because every vertex of G has a corresponding column, and at least one row contains a one in that column.

The second condition is that for all edges (v, w) in $E(G)$, there is a subset X_i in D such that both v and w are contained in X_i. This is also true, as every edge (v, w) in $E(G)$ corresponds to some row r in M^* with 1s in the columns corresponding to vertices v and w. Therefore the set X_r will contain both v and w.

The third condition is clearly satisfied as well: if $x \in X_i$ and $x \in X_k$, then both row i and row k must have either 1s or *s in column x. Every zero was changed to a * between the topmost and bottommost 1 in column x, so it must be that $x \in X_k$ as well for $i \leq k \leq j$.

Therefore, D is a path-decomposition of G. Finally, no X_i contains more than $k + 1$ vertices because no row in M^* contains more than $k + 1$ 1s and *s, and each set X_i contains just those vertices corresponding to columns for which row i has either a 1 or a *. Therefore, D has width at most k.

□

Theorem 6.3:

> Graph G has pathwidth k if and only if the equivalent matrix $M(G)$ has pathwidth k.

Proof: Immediate from the previous two lemmas.

□

6.2.2 Matrix Pathwidth is a special case of Cache Scheduling

Consider an instance of CACHE SCHEDULING with an $n \times m$ matrix M_2 with uniform row weights of two. We may wish to know if M_2 has a k-covering with cost equal to m, the number of columns in the matrix. Recall that we abbreviate this question as $CS(M_2, k, m)$. We show that this is equivalent to the MATRIX PATHWIDTH problem: Does M_2 have pathwidth $k - 1$?

Theorem 6.4:

For any $n \times m$ matrix M_2 with uniform row weights of two, $CS(M_2, k, m)$ is true if and only if M_2 has pathwidth $k - 1$.

Proof: Suppose $CS(M_2, k, m)$ is true. Then there exists a permutation M' of the rows of M_2 with a k-covering of cost m, necessarily having one block per column. If we change each 0 covered by a block to a *, every 0 between each columns topmost and bottommost 1 will have been changed to a *, and no row will have more than k 1s and *s covered. This immediately implies that M_2 has pathwidth $k - 1$.

Conversely, suppose M_2 has pathwidth $k - 1$. Then, a permutation of the rows of M_2 exists such that, if in each column we change to * every 0 lying between the column's topmost and bottommost 1, no row contains more than k 1s and *s. If we cover all the 1s and *s in each column with a single block, no row will have more than k elements covered, and we will have created a k-covering of cost m. This implies $CS(M_2, k, m)$ is true.

□

6.3 Gate Matrix Layout

The problem of Gate Matrix Layout [31] is a generalization of MATRIX PATHWIDTH to allow more than two 1s per row. It is also a special case of Cache Scheduling.

GATE MATRIX LAYOUT (GML)[1]

Instance: A Boolean $m \times n$ matrix M and an integer k.

Question: Can we permute the rows of M so that, if in each column we change to * every 0 lying between the column's topmost and bottommost 1, then no row contains more than k 1's and *s?

---

[1] The usual definition of GML is in terms of permuting columns. For consistency with Cache Scheduling, we permute rows instead.

If such a permutation exists, we state this concisely as GML(M, k) = *true*. A 0 that is changed to a * is termed a *fill-in*.

It is easy to show that this is a special case of Cache Scheduling restricted to one block per column; that is, GML(M, k) is identical to CS(M, k, c) if $c = m$.

We assume there is at least one 1 in each column, and therefore if there is a solution to the CS problem, it must have exactly one block covering all the ones in each column. By converting covered zeros to fill-ins, we obtain a solution to the GML problem. The converse is also true: Any solution to the GML problem is a solution to CS with $c = m$ by creating a block from the topmost 1 to the bottommost 1 in each column.

It is shown in [24] that GML is NP-complete. However, Fellows shows in [15] that GML is in P for any fixed value of k. This proof is long, and only the first part is given in full here, as it is relevant to Cache Scheduling.

Our first task is to devise a way to transform an instance of GML into a suitable representation as a graph. Given a Boolean matrix M, we map M to an "expanded" form $x(M)$ in which any row i with $j > 2$ 1s is replaced by $\binom{j}{2}$ rows, each with two 1s, representing every distinct way to choose a pair of 1s from row i.

Notice that this notion of matrix expansion is many-to-one and hence is not invertible, and that this construction takes time polynomial in the size of M. Figure 6 shows an example of matrix expansion.

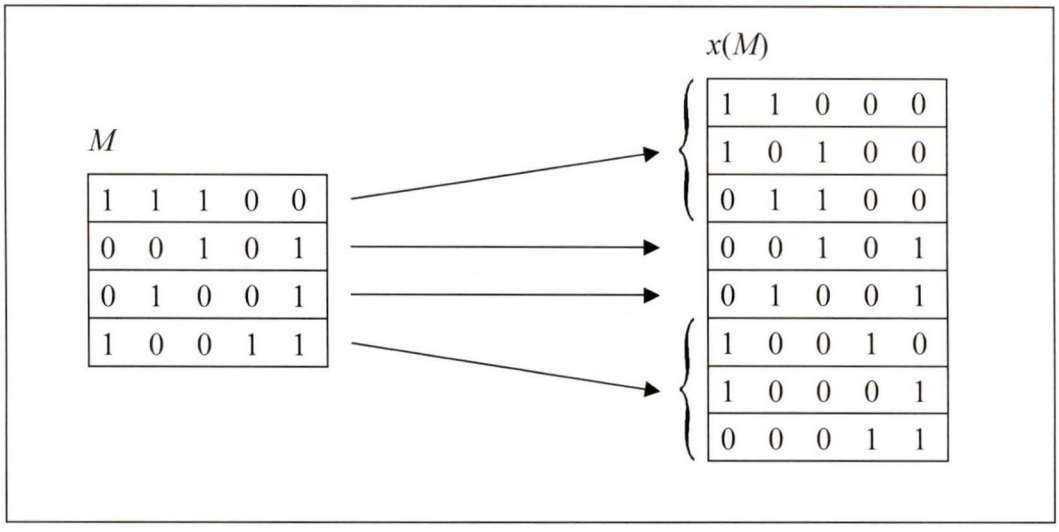


Figure 6: Matrix Expansion

Lemma 6.5 [Fellows]:

$$\text{GML}(M, k) = \text{GML}(x(M), k)$$

Proof: Suppose GML(M, k) = *true*. Then there is a row permutation p of M such that, after *s are included, no row of $p(M)$ contains more than k 1s and *s. Consider the matrix $x(p(M))$, the expanded form of $p(M)$. For a row i of $p(M)$ with $j > 2$ 1s, each of the $\binom{j}{2}$ rows replacing row i in $x(p(M))$ will have a * only where row i had either a 1 or a *. Furthermore, rows above and below row i are unaffected by this expansion. Therefore, GML($x(p(M))$, k) = *true*. Since $x(p(M)) = p'(x(M))$ for some appropriate permutation p' of the rows of $x(M)$, we have GML($x(M)$, k) = *true*.

Conversely, suppose GML($x(M)$, k) = *true*. Then there is some permutation p of the rows of $x(M)$ such that, after *s are included, no row of $x(M)$ contains more than k 1s and *s. Consider any row i of M with $j > 2$ 1s and *s. Let $x(i)$ be the set of $\binom{j}{2}$ expanded rows which replace row i in $x(M)$. Let f and l denote the indices of the first and last rows, respectively, from $x(i)$ in $p(x(M))$.

Since all columns with 1s in row i are pairwise connected by rows in $x(i)$, one of the rows in $x(i)$ must have 1s and *s in every column of row i.

To see why this must be, let columns s and t be two of the j columns which have 1s and *s in M on row i. Specifically, let s be the column which has it's last 1 on the earliest row in $x(M)$. Let t be the column which has its first 1 on the latest row in $x(M)$. Clearly, the first 1 in column t must occur on the same row, or an earlier row than the last 1 in column s, as there is a row h in $x(i)$, and $x(M)$, with 1s in columns s and t. All of the other $j - 2$ columns which have 1s and *s in M on row i must have their first 1s before the first 1 in row t, and their last 1s after the last 1 of column s, and therefore must have a 1 or a * on row h. Therefore row h has a 1 or a * in each of the j columns.

Let h be such a row in $p(x(M))$, where $f \leq h \leq l$. Now replace row h in $p(x(M))$ with row i of M. The new row has more 1s than did the old, but each of those 1s is in a column where the old row had a 1 or a *, so the total number of 1s and *s does not increase.

Next, delete from $p(x(M))$ the $\binom{j}{2} - 1$ other rows of $x(i)$. Any rows remaining between the original location of rows f and l do not have new *s introduced by this procedure, as the first 1 in each column cannot appear on an earlier row, and the last 1 in each column cannot appear on a later row. We conclude that the resulting matrix has no more than k 1s and *s on each row. Iterating this construction yields a permutation p' of the rows of M, and hence GML(M, k) = *true*.

□

Theorem 6.6:

GML is in P for any fixed value of k. [15]

Proof: We have already shown that given any matrix M and integer k forming an instance of Cache Scheduling, we can construct the expanded matrix $x(M)$ in polynomial time, and $x(M)$ has a gate matrix layout of width k if and only if M does. We now show that we can determine, in polynomial time for each fixed k, if $x(M)$ has such a gate matrix layout.

Since each row of $x(M)$ contains at most two 1s, we map $x(M)$ to a graph $g(x(M))$ whose vertices correspond to columns and whose edges represent rows. Notice that g is invertible in that we can recover from $g(x(M))$ the set of rows in $x(M)$. Also, $g(x(M))$ may contain loops for rows in $x(M)$ with a single 1, and multiple edges for duplicated rows. Therefore, for any graph G there exists at least one Boolean matrix M such that $G = g(x(M))$.

This conversion of $x(M)$ to a graph is essentially the same as the graph construction which we have used to prove the equivalence of PATHWIDTH and MATRIX PATHWIDTH, except that multiple edges are allowed, and rows with a single 1 are represented by loops instead of isolated vertices. The proof continues by proving the following two lemmas:

1. The family of graphs G such that $G = g(x(M))$ and GML(M, k) = *true* is closed under the minor ordering.
2. For any fixed k, there exists a planar graph G such that $G = g(x(M))$ and GML(M, k) = *false*.

Therefore the Robertson-Seymour theorem can be applied, and the class of matrices for which GML(M, k) = *true* is polynomial-time recognizable.

□

6.4 Pathwidth, Gate Matrix Layout, and Cache Scheduling

It is evident from the previous sections that these three problems are closely related. In this section we discuss these relationships in more detail.

A technical difference between these problems is the exact meaning of the variable "k". With our definition of pathwidth, "pathwidth k" means $k + 1$ elements per node in a path-decomposition, or $k + 1$ elements covered per matrix row. However, a Gate Matrix Layout of width k or Cache Scheduling with cache size k means k elements covered per row. This is because the definition of the width of a path-decomposition subtracts one from the size of the sets so a simple path has pathwidth one.

In particular, note that the matrix version of Pathwidth is nearly identical to GML. The significant difference is that Pathwidth restricts the matrix to just two 1s on every row, while GML generalizes to allow more than two ones on a row. However, Lemma 6.5 shows that any matrix M has a gate matrix layout of width k if and only if the expanded matrix $x(M)$ does. The expanded matrix has at most two 1's on each row. Any rows with just one 1 can be removed, and if the resulting matrix has pathwidth $(k - 1)$ then the original matrix has a GML of width k. Therefore, GML is essentially the same problem as Pathwidth – any algorithm for Pathwidth can be used for GML and vice versa.

Cache Scheduling further generalizes GML to allow more than one block per column. Unfortunately, when we allow the cost of the k-covering to increase beyond m, the number of columns, the matrix expansion technique cannot be applied. In Cache Scheduling, we can construct an expanded matrix $x(M)$, but a k-covering of the expanded matrix with cost c does not imply the existence of a k-covering for the original matrix of cost c, unless $c = m$. This is because a successful k-covering of $x(M)$ may have covered some of the 1's in expanded rows with more than one block in a column. An obvious example of this is the fact that any expanded matrix has a 2-covering of some cost. But if the original matrix has a row with weight > 2, no 2-covering can exist.

This is a significant problem, because we have no graph construction for Cache Scheduling when there are more than two 1s per row and more than one block per column. Also, techniques for solving or approximating Cache Scheduling with at most two 1s per row will not necessarily apply with more than two 1s per row.

Another important difference between CS and GML is the desired goal: The GML problem is to find a row-permutation that has *minimum width*. However, in CS, given a fixed cache size k it is trivial to determine if a given matrix has a k-covering. The challenge is finding, for some given width k, a k-covering with *minimum cost*.

We summarize some of these results and relationships as follows: If G is a graph with pathwidth k, then,

- G has a path-decomposition D of width k, that is: no set in the path-decomposition D has size greater than $k + 1$
- G is a partial k-path, i.e. G can be embedded into a k-path
- G has a maximum clique size of $k + 1$
- G has an equivalent $m \times n$ row matrix M with two ones per row, and,
- M has pathwidth k, so a permutation of the rows exists such that no row has more than $k + 1$ 1s and *s when 0s between the topmost and bottommost 1 are changed to *s.
- M has a Gate Matrix Layout with width $k + 1$
- M has a Cache Scheduling with a cache size of $k + 1$ and cost m, that is, one block per column and at most $k + 1$ elements covered per row.

As we have shown, MATRIX PATHWIDTH is equivalent to PATHWIDTH, which is NP-complete. GML is also NP-complete. Therefore, CACHE SCHEDULING, as a generalization of both MATRIX PATHWIDTH and GML must also be NP-complete, giving another proof that Cache Scheduling is NP-complete. The first proof is in Section 5.

However, k-PATHWIDTH is not NP-complete, leaving open the possibility that k-CACHE SCHEDULING or other variations of Cache Scheduling may have efficient algorithms.

7. Dynamic Programming Algorithms

There are known dynamic programming approaches to Gate Matrix Layout which, although exponential in running time, offer a very significant improvement over the naïve approach of generating all possible permutations and testing for the optimal arrangement. In Section 7.1, we examine such an algorithm for GML, and in Section 7.2 we give a similar algorithm for Cache Scheduling.

7.1 Dynamic Programming algorithm for GML

Deo, Krishnamoorthy, and Langston [12] present a dynamic programming solution to Gate Matrix Layout which has running time $O(n^2 2^n)$, where n is the number of rows in our definition of the problem. The algorithm returns the minimum k such that GML(M, k) = *true*, and works as follows:

Let S^* be the set of all rows of an $n \times m$ matrix M for which we wish to find an optimal gate matrix layout, and S a subset of S^*. Then the values $c(S, r)$, forming a table, are defined as follows:

$c(S, r) =$ the minimum, over all permutations of S ending with row r, of the maximum number of 1s and *s on any row of S, in a gate matrix layout which has all the rows of S first, and row r the last of these, followed by all the other rows of S^* in some indeterminate order.

Each $c(S, r)$ is the width of an optimal GML of the rows in S, ending with row r, including all the fill-ins on row r necessary to extend the GML to the other rows of the matrix. We can associate with each $c(S, r)$ a permutation of S which has the optimal

width[1]. The optimal gate matrix layout of M is obtained by finding the permutation associated with $\min_{r \in S^*}(c(S^*, r))$ after all the entries in the table have been calculated.

The algorithm iteratively calculates the values $c(S, r)$ and associated permutations for every subset S of S^* and $r \in S$, beginning with the subsets of S^* which have size one. Clearly, $c(\{r\}, r)$ is just the weight of row r. When $|S| > 1$, an optimal permutation of S ending with row r must have some row b preceding r. It is not hard to see that removing row r from the end must give an optimal permutation of $S - \{r\}$ ending with b.

Therefore, $c(S, r)$ can be calculated by considering placing row b immediately before row r, for all $b \in S - \{r\}$, and looking up the values of $c(S - \{r\}, b)$ in the table. The details of this calculation are explained in [12]. The important point is that given the precomputed values of $c(S - \{r\}, b)$, the calculation of $c(S, r)$ can be performed in time $O(|S - \{r\}|)$, the number of rows which might precede row r.

The permutation associated with each $c(S, r)$ can be stored by keeping a single pointer for each $c(S, r)$. This pointer indexes the entry $c(S - \{r\}, b)$, where b is the row immediately preceding r, as described above. Thus the entire permutation can be constructed by following these pointers back to some entry $c(\{a\}, a)$, where a is the first row in the permutation.

Storing the values of $c(S, r)$ for all $S \subseteq S^*$ and $r \in S$ requires $O(n2^n)$ entries, and $O(n^2 2^n)$ comparison operations for the calculations. This gives an $O(n^2 2^n)$ algorithm for Gate Matrix Layout.

7.2 Dynamic Programming Algorithm for Min Cache Scheduling

An essential feature of Gate Matrix Layout exploited by the dynamic programming algorithm above is that each column must be covered by exactly one block. As a

---

[1] In [12], the given algorithm only returns the width of an optimal Gate Matrix Layout. The method given here is a simple extension of their algorithm to also return the permutation.

consequence, during the calculation of $c(S, r)$ the positions of necessary fill-ins on row r are easily calculated, as the relative order of the rows following r is irrelevant.

This is not the case in Cache Scheduling. However, we can generalize this conception of the problem and use a similar technique: by considering subsets of rows with coverings ending at specific cache positions, the order of the remaining rows becomes irrelevant.

Let S^* be the set of all rows of an $n \times m$ matrix M for which we wish to find an optimal cache scheduling, and S a subset of S^*.

An optimal cache scheduling of any set S of rows must end with some cache position p, covering some last row r in the permutation. If $|S| = s$, and $s > 1$, some row immediately precedes row r and is covered by some cache position q, possibly the same as p.

The first $s - 1$ rows in the optimal cache scheduling of S must be an optimal cache scheduling of $S - \{r\}$, constrained to end with cache position q. If this were not so, we could append row r, covered with cache position p, to such an optimal cache scheduling of $S - \{r\}$ and obtain a better cache scheduling of the rows of S; a contradiction.

Therefore, optimal cache scheduling permutations have an optimal substructure that we exploit to create a dynamic programming algorithm. We build a table of values $cs(S, p)$ defined as follows:

$cs(S, p) =$ an optimal cache scheduling permutation of the rows in S, constrained to end with cache position p, which covers some row in S.

We define $cp(S)$ as the set of cache positions which are *relevant* to the set of rows S, that is, all the cache positions which cover some row in S. Then the optimal cache scheduling permutation of M is $\min_{p \in cp(S^*)} \left(cs(S^*, p)\right)$.

We build the table of values from the bottom up, that is, starting with the smallest sets S. When $|S| = 1$, $cs(\{r\}, p)$ is just the sequence $\langle r \rangle$ for all cache positions p which cover row r. So how do we calculate $cs(S, p)$ for $|S| > 1$?

Observe that given any cache scheduling permutation of some set of rows S, ending with some cache position p, we can move all the rows in S which are covered by cache position p to the end of the permutation without increasing the cost of the cache scheduling.

Therefore, to find $cs(S, p)$ we first remove all rows r covered by cache position p from S, obtaining a set of rows S' which is smaller than S. We then find the $q \in cp(S')$ such that $(\text{cost}(cs(S', q)) + w(p, q))$ is minimized, where $w(p, q)$ is the cost of moving from cache position p to cache position q.

We then construct $cs(S, p)$ by appending the rows of S covered by cache position p to the optimal permutation of S' which ends with cache position q. The cost of a k-covering of $cs(S, p)$ is equal to $(\text{cost}(cs(S', q)) + w(p, q))$.

Just as in the gate matrix layout algorithm, each permutation $cs(S, p)$ can be stored as a pointer to the "prefix" permutation $cs(S', q)$, and the cost of an optimal k-covering of $cs(S, p)$. Then each entry in the table takes constant space.

Finally, when building the table from the bottom up, once we have calculated $cs(S, p)$ for all S for $|S| \leq \left\lceil \frac{|S^*|}{2} \right\rceil$ we can calculate the optimal cache scheduling permutation of S^* as follows: The optimal cache scheduling permutation of S^* is $cs(S, p)$ followed by the reverse of $cs(S^* - S, q)$, with S, p, and q chosen so that $cs(S, p) + cs(\overline{S}, q) + w(p, q)$ is minimized, from all subsets S of S^* with size $\left\lceil \frac{|S^*|}{2} \right\rceil$.

To calculate each $cs(S, p)$, $O(nw)$ time is required to determine the rows in S' and the relevant columns for cache positions. There are at most $\binom{m}{k}$ cache positions q to check as possible last cache positions in covering S', and each check takes $O(k)$ time. This gives a calculation time of $O(knwm^k)$.

With $2^{n-1}\binom{m}{k}$ entries to calculate, we have a total running time $O(knwm^{2k}2^{n-1})$, significantly improving the naïve, $O(n!)$ method of testing all possible permutations.

8. Fixed Parameter Tractability

As shown in Section 6.4, the Gate Matrix Layout problem is essentially equivalent to Pathwidth. When k is considered a fixed value, even better algorithms for Treewidth and Pathwidth (and therefore Gate Matrix Layout) are known – for example, Bodlaender's linear time algorithm for k-TREEWIDTH [5].

The existence of efficient algorithms for fixed parameter cases of Pathwidth and Gate Matrix Layout encourage us to search for similar results for Cache Scheduling. As Cache Scheduling generalizes GML to allow more than one block per column, we will introduce similar generalizations for Pathwidth and Treewidth. To this end, we introduce the concept of *defects* in tree and path-decompositions.

We call the resulting problems Almost Pathwidth and Almost Treewidth. We show that the Almost Pathwidth problem is closed under the taking of graph minors for fixed parameters, but Almost Treewidth is not. This implies that Almost Pathwidth has a polynomial time algorithm, but leaves Almost Treewidth as an open problem. We also show that Almost Pathwidth is a special case of Cache Scheduling.

8.1 Almost Pathwidth

We recall the definition of a path-decomposition from Section 2.4:

A *path-decomposition* of a graph $G = (V, E)$ is a list D of subsets X_i of $V(G)$, $1 \leq i \leq l$, which satisfies the following three conditions:

1. The union of all the X_i's = $V(G)$.
2. For all edges (v, w) in $E(G)$, there is a subset X_i in D such that both v and w are contained in X_i.
3. For each vertex $x \in V(G)$, if $x \in X_i$ and $x \in X_k$ with $i \leq k$, then $x \in X_j$ for $i \leq j \leq k$.

The third condition requires that for each vertex $x \in V(G)$ the set of nodes $\{i \mid x \in X_i\}$ forms a single, contiguous sequence. We will now generalize by allowing this condition to be violated d times. That is, d is the number of *defects* allowed in the decomposition.

A *defect* with respect to a vertex v of G in a path-decomposition D is a contiguous sequence of subsets $(X_i, X_{i+1}, \ldots X_j)$ with $v \notin X_k$ for $i \leq k \leq j$, and $v \in X_{i-1}$ and $v \in X_{j+1}$.

A node of a decomposition may be in several distinct defects simultaneously with respect to different vertices, but any two defects with respect to the same vertex must be disjoint.

A *defective path-decomposition* of a graph $G = (V, E)$ is a list D of subsets X_i of $V(G)$, $1 \leq i \leq l$, which satisfies the following two conditions:

1. The union of all the X_i's = $V(G)$.
2. For all edges (v, w) in $E(G)$, there is a subset X_i in D such that both v and w are contained in X_i.

This is exactly the same as the definition of a standard path-decomposition except condition 3 is removed. The definition of the width of a path-decomposition does not change: The *width* of a path-decomposition with (or without) defects is the maximum over $i \in I$ of $|X_i| - 1$.

The total number of defects in a path-decomposition is counted over all vertices in $V(G)$. If a defective path-decomposition actually has no defects, it is in fact a standard path-decomposition as defined in Section 2.4. We generalize and prove this observation as follows:

Lemma 8.1:

> In any defective path-decomposition $D = (X_1, X_2, \ldots X_l)$ of G, let S_v be the number of contiguous, disjoint subsequences of D containing vertex $v \in V(G)$. The number of defects E_v in D with respect to vertex v is $S_v - 1$.

Proof: Any defect with respect to vertex v must, by definition, occur between exactly two contiguous, disjoint subsequences of D containing v. Thus $E_v \leq S_v - 1$.

Any two contiguous, disjoint subsequences of D containing v must be separated by at least one defect with respect to v, or they would not be disjoint. Thus $E_v \geq S_v - 1$.

Therefore $E_v = S_v - 1$.

□

Corollary 8.2:

> A defective path-decomposition with zero defects is in fact a standard path-decomposition.

Proof: If a defective path-decomposition actually has zero defects, then for each v in $V(G)$ there is exactly one contiguous subsequence of D containing vertex $v \in V(G)$. Therefore, condition 3 of the definition of a path-decomposition is satisfied.

□

We observe that there is a tradeoff between the number of defects and the width of the path-decomposition for a fixed graph. We may be able to decrease the number of defects in a path-decomposition by increasing the width, or decrease the width by increasing the number of defects. We will generally try to find path-decompositions with a minimum number of defects for a fixed width, and call a path-decomposition with a minimum number of defects for some fixed width *optimal*. We call the decision problem for path-decompositions with defects Almost Pathwidth:

ALMOST PATHWIDTH

Instance: A graph G and integers k, d.

Question: Does G have a path-decomposition D with at most d defects and width at most k?

The optimization problem we are interested in is defined as follows:

MIN ALMOST PATHWIDTH

Instance: A graph G and integer k.

Question: What is the minimum d such that G has a path-decomposition D with d defects and width k?

Finally, we will consider a version of the problem with both parameters fixed:

(k,d)-ALMOST PATHWIDTH

Instance: A graph G.

Question: Does G have a path-decomposition D with at most d defects and width at most k?

By considering this last problem, with both parameters k and d as fixed, we are able to show (non-constructively) that a polynomial time algorithm for (k,d)-ALMOST PATHWIDTH exists. We define the class of graphs $\boldsymbol{P}(k, d)$ as all graphs G which have a path-decomposition with width at most k and at most d defects.

Lemma 8.3:

$\boldsymbol{P}(k, d)$ is closed under graph minors for all fixed pairs of integers k and d.

Proof: Let G be a graph in $\boldsymbol{P}(k, d)$ with defective path-decomposition D. Let G' be a graph obtained from G by one of the following operations: edge deletion, vertex deletion, or edge contraction. We will show that G' has a path-decomposition with at most d defects and width at most k in each of these cases.

1. D is clearly a path-decomposition of G' when $G' = G$ with an edge removed, and has at most d defects and width k as required.
2. If $G' = G$ with a vertex v deleted, we create a path-decomposition D', equal to D but with v removed from every set X_i in which it occurs. D' is a path-decomposition for G' with width at most k and at most d defects.
3. The only non-trivial case is the contraction of an edge (u, v) in G to obtain G'. We label the vertex resulting from the contraction of (u, v) as w. We construct a path-decomposition D' for G' by replacing each occurrence of u and v in the sets X_i of D with w. It is obvious that D' is a defective path-decomposition of G' and certainly has width at most k.

By Lemma 8.1, E_u, the number of defects in D with respect to u, is at most one less than S_u, the number of contiguous subsequences in D containing u. Thus $E_u + 1 = S_u$. Similarly, $E_v + 1 = S_v$, and in D', $E_w + 1 = S_w$.

Now, one of the disjoint contiguous subsequences of D containing vertex u overlaps one of the subsequences of D containing vertex v, because D must contain a node X_i containing both u and v. Therefore $S_w \leq S_u + S_v - 1$.

Then $E_w + 1 \leq E_u + 1 + E_v$, and $E_w \leq E_u + E_v$. Since u, v, and the new vertex w are the only differences between D and D', this proves that the number of defects in D' is less than or equal to the number of defects in D.

Therefore in all three cases, G' has a path-decomposition D' with at most d defects and width k.

□

Having established that $\boldsymbol{P}(k, d)$ is closed under graph minors we now show that a planar graph is excluded for each k and d.

Lemma 8.4:

$\boldsymbol{P}(k, 0)$ excludes a planar graph for each fixed k.

Proof: Note that $\boldsymbol{P}(k, 0)$ is just the class of k-paths. The following proof is based on one used in [15].

We proceed by induction on k. Our basis is K_4, which is excluded from $\boldsymbol{P}(k, 0)$ for all k less than 3. Given a connected planar graph P_k excluded from $\boldsymbol{P}(k, 0)$ for some k, we can construct a connected planar graph P_{k+1} excluded from $\boldsymbol{P}(k + 1, 0)$ using the following construction:

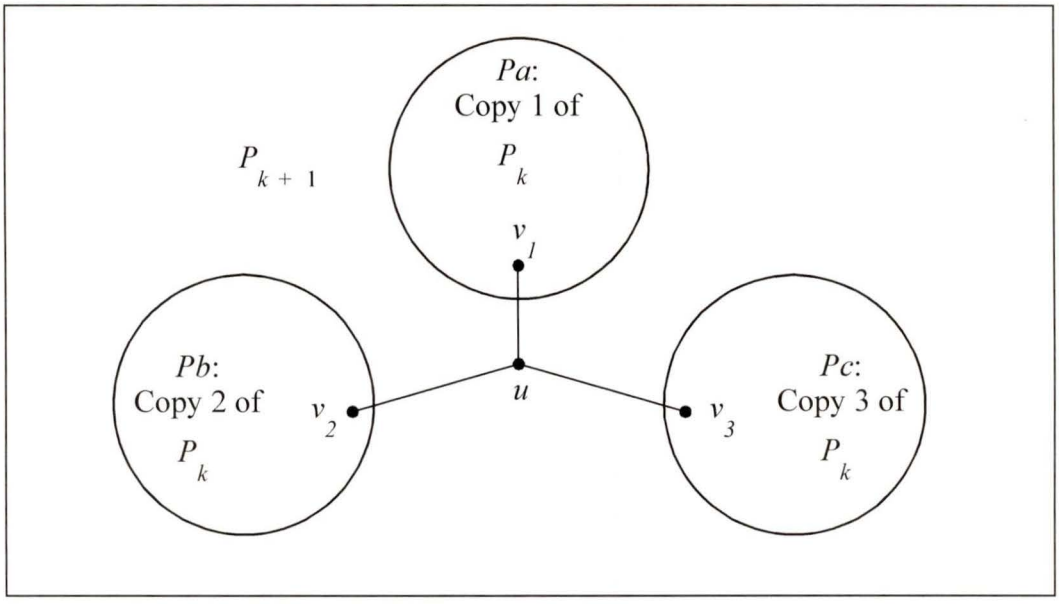


Figure 7: Construction of P_{k+1}

The three vertices v_1, v_2, and v_3 are selected arbitrarily from the three copies of P_k, which we label *Pa*, *Pb*, and *Pc*. We now prove, by contradiction, that there is no defect-free path-decomposition of width k of P_{k+1}, and therefore that P_{k+1} is excluded from $\boldsymbol{P}(k + 1, 0)$. As shown in Section 6.2.1, P_{k+1} has pathwidth k if and only if the corresponding matrix M_{k+1} has pathwidth k.

Suppose M_{k+1} has such a path-decomposition. Then there is a permutation of the rows of M_{k+1} such that no row has more than $(k + 1)$ 1s and *s when all 0s between the topmost and bottommost 1 in each column are changed to *s.

Since each of the copies of P_k is excluded from $\boldsymbol{P}(k,0)$, there is a row r_1 in M_{k+1} representing an edge in *Pa* with at least $k + 1$ 1s and *s in columns representing vertices of *Pa*. Similarly, there are rows r_2 and r_3 in M_{k+1} from *Pb* and *Pc*, each of which has at least $k + 1$ 1s and *s in columns representing vertices in those copies. Without loss of generality, let $r_1 < r_2 < r_3$.

As rows of M_{k+1} directly correspond to edges of P_{k+1}, and columns directly correspond to vertices, henceforth for convenience we will say, for example, "column u" and "row (u,

v_1)" instead of "the column representing vertex u" and "the row representing representing edge (u, v_1)".

Now, all the rows of Pa must occur before r_2. Otherwise, r_2 would have at least one * in columns of Pa because Pa is connected, $r_1 < r_2$, and all 0s between the topmost and bottommost 1 in each column are changed to *s. In particular, there must be a 1 in column v_1 before row r_2. Similarly, all rows of Pc must occur after r_2, and there must be a 1 in column v_3 after row r_2.

Now, if the three rows (u, v_1), (u, v_2), and (u, v_3) are not either all before or all after r_2, then r_2 will need an extra * in column u. This would cause r_2 to have at least $k + 2$ 1s and *s, a contradiction.

But if the three rows (u, v_1), (u, v_2), and (u, v_3) are all before row r_2, then r_2 must have an extra * in column v_3. And if (u, v_1), (u, v_2), and (u, v_3) are all after r_2, then r_2 must have an extra * in column v_1.

Therefore r_2 must have at least $k + 2$ 1s and *s, a contradiction. P_{k+1} cannot have pathwidth k, and P_{k+1} is excluded from $\boldsymbol{P}(k + 1, 0)$.

□

Lemma 8.5:

> $\boldsymbol{P}(k, d)$ excludes a planar graph for each k and d.

Proof: In the proof of the previous lemma we constructed planar graphs P_k excluded from $\boldsymbol{P}(k, 0)$ for all k. Obviously, any k-covering of P_k will require at least 1 defect. A graph $P_{k,d}$ consisting of $d + 1$ distinct copies of P_k will be planar and require at least $d + 1$ defects to cover with width k. Therefore $P_{k,d}$ will be excluded from $\boldsymbol{P}(k, d)$.

□

Theorem 8.6:

> The class of graphs $\boldsymbol{P}(k, d)$ is recognizable in polynomial time for each pair of fixed integers k and d.

Proof: From the previous lemmas, we know that for each fixed pair of integers k and d, $\boldsymbol{P}(k, d)$ is closed under the taking of graph minors, and excludes a planar graph. Therefore, the Robertson-Seymour theorem applies, and a polynomial time algorithm exists to recognize graphs in $\boldsymbol{P}(k, d)$.

□

Corollary 8.7:

> (k,d)-ALMOST PATHWIDTH is in P.

Proof: The graphs $\boldsymbol{P}(k, d)$ are exactly those graphs for which an algorithm for recognizing graphs with (k,d)-ALMOST PATHWIDTH must answer "yes". By Theorem 8.6, such an algorithm exists with polynomial running time.

□

8.1.1 Almost Pathwidth is a special case of Cache Scheduling

Consider Cache Scheduling in $n \times m$ matrices M_2 of maximum row weight 2. We will show that the problem CS(M_2, k, $m + d$) is in P for every fixed pair of integers k and d.

The invertible graph construction method introduced for MATRIX PATHWIDTH in Section 6.2 can be used to construct classes of graphs corresponding to all such M_2, and these graphs are in fact the class of graphs $\boldsymbol{P}(k - 1, d)$, which have defective path-decompositions with width at most $k - 1$ and at most d defects.

CS(M_2, k, $m + d$) is true if M_2 has a row-permutation and k-covering of cost at most $m + d$. (We consider a k'-covering to be a k-covering for all $k' \leq k$.) Since each of the m columns requires at least one block, the parameter d is the number of "extra" blocks allowed to cover the matrix.

Theorem 8.8:

> A graph G is in the class $\boldsymbol{P}(k - 1, d)$ if and only if CS(M, k, $m + d$) is true, when M is the matrix corresponding to G using the graph/matrix conversion method of Section 6.2.

Proof: This proof is similar to the one in Section 6.2.2, showing that Pathwidth $k-1$ is equivalent to Cache Scheduling in $n \times m$ matrices with cache size k and cost m. We extend that proof to show that d extra blocks in the cache scheduling correspond exactly to d defects allowed in the path-decomposition.

Suppose $CS(M, k, m+d)$ is true. Then there exists a row-permutation M' of M with a k-covering of cost at most $m+d$, with at least one block per column. Suppose w. l. o. g. we have such an M' with a k-covering with cost exactly $m+d$ and width exactly k. We assume the elements of M' are labeled $m_{i,v}$ where v is a vertex in G and $1 \le i \le n$. We construct a path-decomposition $D = (X_1, X_2, \ldots X_n)$ of G with width $k-1$ and d defects as follows:

$$X_i = \{v \mid m_{i,v} \text{ is covered by a block of the } k\text{ - covering}\},\ 1 \le i \le n.$$

Each vertex in G is a member of some set X_i, as every vertex corresponds to a column which must have at least one 1. Each edge (u, v) in G corresponds to some row r, so X_i will contain both u and v. For every block in the k-covering in column v, there will be a consecutive sequence of sets $(X_i, X_{i+1}, \ldots X_j)$ containing vertex v, and these sequences will be disjoint as blocks in the same column must be separated by at least one row.

Obviously each X_i has size at most k. Thus the two conditions are satisfied and D is a defective path-decomposition of width $k-1$. By Lemma 8.1 the number of defects with respect to any vertex v is one less than the number of contiguous disjoint subsequences in D containing v. This is one less than the number of blocks in the k-covering in column v. Therefore the total number of defects is equal to the total number of blocks in the covering $-m$. We have $m+d$ blocks in the covering, so exactly d defects.

Therefore G is in $\boldsymbol{P}(k-1, d)$.

Conversely, suppose G is in $\boldsymbol{P}(k-1, d)$, and has a path-decomposition with width at most $k-1$ and at most d defects. Again, we suppose without loss of generality that we have

such a path-decomposition $D = (X_1, X_2, ...X_t)$ with width exactly $k - 1$ and exactly d defects.

We construct a row-permutation of M and k-covering by organizing the rows of M into t groups $g_1...g_t$. We place each row into the first group g_i such that X_i contains both endpoints of the edge in G corresponding to the row in M. We order the rows in each group arbitrarily to obtain a row-permutation M'. We know that for each vertex v in G, there are E_v disjoint contiguous sequences of nodes $X_i...X_j$ in the path-decomposition. We construct a k-covering of M' by creating a block for each of these sequences, beginning at the first row in g_i and extending to the last row in g_j in column v.

It is easy to see from this construction that there are at most $m + d$ blocks in the k-covering, that no row has more than k elements covered, and that every one in the matrix is covered by a block. This implies $CS(M, k, m + d)$ is true.

□

Corollary 8.9:

> For every fixed pair of integers k and d, and all $n \times m$ matrices M with maximum row weight 2, CS(M, k, $m + d$) is solvable in polynomial time.

Proof: As shown above, instances of Cache Scheduling subject to these restrictions are equivalent to graphs in the class $\boldsymbol{P}(k - 1, d)$. In Theorem 8.6 we have shown that these classes of graphs are polynomial-time recognizable for fixed k and d.

□

Unfortunately, this does not give us a very effective way to solve Cache Scheduling problems. This proof is non-constructive, and a separate algorithm is required for each k and d. Algorithms based on this technique typically have very large constant time factors. For example, Bodlaender's algorithm, mentioned in Section 2.4.2, has a constant exponential in the cube of the treewidth of the input graph.

Even worse is the restriction to two 1's per row. The matrix expansion technique described in Section 6.3 for Gate Matrix Layout seems to offer a way around this restriction. Unfortunately, as shown in Section 6.4, this technique does not necessarily work for Cache Scheduling. The question of whether Cache Scheduling with fixed k and d can be shown to be equivalent to a graph property which is closed under graph minors for matrices with row weights > 2 remains open.

8.2 Almost Treewidth

We can generalize Treewidth in the same way that we have generalized Pathwidth. The third condition of the definition of a tree-decomposition requires that for each vertex $x \in V(G)$ the set of nodes $\{i \mid x \in X_i\}$ forms a subtree of T. As in Almost Pathwidth, we allow this condition to be violated d times: the number of *defects* allowed in the decomposition.

A *defect* Z with respect to a vertex v of G in a tree-decomposition is a connected subgraph of T such that for every $i \in V(Z)$, X_i does not contain some vertex $v \in V(G)$. Z must be maximal in the sense that for every vertex $i \in V(Z)$, if vertex j is adjacent to i and $j \notin V(Z)$, then X_j contains v. Less formally, every node in a defect is on a path between two nodes that are not in the defect. As described for Almost Pathwidth, defects are considered with respect to individual vertices.

A *defective tree-decomposition* of a graph $G = (V, E)$ is a pair $D = (S, T)$ with $S = \{X_i \mid i \in I\}$, a collection of subsets X_i of the vertices of G, and $T = (I, F)$ a tree, with one node for each subset of S, such that the following two conditions are satisfied:

1. The union of all the X_i's = $V(G)$.
2. For all edges (v, w) in $E(G)$, there is a subset $X_i \in S$ such that both v and w are contained in X_i.

The *width* of a tree-decomposition with or without defects is the maximum over $i \in I$ of $|X_i| - 1$. If a defective tree-decomposition actually has no defects, it is in fact a standard tree-decomposition as defined in Section 2.4. This is not hard to see, since if for some vertex $x \in V(G)$ the set of nodes $\{i \mid x \in X_i\}$ forms two or more disjoint subgraphs of T,

these subgraphs will be separated in T by defects with respect to the vertex x. If there are no defects, then for every $x \in V(G)$ the set of nodes $\{i \mid x \in X_i\}$ forms a single subgraph of T as required by the normal definition of tree-decompositions.

We call the decision problem for tree-decompositions with defects Almost Treewidth:

ALMOST TREEWIDTH

Instance: A graph G and integers k, d.

Question: Does G have a tree-decomposition D with at most d defects and width k?

We can define other versions of the problem in exactly the same way as we have for ALMOST PATHWIDTH. We would like to show that a polynomial time algorithm exists for (k,d)-ALMOST TREEWIDTH as it does for the Pathwidth variation by considering both parameters k and d as fixed, constant values.

We define the classes of graphs $\boldsymbol{T}(k, d)$ as all graphs G which have a tree-decomposition with width at most k and at most d defects. Unfortunately, $\boldsymbol{T}(k, d)$ is not closed under the taking of graph minors. A simple example shows that if an edge (u, v) is contracted to a vertex w, replacing each instance of u and v with w in nodes of D can result in a decomposition D' with more defects than are in D.

9. Graph Variant of Cache Scheduling

In this section, we give an alternate description of Cache Scheduling as a graph problem we call Cache Graph Scheduling. This highlights similarities to other problems and suggests possible approximation algorithms.

When considering possible approximation algorithms or heuristics for Cache Scheduling, one notices that the problem can be viewed as choosing an ordered list of cache positions which *cover* all of the rows of the matrix.

9.1 Graph of Cache Positions

Consider an instance of Cache Scheduling formed by an $n \times m$ matrix M and integers k and c with $w(M) \leq k$. This models a problem with m pages of memory. We have defined a *cache position* as a particular subset of at most k pages – in this section we assume all cache positions have exactly k pages, except if noted otherwise.

Each row i of M has at most w ones, forming a set $R_i = \{j \mid M_{i,j} = 1\}$. We call R_i the *page set* of row i. We say a cache position *covers* row i if R_i is a subset of that cache position. A list L of cache positions *covers* all the rows covered by any of the cache positions in L, and if L covers all the rows of M, then L *covers* M.

The *cost* ω of moving from one cache position to another is equal to the number of pages that must be loaded from secondary storage. For example, if we have two cache positions $v_1 = \{1, 2, 3, 4\}$ and $v_2 = \{1, 2, 7, 8\}$, then $\omega(v_1, v_2) = 2$. If the cache positions are represented as m digit Boolean strings with k 1s, ω is simply the half the Hamming distance between the strings. This is a symmetric function, with range 0 to k.

For cache scheduling problems with cache size k and memory size m, with $k \leq m$, we can construct an edge-weighted graph $G_{k,m} = (V, E, \omega)$ which represents all $\binom{m}{k}$ possible cache positions as vertices, and the costs of moving between cache positions as weighted

edges using the cost function ω defined above. We denote the set of pages in the cache position represented by vertex v as $s(v)$.

A path P in $G_{k,m}$ specifies a list L of cache positions, so for convenience we will say that P *covers* the rows covered by L. The cost of path P, $\omega(P)$, is the sum of the costs of the edges in P.

Throughout the following we assume that all problem instances will require at least k cache load operations, although small problem instances exist for which this is not true. Such small problems can be solved with at most one load per page.

We define the problem as follows:

CACHE GRAPH SCHEDULING (CGS)

Instance: A list $\boldsymbol{R} = (R_1, R_2, \ldots R_n)$, and integers m, c and k, with $R_i \subset \{1, 2, \ldots m\}$ and $|R_i| \leq k$, for $1 \leq i \leq n$.

Question: Does a path $P = (v_1, v_2, \ldots v_t)$ exist in $G_{k,m}$ with $\omega(P) \leq c - k$, such that for $\forall$ $i \in \{1 \ldots n\}$, $R_i \subset s(v_j)$ for some $j \in \{1 \ldots t\}$?

The following theorem essentially states that $G_{k,m}$ has a path covering the rows of M with weight $\leq c - k$ if and only if M has a row permutation and k-covering with cost $\leq c$. This establishes the connection between CS and CGS:

Theorem 9.1:

> An instance of CGS ($\boldsymbol{R}$, m, c, k) has a solution if and only if the Cache Scheduling problem CS(M, k, c) = *true*, where M is an $n \times m$ matrix and each $R_i \in \boldsymbol{R}$ is the row set for row i of M.

Before we prove this theorem, we note that the allowed weight of the path in $G_{k,m}$ is $c - k$ because the cost of loading the first k blocks into the cache is not included in the weight of the path. Since we assume no problem can be solved with cost less than k, $c - k$ is always non-negative.

Figure 8 shows a k-covering and the corresponding path of cache positions. Note that at the beginning and end of a k-covering, fewer than k pages may be required in cache positions. In Figure 8, this occurs on the first and last rows. In k-coverings we have found it convenient to assume that all blocks begin and end with 1s, but in the cache graph it is convenient to assume that all cache positions have exactly k pages. We cannot satisfy both assumptions simultaneously while maintaining a one-to-one correspondence between k-coverings and paths in $G_{k,m}$, but there is no real difficulty as we can easily add pages to fill in cache positions from a k-covering, or remove unnecessary pages from cache positions in a path in $G_{k,m}$. In Figure 8 we have filled in the cache positions by arbitrarily choosing pages, so the block in the first column of M begins with a 0 and the block in the fourth column ends with a 0.

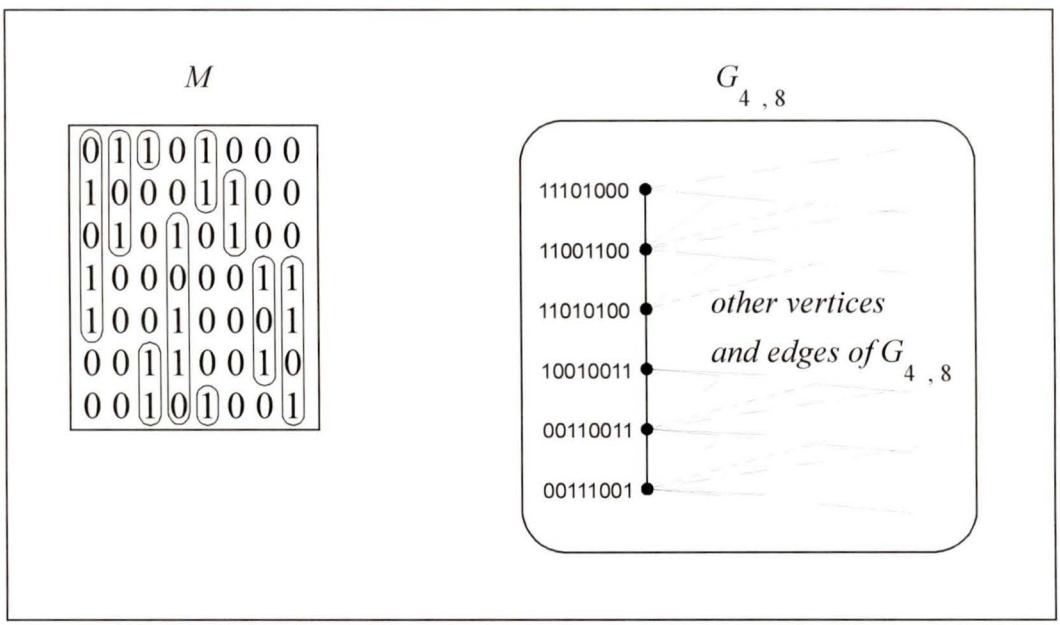


Figure 8: A k-covering and a corresponding path of cache positions

Proof of Theorem 9.1:

The forward implication is obvious from the construction of $G_{k,m}$ and $\boldsymbol{R}$: Given a row permutation and k-covering, we can determine the cache position at each row, list these cache positions in order and remove consecutive duplicates to obtain a path in $G_{k,m}$ which

covers the rows of M. The weight of this path, plus k for loading the first cache position, equals the total number of cache load operations, which is equal to the cost of the k-covering.

The reverse implication is also straightforward. Given the requisite path $P = (v_1, v_2, \ldots v_t)$, we partition the rows of M into a list of groups S_i, $1 \leq i \leq t$. Each row j is placed into the first group S_i such that $R_j \subseteq s(v_i)$. We obtain a row permutation of M by taking the groups S_i in order, ordering rows within groups arbitrarily.

We construct a k-covering by starting between 1 and k blocks at the first row of each group S_i, so that cache position v_i covers the rows in S_i. If $\omega(v_i, v_{i+1}) = x$, then we will have to start x blocks at the first row in group S_{i+1}, while $k - x$ blocks will continue from group S_{i+1} into group S_{i+1}. It is easy to see that this produces a k-covering with cost equal to the weight of the path plus k for the first cache position.

□

In following sections, we will consider approximation algorithms for CGS. Unfortunately, the existence of a good approximation algorithm for arbitrary instances of CGS would not necessarily imply such an algorithm for CS, because of the blowup in the problem size in the graph construction. With $\binom{m}{k}$ vertices, even moderately sized problem instances will produce extremely large graphs. However, for small or fixed k this may be a useful approach.

Even better, we may be able to avoid constructing the entire graph by using some sort of implicit representation. Edge weights can easily be computed if needed so need not be stored. We may also be able to reduce the size of the graph under consideration. For example, if we only consider cache positions that cover at least one row, the number of vertices may be reduced by several orders of magnitude. Furthermore, specific algorithms based on this idea may be able to avoid constructing the graph in other ways.

9.1.1 Properties of $G_{k,m}$

In this section we describe some basic properties of the graph $G_{k,m}$ and of potential solutions to instances of CGS.

A *metric space* has the following properties, where ω is the distance function between any points u, v and w in the space:

1. $\omega(v, u) \geq 0$, and $\omega(v, u) = 0$ if and only if $v = u$
2. $\omega(v, u) = \omega(u, v)$ for all u, v
3. $\omega(v, w) \leq \omega(v, u) + \omega(u, w)$ for all u, v, w

Lemma 9.2:

> $G_{k,m}$ is a metric space.

Proof: All three properties are obvious from the definition of ω.

□

Note that if the vertices of the m-dimensional Boolean hypercube B_m are labeled with bitstrings, the vertices of $G_{k,m}$ are the subset of $V(B_m)$ which have exactly k ones in their bitstrings.

Lemma 9.3:

> For any edge (u, v) in $G_{k,m}$, there is a path of length $\omega(u, v)$ of edges of weight 1 between u and v.

Proof: If $\omega(u, v) > 1$, then there is a page p_u in u which is not in v, and a page p_v in v but not in u. If cache position $w_1 = (v \setminus \{p_u\}) \cup \{p_v\}$, then $\omega(v, w_1) = 1$ and $\omega(u, w_1) = \omega(u, v) - 1$. This can be repeated as necessary to obtain $w_2, w_3, \ldots w_{\omega(u, v)-1}$, forming a path

$$(u, w_1, w_2, \ldots w_{\omega(u, v)-1}, v)$$

of $\omega(u, v)$ weight 1 edges.

□

By repeated applications of Lemma 9.3 to any path P in $G_{k,m}$, we can obtain a path with cost equal to $\omega(P)$, which includes all the vertices of P, but has only edges of weight 1. Furthermore, it is easy to see that all these edges are also edges in the Boolean hypercube B_m. In fact, any edge (v_x, v_y) in $G_{k,m}$ with weight $w > 1$ is equivalent to $\sum_{i=1}^{w} i^2$ distinct paths of weight 1 edges between v_x, and v_y, passing through any of $\binom{2k}{k}$ vertices.

Therefore, if all the vertices of B_m which do not have exactly k 1s in their bitstrings are deleted, the graph which remains is connected, and is in fact $G_{k,m}$ with all edges of weight > 1 deleted. We will call this graph $G'_{k,m}$. Any path in $G_{k,m}$ representing a solution to an instance of CGS can be converted to a path in $G'_{k,m}$ which is also a solution. Consequently, we can work with $G'_{k,m}$ if that is more convenient.

We will now briefly examine some previously studied problems that are similar to Cache Graph Scheduling.

9.2 Errand Scheduling

Cache Graph Scheduling is similar to the Errand Scheduling problem, defined in [44].

ERRAND SCHEDULING (ESP):

Instance: U is a set of size m. G is an edge-weighted graph (V, E). Associated with each vertex v in $V(G)$ is a set $S_v \subseteq U$.

Problem: Compute a minimum weight partial tour $T = (v_1, v_2, \ldots v_t)$ through G such that the union over $1 \leq i \leq t$ of the sets S_{v_i} is equal to U.

U is the set of errands. The vertices of G represent locations, and the edge weights represent distances between the locations. Each location v is associated with a set S_v of "errands" that can be performed there. A solution to ESP gives a minimum length tour that allows all the errands to be performed.

Despite the obvious similarity between ESP and CGS, we note that ESP solutions are cycles, or tours, while CGS solutions are paths. However, the maximum difference between the cost of an optimal tour and an optimal path is bounded by the weight of a single edge in the graph, which is k in $G_{k,m}$. Aside from that difference, Errand Scheduling is a generalization of the Cache Graph Scheduling problem to arbitrary edge-weighted graphs. Alternatively, CGS is a restriction of ESP to the special edge weighted graphs $G_{k,m}$.

Unfortunately, known approximation methods for Errand Scheduling on arbitrary graphs do not result in very good solutions to CGS, and better approximation methods for restricted classes of graphs (such as trees) are not applicable. In [44], an algorithm based on a linear programming relaxation of an integer programming version of Errand Scheduling is given. If each errand can be performed at most p vertices, this algorithm can approximate ESP to within $3p/2$. In Cache Scheduling, a row with weight w can be covered by exactly $\binom{m-w}{k-w}$ cache positions. Therefore, this algorithm approximates an optimal solution to within $O((m\text{-}w)^{k\text{-}w})$. This is probably not a good enough approximation to be useful. However, it is possible that the algorithm actually performs better on average than this analysis indicates. Alternatively, a different approach based on the same concept of integer programming relaxation might be more successful.

9.3 Traveling Salesman with Neighborhoods

Traveling Salesman with Neighborhoods (TSPN) is also known as the Geometric Covering Salesman Problem. Cache Graph Scheduling is analogous to a graph version of the geometric problem of TSPN. In this section we consider the possibility of applying known results for TSPN to Cache Graph Scheduling. We do not know of a way to do this, and we briefly discuss some of the difficulties encountered.

TRAVELLING SALESMAN WITH NEIGHBORHOODS

Instance: S, a collection of n possibly overlapping simple polygons ("neighborhoods") in the plane.

Problem: Find a shortest tour that visits (intersects) each of the neighborhoods.

This is a generalization of Euclidean TSP (from points to neighborhoods) and so is NP-hard. An $O(\log n)$ approximation algorithm for TSPN is given by Mata and Mitchell in [33]. An earlier result by Arkin and Hassin in [1] gives an $O(1)$ approximation ratio for the case of "round", approximately equal-sized neighborhoods such as unit disks.

Given an instance of CGS, the sets of cache positions which cover any particular row are like "neighborhoods". These neighborhoods may overlap, as any particular cache position may cover several rows. We want to find a shortest path that visits a point in each neighborhood, thereby covering each row. Like ESP, TSPN solutions are tours, while CGS solutions are paths, but the cost of minimum weight tours and paths differ by at most k in $G_{k,m}$ so this is not a significant problem.

The difficulty in applying known results for TSPN to CGS is that TSPN is defined for a planar geometric space, with a Euclidean distance metric. Known results for TSPN appear to depend heavily on geometric properties of the problem. If we wish to apply these results to CGS, we must adapt them to a space quite different from the plane, and we have not yet found any way to do so.

10. Two Approximation Algorithms

In this section we present two approximation algorithms for Min Cache Scheduling. The first is based on Christofides' algorithm [10] for the Travelling Salesman problem. This approach computes cost from row to row, rather than between cache positions. In effect, this ignores spaces in the cache that could be used to store pages for later use. The second algorithm is based on a greedy approximation algorithm for Minimum Cover [17]. This algorithm ignores the variations in costs of moving between different pairs of cache positions, treating them all equally.

10.1 Traveling Salesman

In this section, we consider an approximation of Cache Scheduling based on Christofides' approximation algorithm for the Travelling Salesman problem:

TRAVELLING SALESMAN (TSP) [17]

Instance: A set C of m cities, a distance $d(c_i, c_j)$ for each pair of cities $c_i, c_j \in C$, a positive integer B.

Question: Is there a tour of C having length B or less, i.e. a permutation $\langle c_{\pi(1)}, c_{\pi(2)}, \ldots c_{\pi(m)} \rangle$ of C such that

$$\left(\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)})\right) + d(c_{\pi(m)}, c_{\pi(1)}) \leq B\,?$$

It is convenient to represent an instance of TSP as a complete edge weighted graph G, with vertices representing cities, and edge weights representing distances. Then the TSP question is: Does a Hamiltonian cycle of weight $\leq B$ exist in G?

10.1.1 Approximation algorithm for Min Hamiltonian Path

Christofides' [10] approximation algorithm for TSP gives a Hamiltonian Cycle with weight at most $\frac{3}{2}$ times the minimum, and only requires that the edge weights of the

graph obey the triangle inequality. We use this algorithm to approximate a minimum weight Hamiltonian path of a complete, edge-weighted graph G as follows:

Run Christofides' to obtain a tour T of G of weight $\omega(T)$, with $\omega(T) \leq \frac{3}{2}\omega(T_o)$. T_o is an optimal Hamiltonian cycle. Remove the heaviest edge e from T to obtain a path P of weight $\omega(P) = \omega(T) - \omega(e)$. Any Hamiltonian path in G can be closed to form a cycle by adding a single edge of cost at most x, where x is the weight of the heaviest edge in G. Therefore we obtain $\omega(P) \leq \frac{3}{2}\omega(P_o) + \frac{3}{2}x - \omega(e)$, where P_o is a minimum weight Hamiltonian Path in G.

10.1.2 Application to Cache Scheduling

Given an instance of Cache Scheduling consisting of an $n \times m$ matrix M and a cache size k, we can create an edge-weighted graph G with a vertex for each row, and edge weights equal to half the Hamming distances between rows. This is essentially the reverse of the construction used in Section 5 where Hamiltonian Path was reduced to Cache Scheduling.

If we apply the approximation algorithm described above for Min Hamiltonian Path to G, we will obtain a Hamiltonian path in G which is equivalent to a permutation of the rows of M. The edge weights of G will be in the range $[1...w]$.

How does this approach fare as an approximation algorithm for Cache Scheduling? Since the effect of the cache is ignored, we may expect that as the cache size increases relative to the row weight, the accuracy decreases.

When the cache size k equals w, the cost of a k-covering of any row permutation is equal to the sum of the edge weights between the vertices, plus w for the first row. This is the weight of the Hamiltonian Path + w, so if C is the cost of the resulting k-covering and C_o is the cost of an optimal k-covering, we have

$$C - w \leq \frac{3}{2}(C_o - w) + \frac{3}{2}x - e$$

In the worst case, $e = 1$ and $x = w$, so

$$C \leq \frac{3}{2} C_o + w - 1$$

When $k > w$, the actual cost of moving from row v to row u in an optimal k-covering may be lower than the edge weight between the vertices for those rows.

The exact relationship between C and C_o when $k > w$ remains an open problem. It seems likely that as k becomes relatively larger than w, the accuracy of this algorithm will decrease.

10.2 Minimum Cover

The minimum cover problem is as follows:

MINIMUM COVER

> Given a collection C of subsets of S, and a positive integer k, does C contain a cover for S of size k or less, i.e. a subset C' of C with $|C'| \leq k$ such that every element of S is contained in at least one member of C'?

MINIMUM COVER is NP-complete even if each subset in C has size ≤ 3 [17].

Minimum Cover has an simple greedy approximation algorithm [22], which repeatedly chooses one set at a time which covers the most elements among the uncovered ones. Johnson and Lovász [22], [32] proved that this algorithm finds a result within $H(d)$ of optimal, where d is the size of the biggest set. $H(d) = \sum_{i=1}^{d} \frac{1}{d}$, and is bounded by 1 + $\log(d)$. In Cache Scheduling, we are looking for a minimum weight covering path instead of a minimum size covering set. However, we can approximate Min Cache Scheduling with a similar algorithm.

10.2.1 Application to Min Cache Scheduling

We can apply the greedy strategy for Minimum Cover to approximate MIN CACHE SCHEDULING. We repeatedly choose cache positions that cover a maximum number of uncovered rows until all rows are covered. Then we approximate an optimal ordering of the chosen cache positions by computing an approximately minimal cost path in the cache position graph which spans the chosen positions.

This ordering of cache positions can be obtained by running Christofides' TSP algorithm to obtain a tour, and then deleting an edge of maximum weight.

An optimal Cache Scheduling must use at least as many cache positions as a minimum size covering set of cache positions, and these cache positions must have a distance of at least 1 between them as they are distinct. The cache positions in a minimum size covering set can be a maximum distance of k from each other. Therefore the difference in cost between an optimal cache scheduling and a minimum size covering set of cache positions is at most $k - 1$ times the number of elements in a minimum size covering.

Each row is covered by $\binom{m-w}{k-w}$ different cache positions. This gives an upper bound of $(k-1)(1 + \log((m-w)^{k-w})$ times the cost of an optimal cache scheduling. As $m \leq nw$, this algorithm gives results which are $O(k^2 \log nw)$ times optimal.

We expect that this algorithm will perform best when a single cache position may cover many rows.

11. Cache Scheduling with b blocks per column

Throughout this thesis we have defined the cost of a k-covering as the total number of blocks in that covering. We now consider a variation of Cache Scheduling in which the cost is measured as the maximum number of blocks in any one column.

b-BLOCK PER COLUMN CACHE SCHEDULING (CS-b)

Instance: A Boolean matrix M, an integer k with $w(M) \leq k$, and an integer b.

Question: Can we compute M^* and C such that M^* is a permutation of the rows of M, and C is a k-covering of M^* with at most b blocks per column?

This variation of the problem, like the standard definition of Cache Scheduling, has a close relationship to Pathwidth and Gate Matrix Layout. If $b = 1$, CS-b with width k is exactly the same as GML for width k, and if in addition the row weights are uniformly two, this is the same as Pathwidth with width $k - 1$.

There is also a relationship between these problems when $b > 1$:

Consider a graph G with n vertices and m edges. Suppose we can create b subgraphs G_1, G_2, … G_b of G such that:

- $V(G_i) = V(G)$ for $1 \leq i \leq b$
- $\bigcup_{1 \leq i \leq b} E(G_i) = E(G)$
- Each G_i has pathwidth k

We call this an *edge-partitioning* of width k of G into b subgraphs. Consider the matrix M corresponding to G using the construction method for Matrix Pathwidth introduced in Section 6.2.1 (columns for vertices and rows for edges).

Theorem 11.1:

> If G has an edge-partitioning of width k into b subgraphs, and M is the matrix equivalent to G using the construction of Section 6.2.1, then M must have a b-block per column, width $k + 1$ cache scheduling.

Proof: Given the edge-partitioning of G and path-decompositions for each G_i we can obtain a cache-scheduling for M as follows. We partition the rows of M into groups according to which subgraph G_i the corresponding edge is in. Each of these groups certainly has a 1-block per column width $k + 1$ cache scheduling, as shown in Section 6.2. We can place the b row-permutations for each group in any order to obtain a b-block per column, width $k + 1$ cache scheduling of all the rows of M.

□

We give an example with the graph K_5, which has an edge-partitioning into two subgraphs with pathwidth 2. This is equivalent to a width 3, two block per column cache scheduling.

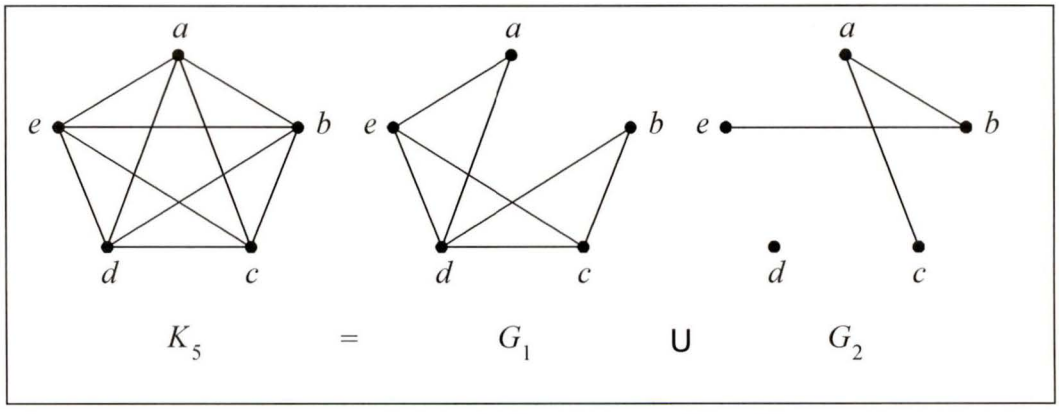
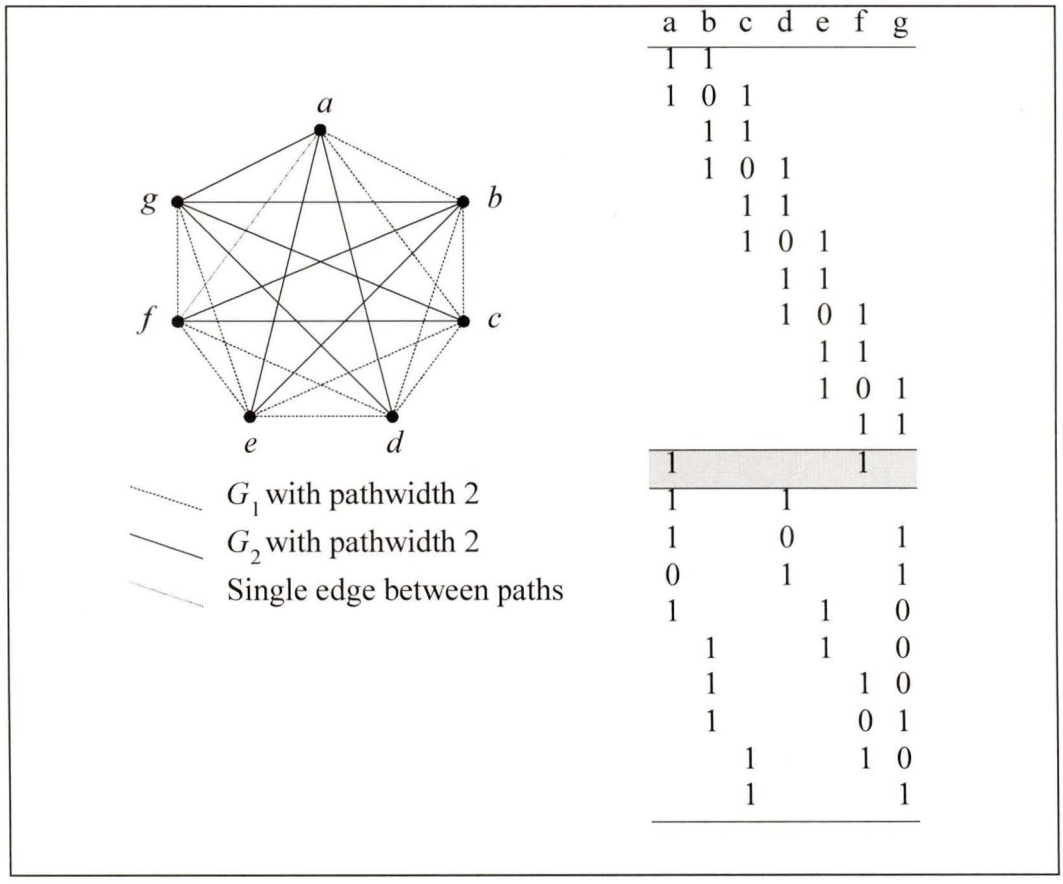


Figure 9: Partitioning K_5 into two subgraphs of pathwidth 2

All complete graphs up to K_6 can be covered as two 2-paths. One might suspect that the converse of Theorem 11.1 might be true, that is, for any matrix with a b-block per column cache scheduling of width k, the equivalent graph has an edge-partitioning into b subgraphs with pathwidth $k - 1$. This is not the case however. For example, the matrix

equivalent to K_7 has a two block per column, width 3 cache scheduling, but K_7 has no edge-partitioning into two 2-paths.

Figure 10 shows a two block per column Cache Scheduling of the matrix corresponding to K7. The shaded row corresponds to the single edge which cannot be added to either of the two 2-paths. (0's are only shown in the matrix if they are covered by blocks).



a	b	c	d	e	f	g
1	1					
1	0	1				
	1	1				
	1	0	1			
		1	1			
		1	0	1		
			1	1		
			1	0	1	
				1	1	
				1	0	1
					1	1
1					1	
1			1			
1			0			1
0			1			1
1				1		0
	1			1		0
	1				1	0
	1				0	1
		1			1	0
		1				1

Figure 10: K_7 cannot be partitioned into two 2-paths.

As Figure 10 shows, extra rows may appear between two submatrices corresponding to *k*-paths without violating the two-blocks-per-column restriction. Therefore, finding an optimal *b*-block per column cache scheduling, even in a matrix with two ones per row, is not the same problem as finding an edge-partitioning into a minimal number of *k*-paths.

12. Conclusions

In this section, we present conclusions, conjectures, and suggestions for further research. We begin with the complexity of special cases of Cache Scheduling.

Recall that we use variables as follows: Given a matrix M, n and m are the number of rows and columns respectively. We use k to refer to the cache size, and w to refer to the maximum row weight of a matrix, which we usually assume is uniform. The allowed cost of the cache scheduling is c blocks.

Cache Scheduling with $k = 3$ and $w = 2$ is the smallest case that is neither trivial nor equivalent to a previously studied problem. Cache Scheduling with $k = 3$ and $w = 2$ is NP-complete, as has recently been proved by Irani, Dillencourt, and Halem [21]. If $w = 1$, Cache Scheduling is trivial for any k. If each row is unique, the permutation of rows is irrelevant and an optimal k-covering simply covers k rows with the first cache position and then modifies the cache position one page at a time to cover the remaining rows.

In the NP-completeness proof of Chapter 5, it is shown that Cache Scheduling with $k = w$ is equivalent to Hamiltonian Path. This is NP-complete, but leads to the application of approximation algorithms for Min Hamiltonian Path, as shown in Section 10.1.1.

As shown in sections 6.2.2 and 6.3, Pathwidth and Gate Matrix layout are special cases of Cache Scheduling. This implies that Cache Scheduling is NP-complete for $m = c$.

We conjecture that Cache Scheduling is in fact NP complete for all general cases with $2 \leq w \leq k$, unless parameters are fixed. Therefore we do not expect to find polynomial time algorithms for the general Cache Scheduling problem. With these expectations, the $O(knwm^{2k}2^{n-1})$ algorithm presented in Section 7.2 is useful as it significantly improves on the $O(mnn!)$ naïve algorithm of generating all permutations of rows and checking them with the algorithm of Chapter 4.

However, as shown in Chapter 8, we may be able to do much better when the cache size k and allowable number of defects d are considered constants. The Almost Pathwidth

problem, which generalizes Pathwidth in the same way that Cache Scheduling generalizes Gate Matrix Layout, is closed under graph minors. This provides a non-constructive proof that a polynomial time algorithm exists for Almost Pathwidth.

We believe a constructive algorithm can be given for Almost Pathwidth for fixed k and d, possibly by modifying a known algorithm for Pathwidth. This is the current goal of our ongoing research.

A related avenue of research that we have not yet explored is modifying some of the known approximation algorithms for Pathwidth to solve Almost Pathwidth and Cache Scheduling. This might give more practical results, since for any actual implementation of a Cache Scheduling algorithm, a fast running time would be more important than an exact answer. The simple approximation algorithms given in Section 9.2 and Chapter 10 may have insufficient accuracy for implementation in real systems.

Considering actual implementations leads to a related problem. Consider the dynamic Cache Scheduling problem, when new processes are constantly being added to a running system. This raises interesting questions, such as how much time is required to maintain an optimally ordered queue of processes as processes are added? It seems that any solution to this problem would have very wide practical application.

In summary, we believe the Cache Scheduling problem and its variants are very interesting from both theoretical and practical standpoints, with many potentially fruitful avenues of research.

13. References

[1] ARKIN, E. M. AND R. HASSIN [1994]. Approximation algorithms for the Geometric Covering Salesman Problem. *Discrete Applied Mathematics 55*, pp. 197-218.

[2] ARNBORG, S., D. G. CORNEIL, AND A. PROSKUROWSKI [1987]. Complexity of finding embeddings in a *k*-tree. *SIAM J. Alg. Disc. Meth. 8*, pp. 277-285.

[3] ARNBORG, S., J. LAGERGREN AND D. SEESE [1991]. Easy problems for tree-decomposable graphs. *J. Algorithms 12*, pp. 308-340.

[4] BODLAENDER, H. L. [1993]. A tourist guide through treewidth. *Acta Cybernetica 11*, pp.1-23.

[5] BODLAENDER, H. L. [1996]. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput. 25*, pp. 1305-1317.

[6] BODLAENDER, H. L. AND T. KLOKS [1996]. Efficient and constructive algorithms for the pathwidth and treewidth of graphs, *J. Algorithms 21*, pp. 358-402.

[7] BODLAENDER, H. L., J. R. GILBERT, H. HAFSTEINSSON, AND T. KLOKS [1995]. Approximating treewidth, pathwidth, and minimum elimination tree height. *J. Algorithms 18*, pp. 238-255.

[8] BODLAENDER, H.L. [1988]. Dynamic programming algorithms on graphs with bounded treewidth. *Proceedings of the 15th International Colloquium on Automata, Languages, and Programming, Springer-Verlag, Lecture Notes in Computer Science 317*, pp. 105-119.

[9] BOOTH, K.S AND G. S. LUEKER [1976]. Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using *PQ*-Tree Algorithms, *Journal of Computer and System Sciences 13*, pp. 335-379.

[10] CHRISTOFIDES, N. [1976]. *Worst-case analysis of a new heuristic for the travelling salesman problem.* Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA.

[11] CORMEN, T. H., C. E. LEISERSON, AND R. L. RIVEST [1989]. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts.

[12] DEO, N., M. S. KRISHNAMOORTHY AND M. A. LANGSTON [1987]. Exact and approximate solutions for the gate matrix layout problem. *IEEE Transactions on Computer Aided Design 6,* pp. 79-84.

[13] ELLIS, J. A., I. H. SUDBOROUGH, AND M. A. LANGSTON [1987]. Graph separation and search number, Technical report DCS-66-IR, University of Victoria.

[14] ELMASRI, R. AND S. NAVATHE [1989]. *Fundamentals of database systems.* Benjamin/Cummings Publishing Company, Inc. Redwood City, California.

[15] FELLOWS, M. R. AND M. A. LANGSTON [1987]. Nonconstructive advances in polynomial-time complexity, *Information Processing Letters 26,* pp. 157-162

[16] FULKERSON, D. R. AND O. A. GROSS [1965]. Incidence matrices and interval graphs, *Pacific J. Math 15*, pp. 835-855.

[17] GAREY, M. R. AND D. S. JOHNSON [1979]. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company.

[18] GHOSH, S. P. [1972]. File organization: the consecutive retrieval property, *Comm. ACM 9*, pp. 802-808.

[19] HARARY, F. [1969]. *Graph Theory*. Addison-Wesley Publishing Company, Reading, Massachusetts.

[20] HOCHBAUM, D. S. [1997] (editor). *Approximation algorithms for NP-hard problems.* PWS Publishing Company, Boston, MA.

[21] IRANI, S., M. DILLENCOURT, AND D. HALEM [1997]. Unpublished draft.

[22] JOHNSON, D. S. [1974]. Approximation Algorithms for Combinatorial Problems. *J. Comput. System Sci. 9*, pp. 256-278.

[23] KARP, R. M. [1972]. Reducibility among combinatorial problems, in *Complexity of Computer Computations*. R. E. Miuller and J. W. Thatcher (eds.), Plenum Press, New York, pp. 85-103.

[24] KASHIWABARA, K. AND T. FUJISAWA [1979]. An NP-complete problem on interval graphs. *Proc. IEEE Symp. On Circuits and Systems*, pp. 82-83.

[25] KIMBREL, T. [1997]. Parallel Prefetching and Caching. *University of Washington CSE technical report* 97-07-03.

[26] KLOKS, T. [1993]. *Treewidth.* Doctorate Thesis, Department of Computer Science, University of Utrecht, Utrecht, Netherlands.

[27] KLOKS, T. [1994]. *Treewidth. Computations and Approximations*, Volume 842 of Lecture Notes in Computer Science. Springer-Verlag, Berlin.

[28] KOU, L. T. [1977]. Polynomial Complete Consecutive Information Retrieval Problems, *SIAM J. Computing. Vol. 6 No. 1,* pp. 67-75.

[29] LEVCOPOULOS, C. AND A. LINGAS [1984]. Bounds on the length of convex partitions of polygons. *Proc. 4th Conf. Found. Software Tech. Theoretical Computer Science*, LNCS Vol. 181, pp. 279-295. Springer-Verlag, Berlin.

[30] LI, K. [1996]. Personal communication from Kai Li to Valerie King.

[31] LOPEZ, A. AND H. LAW [1980]. A dense gate matrix layout for MOS VLSI. *IEEE Trans. Electron. Devices 27,* pp. 1671-1675.

[32] LOVÁSZ, L. [1975]. On the Ratio of Optimal Integral and Fractional Covers. *Discrete Math. 13*, pp. 383-390.

[33] MATA, C. S. AND J. S. B. MITCHELL [1995]. Approximation Algorithms for Geometric Tour and Network Design Problems. *Proc. 11th ACM Symp. Computational Geometry*, pp. 360-369.

[34] MATOUSĚK, J. AND R. THOMAS [1991]. Algorithms for finding tree-decompositions of graphs, *J. Algorithms 12*, pp. 1-22.

[35] MÖHRING, R. H. [1990]. Graph problems related to gate matrix layout and PLA folding. In E. Mayr, H. Noltemeier, and M. Syslo (Eds.), *Computational Graph Theory, Computing Suppl. 7*, pp. 17-51. Springer-Verlag, Berlin.

[36] PAPADIMITRIOU, C. H. [1994]. *Computational Complexity*. Addison-Wesley Publishing Company, Reading, Massachusetts.

[37] PHILBIN, J., J. EDLER, O. J. ANTHUS, C. C. DOUGLAS, AND K. LI [1996]. Thread Scheduling for Cache Locality, *Proceedings of the ACM Architectural Support for Programming Language and Operating Systems, October 1996.*

[38] ROBERTSON, N. AND P. SEYMOUR [1983]. Graph Minors. I. Excluding a forest. *J. Comp. Theory Series B 35*, pp. 39-61.

[39] ROBERTSON, N. AND P. SEYMOUR [1985]. Disjoint paths - A survey. *SIAM J. Alg. Disc. Meths. 6*, pp. 300-305

[40] ROBERTSON, N. AND P. SEYMOUR [1985]. Graph Minors - a survey. In I. Anderson (Ed.), *Surveys in Combinatorics*, pp. 153-157. Cambridge University Press, London.

[41] ROBERTSON, N. AND P. SEYMOUR [1986]. Graph Minors. II. Algorithmic aspects of tree-width. *J. Algorithms 7*, pp. 309-322.

[42] ROBERTSON, N. AND P. SEYMOUR [1996]. Graph Minors. XV. Giant steps. *J. Comp. Theory Series B 68*, pp. 112-148

[43] SANDERS, D. P. [1996]. On linear recognition of tree-width at most four. *SIAM J. Disc. Meth. 9,* (1) pp. 101-117.

[44] SLAVIK, P. [1996]. The Errand Scheduling Problem. `http://www.math.buffalo.edu/~slavik/papers/esp.ps`

Vita

Surname:	Hoffman	Given Names:	Torrey Luke
Place of Birth:	Los Angeles, California, United States of America		

Educational Institutions Attended:

University of Victoria	1990 - 1997

Degrees Awarded:

B.Sc. (Honors)	University of Victoria	1995

Honors and Awards:

Natural Sciences and Engineering Research Council PSG-A	1995 - 1997
British Columbia Advanced Systems Institute Scholarship	1995
Canada Scholar Award	1990 - 1995
Hi-Tech Systems Ltd. Scholarship	1994
University of Victoria President's Entrance Scholarship	1990
British Columbia Provincial Academic Scholarship	1990

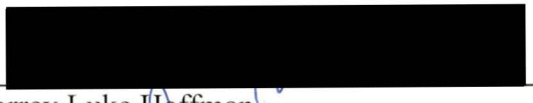
Partial Copyright License

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

Cache Scheduling

Author:



Torrey Luke Hoffman

April 7, 1998