October 1990

Report No. STAN-CS-90-1338

PB96-149885

Triangularization: A Two-Processor Scheduling Problem

by

Ramsey W. Haddad

DISTRIBUTION STATEMENT K
Approved for public release;
Distribution Unlimited

DTIC QUALITY INSPECTED

Department of Computer Science

Stanford University

Stanford, California 94305

19970610 097

TRIANGULARIZATION: A TWO-PROCESSOR SCHEDULING PROBLEM

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Ramsey Haddad
May 1990

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Donald Knuth
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Ernst Mayr

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Andrew Goldberg

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

We explore the following matrix problem: Given an $n \times n$ boolean matrix, is there a permutation of the rows and a permutation of the columns such that the resulting matrix is lower triangular? We show the relationship of this matrix problem to the two important scheduling problems: optimization of code for pipelined execution and microcode compaction for very long instruction computers.

This matrix problem is unclassified—it is unknown whether it is *NP*-Complete or whether it can be solved by a polynomial time algorithm. We find several minor extensions that would make the problem *NP*-Complete. Also, we show polynomial algorithms for a number of special cases of the problem, and develop a number of interesting techniques in the process. We also explore approximation algorithms and lower bounds.

Acknowledgements

Foremost, I would like to thank my advisor Donald Knuth. He started out by saving me from the sirens of Artificial Intelligence. Then he provided me with a fertile environment in which to explore many problems. Discussions with him proved stimulating as he indefatigably poked holes in my proofs, gave me new ideas and guidance, and recalled the just-right obscure reference in the literature. Also, while I will never acquire his ability for precision and clarity in thought and writing, I will be better off from having seen his example and from being pushed into trying.

Besides the constructive comments he made on this thesis, Ernst Mayr was the professor at Stanford whose numerous courses and seminars provided the greatest service and inspiration to young algorithms students of my class. At least we students at Stanford were very appreciative.

I also would like to thank my third reader Andrew Goldberg. I only wish that he had come to Stanford earlier in my career so that I would have had more opportunities to learn from him.

The papers that I co-authored while at Stanford provided another great source of learning and enjoyment at Stanford. I would like to thank all my co-authors for allowing me to enter their domains and for tolerating my leaden pace of writing: Evan Cohn, Anton Dahbura, Joan Feigenbaum, Jeff Naughton, Shaibal Roy, Alejandro Schäffer.

Numerous other friends also helped make the many years go by faster: Elizabeth Buss, Helen Davis, Marcia Derr, Scott Gimpel, Keith Hall, Peter Karp, Andrew Kosoresow, Tracy Larrabee, Sherry Listgarten, Oren Patashnik, Karen Peiper, Evan Reid, and others.

I am also indebted to my colleagues at the Western Research Lab of the Digital Equipment Corporation, who actively encouraged me during the final writing stretch. Equally important, they patiently tolerated bugs in my computer programs.

I received financial support from a variety of sources. My first year at Stanford was funded by a General Electric Foundation Fellowship. Subsequent years were funded by grants from DARPA, NSF and ONR. Much of the write-up of the thesis was supported by DEC Western Research Lab.

Next, I'd like to thank David Robbins. He is the high school math professor who first took me aside and suggested that there was more to computers than computer games. He directed me to a multi-volume set of books in the library. Local zoning limited buildings to four floors, so they numbered the floors of the library: ground floor, mezzanine, first floor, first mezzanine, second floor, second mezzanine, third floor, third mezzanine, fourth floor. I can remember it clearly now: there I was sitting in the mammoth library of one of the best prep schools in the country, wondering, "Why are they so cheap that they only bought the first three volumes?"

Lastly, my mother and father have always emphasized the importance of education and have made numerous sacrifices to ensure that I had the best. I will always be greatly indebted to them.

Contents

Chapter 1

Introduction

1.1 Probing the Innards

The triangularization problem that this thesis focuses on is deceptively simple to state:

> Given an $n \times n$ boolean matrix, is there a permutation of the rows and a permutation of the columns such that the resulting matrix is lower triangular?

This simplicity of definition is one of the things that makes this problem interesting. Another is that the problem is elusive: No one has been able to show either that the problem is in P or that the problem is NP-Complete. For the theoretically inclined, these two things are enough to warrant further investigation: simple, yet elusive. There must be something interesting going on here.

The practical minded may not be so easily intrigued. Yet, there is something here for them, too. For the problem can be viewed as part of a very fundamental and useful class: the class of scheduling problems. The tools developed as we dissect and probe the innards of the triangularization problem will surely be useful in other explorations.

Chapter 2

Basics and Warmups

2.1 The Basic Problem

We will focus first on a way to formulate the triangularization problem as a scheduling problem. A comprehensive recent overview of work on other scheduling problems can be found in [LLRS85].

We have two processors, p_X and p_Y, and two sets of jobs, $X = \{x_1, \ldots, x_m\}$ and $Y = \{y_1, \ldots, y_n\}$. We want to execute the jobs in the minimum amount of time. There are restrictions that we must obey:

- First, the X jobs must be executed on processor p_X and the Y jobs must be executed on processor p_Y. There are two ways to view this. We can view the two processors as being different; for example, one processor can perform only floating point operations and one can perform only integer operations. Alternately, we can consider the jobs as being pre-assigned to processors.

- Second, each job takes one unit of time to execute. Whether this unit is a minute or a micro-second is irrelevant—just so long as all the jobs take the same amount of time.

- Third, the jobs are non-pre-emptive. That is, once a job is started, it runs to completion; the job can not be halted and then restarted.

- Fourth, there exist $x_i \rightarrow y_j$ constraints between jobs. The constraint $x_1 \rightarrow y_3$ means that job x_1 must be completed before job y_3 can be started.

	$X = \{x_1, x_2, x_3, x_4\}$
	$Y = \{y_1, y_2, y_3, y_4\}$
Constraints:	$x_1 \rightarrow y_1,\ x_2 \rightarrow y_1,\ x_3 \rightarrow y_2,\ x_3 \rightarrow y_3,\ x_3 \rightarrow y_4,$ and $x_4 \rightarrow y_4$
Valid schedule:	$S_X = [x_3, x_4, x_1, x_2, \emptyset]$
	$S_Y = [\emptyset, y_2, y_3, y_4, y_1]$

Figure 2.1: A valid schedule

The specification of what job to execute at what time is referred to as a *schedule*. The schedule $S = [x_3, x_1, \emptyset, x_2]$ means that at time = 0 we start executing job x_3. When it finishes at time = 1, we start job x_1. Between time = 2 and time = 3 we execute no job; $\emptyset$ will sometimes be referred to as an *idle job*. At time = 3 we start executing job x_2. This job and the schedule terminate at time = 4.

For our problem, we will need two schedules, S_X and S_Y, to specify when the jobs are executed on processors p_X and p_Y, respectively.

A pair of schedules (S_X, S_Y) is *valid* if: (1) S_X contains each X job exactly once and contains no Y jobs; (2) S_Y contains each Y job exactly once and contains no X jobs; (3) all $x_i \rightarrow y_j$ constraints are satisfied. The schedule (S_X, S_Y) in Figure 2.1 is valid. Indeed as we will have little need to refer to invalid schedules, we will usually refer to "valid schedules" merely as "schedules".

The *length* of a schedule is the largest time when a non-$\emptyset$ job is executing. So the length of (S_X, S_Y) of Figure 2.1 is 5 time units, since p_Y's last non-$\emptyset$ job executes at time = 5. Since there is no valid (S_X, S_Y) schedule with length < 5, the schedule is a *minimum time* schedule or an *optimum* schedule.

And this is our task: to find an optimum schedule.

2.1.1 Viewed as a Graph Problem

Frequently, it will be convenient to think of the set of constraints as a graph, rather than merely as a set of $x_i \rightarrow y_j$ constraints. In this graph we have a node for each job and a directed $x_i \rightarrow y_j$ edge for each $x_i \rightarrow y_j$ constraint. Figure 2.2 shows the graph for Figure 2.1.

For this constraint graph we can build an $n \times m$ adjacency matrix. Each column will correspond to an x_i job and each row to a y_j job. There will be a 1 in x_i's column and y_j's

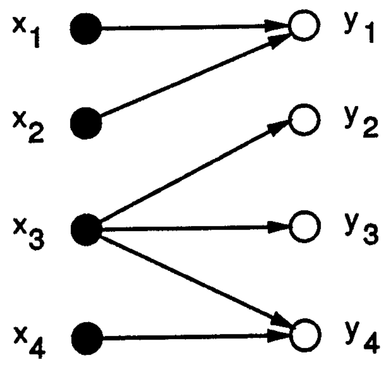


Figure 2.2: Graph representation

$$\begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ y_1 & \times & \times & \cdot & \cdot \\ y_2 & \cdot & \cdot & \times & \cdot \\ y_3 & \cdot & \cdot & \times & \cdot \\ y_4 & \cdot & \cdot & \times & \times \end{array}$$

Figure 2.3: Matrix representation

row iff there is an $x_i \rightarrow y_j$ edge—that is, an $x_i \rightarrow y_j$ constraint. Otherwise the matrix entry will be a 0. When we actually show these matrices, we will show the 1s as $\times$s and the 0s as $\cdot$s. Figures 2.3 and 2.4 show two possible adjacency matrices for the graph in Figure 2.2.

Note that the arrangement in Figure 2.4 is lower triangular—all the $\times$s are on or below the main diagonal. This indicates that the length 5 schedule $S_X = [x_3, x_4, x_2, x_1, \emptyset]$ and $S_Y = [\emptyset, y_2, y_3, y_4, y_1]$ is valid. The arrangement in Figure 2.3 protrudes one diagonal above the main diagonal. Thus, the following length 6 schedule is valid: $S_X = [x_1, x_2, x_3, x_4, \emptyset, \emptyset]$, $S_Y = [\emptyset, \emptyset, y_1, y_2, y_3, y_4]$.

$$\begin{array}{c|cccc} & x_3 & x_4 & x_2 & x_1 \\ y_2 & \times & \cdot & \cdot & \cdot \\ y_3 & \times & \cdot & \cdot & \cdot \\ y_4 & \times & \times & \cdot & \cdot \\ y_1 & \cdot & \cdot & \times & \times \end{array}$$

Figure 2.4: Alternate matrix representation

Definition 2.1 An *ordering*, like a schedule, specifies the order of execution of jobs; but unlike a schedule it does not indicate the exact time of execution. Thus, an ordering uses no $\emptyset$ jobs. For example, the ordering $O_Y = [y_1, y_2, y_4, y_3]$ indicates that we first execute y_1 then y_2 then y_4 then y_3, but it doesn't specify when we will execute them. Thus an ordering is simply a permutation of its jobs.

Definition 2.2 A *compact* schedule is one in which there are no idle times *between* jobs. There will only be idle times before and after execution. For example, the schedule $S = [\emptyset, \emptyset, x_1, x_2, \ldots, x_m, \emptyset, \emptyset]$ is compact.

Definition 2.3 The *delay* for a compact schedule (S_X, S_Y) is the difference between the number of $\emptyset$ jobs at the beginning of the S_Y schedule and the number of $\emptyset$ jobs at the beginning of the S_X schedule. For example, for the schedule $S = (S_X, S_Y)$ with $S_X = [x_1, x_2, x_3, x_4, \emptyset, \emptyset]$, and $S_Y = [\emptyset, \emptyset, y_1, y_2, y_3, y_4]$, then delay$(S) = 2 - 0 = 2$. This definition allows negative delays. For non-compact schedules, we will define the delay as the difference between the number of $\emptyset$ jobs before the last Y job in the S_Y schedule and the number of $\emptyset$ jobs before the first X job in the S_X schedule.

It turns out that:

Lemma 2.1 If $S = (S_X, S_Y)$ is valid and has delay$(S) = k$ then there is a *compact* schedule $S' = (S'_X, S'_Y)$ that has delay$(S') = k$.

Proof: Look at all idle Y jobs that execute before the last non-idle Y job. Move these jobs to the front of the Y schedule, yielding a compact schedule for Y.

Look at all idle X jobs that execute after the first non-idle X job. Move these jobs to the end of the X schedule, yielding a compact schedule for X.

This resulting schedule is compact and has the same delay as the original. ▮

So, given an ordering of the X jobs and an ordering of the Y jobs, we can easily determine the delay right away. Form the adjacency matrix with rows and columns arrayed as in the given orderings. If the $\times$s protrude above the main diagonal by k diagonals, then the delay will be $k + 1$ units of time.

There is another interesting aspect to note here.

Lemma 2.2 A square matrix M is triangularizable iff its transpose is triangularizable.

Proof: This is easy to see by example. Say that our original matrix is:

$$\begin{array}{ccccc} & x_1 & x_2 & x_3 & x_4 \\ y_1 & \times & \cdot & \cdot & \cdot \\ y_2 & \times & \times & \cdot & \cdot \\ y_3 & \times & \times & \times & \cdot \\ y_4 & \times & \times & \times & \times \end{array}$$

Hence, the transpose is:

$$\begin{array}{ccccc} & y_1 & y_2 & y_3 & y_4 \\ x_1 & \times & \times & \times & \times \\ x_2 & \cdot & \times & \times & \times \\ x_3 & \cdot & \cdot & \times & \times \\ x_4 & \cdot & \cdot & \cdot & \times \end{array}$$

If we merely reverse the order of the xs and the ys we get:

$$\begin{array}{ccccc} & y_4 & y_3 & y_2 & y_1 \\ x_4 & \times & \cdot & \cdot & \cdot \\ x_3 & \times & \times & \cdot & \cdot \\ x_2 & \times & \times & \times & \cdot \\ x_1 & \times & \times & \times & \times \end{array}$$

Thus, we can see that if a matrix is triangularizable, then so is its transpose. Since transposition is its own inverse, the converse is also true. ▮

While this is relatively intuitive when we look at the problem as a matrix problem, its equivalent statement as a scheduling problem is not as obvious.

Definition 2.4 The reverse of a schedule S, indicated by $\text{rev}(S)$, is obtained by listing the elements of the schedule in reverse order. That is, $\text{rev}([x_1, x_2, x_3, \emptyset]) = [\emptyset, x_3, x_2, x_1]$.

With this notation and a slight extension of the proof, we can state a stronger scheduling problem variation of Lemma 2.2.

Corollary 2.3 For a constraint graph G with $|X| = |Y|$, there is a schedule $S = (S_X, S_Y)$ with $\text{delay}(S) = k$ iff there is a schedule $S' = (\text{rev}(S_Y), \text{rev}(S_X))$ with $\text{delay}(S') = k$ for the constraint graph G' obtained from G by reversing all the arcs and interchanging the roles of the X nodes and Y nodes. ▮

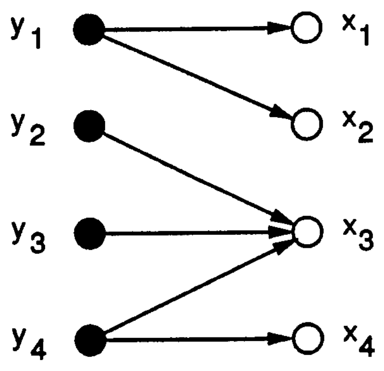


Figure 2.5: Transposed version of Figure 2.2

That is, instead of the graph in Figure 2.2 we would have the graph in Figure 2.5. Visually comparing these two figures, it is not obvious that they have the same delay; but we've seen that they do. And this delay is achieved in the reversed arc case by reversing the schedules from the non-reversed case.

Lastly, there is an even stronger corollary.

Corollary 2.4 If (S_X, S_Y) is optimal for the square matrix M, then $(\text{rev}(S_Y), \text{rev}(S_X))$ is optimal for M^T. ▮

2.1.2 Implied Orderings

We can make a further observation. If we are given an ordering of the X jobs we can easily find an ordering of the Y jobs that minimizes the delay for the given ordering of the X jobs; that is, the ordering for the Y jobs is implied by the ordering for the X jobs. There are a number of equivalent ways to look at the method for ordering Y.

VIEW 1: Start with the constraint graph, an ordering of the X nodes and with a null ordering of the Y nodes.

1. If any Y node is isolated, remove it and append its job to the end of the Y ordering.

2. Repeat until the graph is empty: Remove the next X node according to the ordering; if any Y node is isolated, remove it and append its node to the end of the Y ordering.

So, for Figure 2.2 with $O_X = [x_2, x_3, x_1, x_4]$, the iterations through the loop modify the graph and schedule as shown in Figure 2.6.

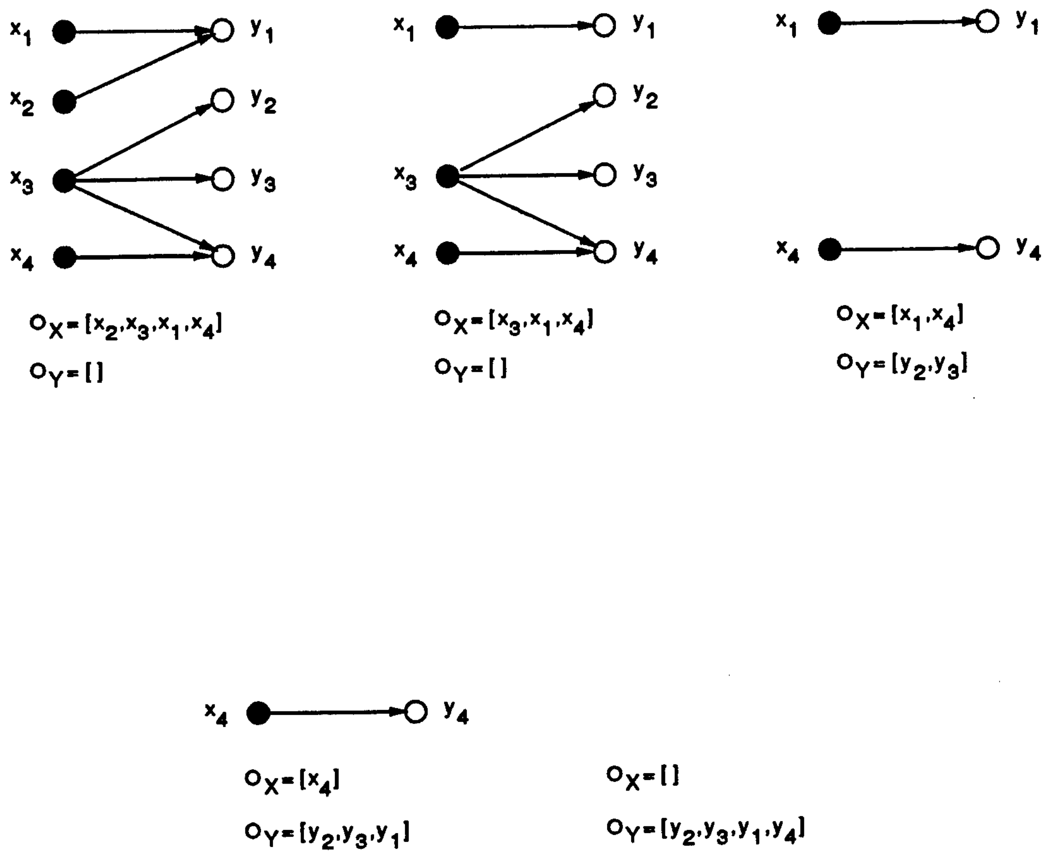


Figure 2.6: Snapshots of VIEW 1

$$\begin{array}{ccccc} & x_2 & x_3 & x_1 & x_4 \\ y_1 & \times & \cdot & \times & \cdot \\ y_2 & \cdot & \times & \cdot & \cdot \\ y_3 & \cdot & \times & \cdot & \cdot \\ y_4 & \cdot & \times & \cdot & \times \end{array} \qquad \begin{array}{ccccc} & x_2 & x_3 & x_1 & x_4 \\ y_1 & \times & \cdot & \times & \cdot \\ y_2 & \cdot & \times & \cdot & \cdot \\ y_3 & \cdot & \times & \cdot & \cdot \\ y_4^* & \cdot & \times & \cdot & \times \end{array}$$

$$\begin{array}{ccccc} & x_2 & x_3 & x_1 & x_4 \\ y_2 & \cdot & \times & \cdot & \cdot \\ y_3 & \cdot & \times & \cdot & \cdot \\ y_1^* & \times & \cdot & \times & \cdot \\ y_4^* & \cdot & \times & \cdot & \times \end{array} \qquad \begin{array}{ccccc} & x_2 & x_3 & x_1 & x_4 \\ y_2^* & \cdot & \times & \cdot & \cdot \\ y_3^* & \cdot & \times & \cdot & \cdot \\ y_1^* & \times & \cdot & \times & \cdot \\ y_4^* & \cdot & \times & \cdot & \times \end{array}$$

Figure 2.7: Snapshots with fixed O_X.

VIEW 2: Consider the adjacency matrix with the columns in O_X order. At the beginning all rows are "unfrozen". When we freeze a row it can't be moved anymore. A frozen row will appear as y_i^*. Iterating through the columns in reverse order, for each column: compact downward all unfrozen rows with an $\times$ in this column and then freeze those rows. For example, VIEW 2 gives the snapshots shown in Figure 2.7. At the end, we can read off the rows from top to bottom, yielding $O_Y = [y_2, y_3, y_1, y_4]$.

We can also perform an analogous algorithm for a fixed O_Y and thus find the optimum O_X for that O_Y. Rather than specifying the analogous algorithm for a fixed O_Y, we will devise an equivalent algorithm by using the algorithm for a fixed O_X as a sub-routine. This equivalent algorithm merely consists of transposing the matrix, reversing the order of the X nodes and Y nodes and then calling the algorithm for a fixed O_X. The resulting snapshots are shown in Figure 2.8.

So iterating this procedure has given us a delay $= 1$ optimum schedule. In general, iterating this procedure will not always produce globally optimal schedules. For instance, the fixed point shown in Figure 2.9 is non-optimum. The optimum Y order for $[x_1, x_2, x_3, x_4]$ is $[y_1, y_2, y_3, y_4]$ and the optimum X order for $[y_1, y_2, y_3, y_4]$ is $[x_1, x_2, x_3, x_4]$. These have a delay of 2; but the schedule with $S_X = [x_4, x_3, x_2, x_1]$ and $S_Y = [y_4, y_3, y_2, y_1]$ gives us a lower triangular matrix—that is, a schedule with a delay of only 1.

$$\begin{array}{c|cccc} & y_4 & y_1 & y_3 & y_2 \\ \hline x_4 & \times & \cdot & \cdot & \cdot \\ x_1 & \cdot & \times & \cdot & \cdot \\ x_3 & \times & \cdot & \times & \times \\ x_2 & \cdot & \times & \cdot & \cdot \end{array} \qquad \begin{array}{c|cccc} & y_4 & y_1 & y_3 & y_2 \\ \hline x_4 & \times & \cdot & \cdot & \cdot \\ x_1 & \cdot & \times & \cdot & \cdot \\ x_2 & \cdot & \times & \cdot & \cdot \\ x_3^* & \times & \cdot & \times & \times \end{array}$$

$$\begin{array}{c|cccc} & y_4 & y_1 & y_3 & y_2 \\ \hline x_4 & \times & \cdot & \cdot & \cdot \\ x_1^* & \cdot & \times & \cdot & \cdot \\ x_2^* & \cdot & \times & \cdot & \cdot \\ x_3^* & \times & \cdot & \times & \times \end{array} \qquad \begin{array}{c|cccc} & y_4 & y_1 & y_3 & y_2 \\ \hline x_4^* & \times & \cdot & \cdot & \cdot \\ x_1^* & \cdot & \times & \cdot & \cdot \\ x_2^* & \cdot & \times & \cdot & \cdot \\ x_3^* & \times & \cdot & \times & \times \end{array}$$

Figure 2.8: Snapshots with fixed O_Y.

$$\begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline y_1 & \times & \times & \cdot & \cdot \\ y_2 & \cdot & \times & \times & \cdot \\ y_3 & \cdot & \cdot & \times & \times \\ y_4 & \cdot & \cdot & \cdot & \times \end{array} \qquad \begin{array}{c|cccc} & y_4 & y_3 & y_2 & y_1 \\ \hline x_4 & \times & \times & \cdot & \cdot \\ x_3 & \cdot & \times & \times & \cdot \\ x_2 & \cdot & \cdot & \times & \times \\ x_1 & \cdot & \cdot & \cdot & \times \end{array}$$

Figure 2.9: Non-optimum fixed point

$$\begin{array}{ccccc} & x_1 & x_2 & x_3 & x_4 \\ y_1 & \times & \cdot & \cdot & \cdot \\ y_2 & \cdot & \times & \cdot & \cdot \\ y_3 & \times & \times & \cdot & \times \\ y_4 & \cdot & \cdot & \times & \cdot \\ y_5 & \times & \cdot & \times & \times \end{array}$$

Figure 2.10: M_1 satisfies $\mathrm{lt}(M_1, 2)$ but not $\mathrm{lt}(M_1, 1)$

$$\begin{array}{ccccc} & x_1 & x_2 & x_3 & x_4 \\ y_1 & \times & \cdot & \cdot & \cdot \\ y_2 & \cdot & \times & \cdot & \cdot \\ y_4 & \cdot & \cdot & \times & \cdot \\ y_3 & \times & \times & \cdot & \times \\ y_5 & \times & \cdot & \times & \times \end{array}$$

Figure 2.11: M_2 satisfies $\mathrm{lt}(M_2, 1)$

2.2 More Basics

We've been relatively informal so far. But now we will need to get more mathematical and precise. For this let us concentrate on the matrix formulation.

An $n \times m$ matrix M is lower triangular below the k-diagonal if all the 1s are strictly below the k-th diagonal, where the main diagonal is labelled the 0-th. Let's define a function which tells us if a matrix is lower triangular below the k-th diagonal.

Definition 2.5 For an $n \times m$ matrix M, let the boolean function $\mathrm{lt}(M, i)$ be true iff each element of $M[r][c]$ satisfies $c - r \geq i \Rightarrow M[r][c] = 0$.

The matrix M_1 in Figure 2.10 satisfies $\mathrm{lt}(M_1, 2)$ but not $\mathrm{lt}(M_1, 1)$. When we interchange rows y_3 and y_4 yielding the matrix M_2 in Figure 2.11, then M_2 satisfies $\mathrm{lt}(M_2, 1)$.

Hence, if the order of the rows indicates the ordering of the Y jobs in a particular schedule and the order of the columns indicates the ordering of the X jobs in the same schedule, then the delay of the schedule is equal to the smallest i such that $\mathrm{lt}(M, i)$ is true.

What we are ultimately after, though, is the minimum delay over all permutations of the rows and columns.

Definition 2.6 Let the boolean function $\mathrm{plt}(M, i)$ be true iff there is a matrix M', derived from M by permutations of rows and columns, such that $\mathrm{lt}(M', i)$ is true.

So for the matrix M_1 in Figure 2.10 and the matrix M_2 in Figure 2.11, $\mathrm{plt}(M_1, 1)$ and $\mathrm{plt}(M_2, 1)$. In general, if M_1 is a permutation of M_2, then $\mathrm{plt}(M_1, i) \equiv \mathrm{plt}(M_2, i)$.

With these new definitions, our transposition result in Corollary 2.3 can be stated as:

Corollary 2.5 For a square matrix M, $\mathrm{plt}(M, i) \iff \mathrm{plt}(M^{\mathrm{T}}, i)$. ▮

This can be generalized for rectangular matrices.

Corollary 2.6 For an $n \times m$ matrix M, $\mathrm{plt}(M, i) \iff \mathrm{plt}(M^{\mathrm{T}}, i + n - m)$. ▮

Corollary 2.4 can also be generalized for rectangular matrices.

Corollary 2.7 If (S_X, S_Y) is optimal for the $n \times m$ matrix M, then $(\mathrm{rev}(S_Y), \mathrm{rev}(S_X))$ is optimal for M^{T}. ▮

For any given matrix M, we will refer to the smallest i such that $\mathrm{plt}(M, i)$ holds as the *overhang* of the matrix. This will occasionally be written as $\mathrm{overhang}(M)$.

Alternately, we could define

$$\mathrm{overhang}(M) = \min_{A,B} \mathrm{delay}(A\,M\,B)$$

where A and B are row and column permutation matrices, respectively.

Let's explore the following problem.

Permutable into Lower Triangular (PLT_i)

INSTANCE: An $n \times m$ matrix M and a number i.

QUESTION: Is $\mathrm{plt}(M, i)$ true?

Thus, PLT_1 is simply the triangularization problem. PLT_0 is similar, except that 1s aren't allowed on the diagonal.

The first obvious question is, how are these problems related to each other for various values of i?

Let's start by taking an arbitrary $n \times m$ matrix M and adding to it a new column x_{m+1} with all 1s. Call the new matrix M'. If we do this to M_2, we get the matrix M_2'

$$\begin{array}{cccccc}
 & x_1 & x_2 & x_3 & x_4 & x_5 \\
y_1 & \times & \cdot & \cdot & \cdot & \times \\
y_2 & \cdot & \times & \cdot & \cdot & \times \\
y_3 & \cdot & \cdot & \times & \cdot & \times \\
y_4 & \times & \times & \cdot & \times & \times \\
y_5 & \times & \cdot & \times & \times & \times
\end{array}$$

Figure 2.12: M'_2

$$\begin{array}{cccccc}
 & x_5 & x_1 & x_2 & x_3 & x_4 \\
y_1 & \times & \times & \cdot & \cdot & \cdot \\
y_2 & \times & \cdot & \times & \cdot & \cdot \\
y_3 & \times & \cdot & \cdot & \times & \cdot \\
y_4 & \times & \times & \times & \cdot & \times \\
y_5 & \times & \times & \cdot & \times & \times
\end{array}$$

Figure 2.13: plt$(M'_2, 2)$

shown in Figure 2.12. By changing the column order as shown in Figure 2.13 we see that plt$(M_2, 1) \iff$ plt$(M'_2, 2)$. Our claim is that for any choice of M, plt$(M, 1) \iff$ plt$(M', 2)$.

This is fairly straightforward to see. If we are given some delay $= 1$ schedule (S_X, S_Y) for M, then it is clear that adding the new column to the beginning of S_X yields a delay $= 2$ schedule for M'.

Similarly, if we are given some schedule for M', (S'_X, S'_Y), with delay $= 2$, then it is clear that by merely dropping out the new column, we get a schedule for M with delay $= 1$, because x_{m+1} must be executed before any Y job and now the Y jobs can all be executed one time unit earlier.

So for any M, we can find a matrix M' such that plt$(M, 1) \iff$ plt$(M', 2)$. This tells us that $\text{PLT}_1 \preceq_{\mathcal{P}} \text{PLT}_2$. More generally, this same technique tells us that:

Lemma 2.8 $\text{PLT}_i \preceq_{\mathcal{P}} \text{PLT}_{i+1}$, for $i \geq 0$. ▮

Let's formalize some of the facts that we implicitly used.

Lemma 2.9 If there is some job x_k with all 1s in its column, then there is an optimal schedule that has x_k first in S_X.

$$
\begin{array}{cccccc}
 & x_1 & x_2 & x_3 & x_4 & x_5 \\
y_1 & \times & \times & \cdot & \cdot & \times \\
y_2 & \cdot & \times & \times & \times & \cdot \\
y_3 & \times & \times & \cdot & \cdot & \times \\
y_4 & \cdot & \times & \times & \times & \cdot \\
y_5 & \cdot & \times & \times & \cdot & \cdot
\end{array}
$$

Figure 2.14: Example of domination

Proof: Assume without loss of generality that there is an optimal schedule with $S_X = [x_1, x_2, \ldots, x_m]$. Consider changing the schedule to $S'_X = [x_k, x_1, x_2, \ldots, x_{k-1}, x_{k+1}, \ldots, x_m]$ and leaving $S'_Y = S_Y$. Since no Y job could have started before time $k+1$, this new schedule is still valid. ▮

Similarly, we can establish a dual result,

Lemma 2.10 If there is some job y_k with all 1s in its row, then there is an optimal schedule which has y_k last in S_Y.

Proof: Instead of using a proof analogous to that of Lemma 2.9, let's use what we know about transposition. Look at M^{T}. Now, by Lemma 2.9, we know that there is an optimum schedule, (S_Y, S_X), for M^{T} with y_k first in S_Y. By our transposition result of Corollary 2.7, $(\mathrm{rev}(S_X), \mathrm{rev}(S_Y))$ is optimum for $M^{\mathrm{TT}} = M$. Clearly this schedule uses y_k last. ▮

This idea can be extended. We will say that a column of a matrix *dominates* another if the first one has a 1 in every row that the second one does.

For example, in Figure 2.14 column x_2 dominates columns x_1, x_3, x_4, and x_5. Column x_3 dominates column x_4. Columns x_1 and x_5 dominate each other. Clearly two columns can dominate each other only if they are equal.

This allows us to generalize Lemma 2.9.

Lemma 2.11 If column x_j dominates column x_k, then there is an optimum schedule with job x_j preceding job x_k.

Proof: Similar idea to that in Lemma 2.9. Assume we have an optimum schedule (S_X, S_Y) with x_k before x_j. Without loss of generality, we can assume that $k < j$ and that

$S_X = [x_1, x_2, \ldots, x_m]$. Let $S'_X = [x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_j, x_k, x_{j+1}, \ldots, x_m]$. Once again it is clear that (S'_X, S_Y) is a valid schedule. ▮

We can define the same idea for rows. In Figure 2.14 rows y_2 and y_4 dominate each other and row y_5. Transposition clearly gives us:

Corollary 2.12 If the row for job y_j dominates the row for job y_k, then there is an optimum schedule with job y_j following job y_k. ▮

We say that a column *strictly dominates* another if it dominates the other and they are unequal. Thus, in Figure 2.14, column x_1 does not strictly dominate x_5 nor vice versa. But column x_2 strictly dominates columns x_1, x_3, x_4, and x_5 and column x_3 strictly dominates columns x_4.

By applying the same technique from Lemma 2.11 iteratively, we get an even stronger result:

Lemma 2.13 There is an optimum schedule such that

1. For every pair of jobs such that x_j strictly dominates x_k, job x_j precedes x_k.
2. For every pair of jobs such that y_j strictly dominates y_k, job y_j follows y_k.
3. For every pair of X jobs that have equal columns, every intervening X job has an equal column.
4. For every pair of Y jobs that have equal rows, every intervening Y job has an equal row.

▮

Also note that given any optimal schedule that doesn't satisfy these conditions, we can modify it to satisfy them in polynomial time.

While all of this is interesting and will be useful later, let's return to our immediate task of exploring the relationship between the PLT$_i$ problems. We know that PLT$_i \preceq_{\mathcal{P}}$ PLT$_{i+1}$ if $i \geq 0$. What if $i < 0$?

Lemma 2.14 If plt(M, i) for $i \leq 0$, then M must have at least $|i| + 1$ rows of all 0s. ▮

So assume we are trying to solve $\text{plt}(M, i)$ with $i \leq 0$. Since there is at least one row of all 0s, we create matrix M' by deleting one such row.

Clearly if we have a delay $= i$ schedule for M, then we can obtain a delay $= i + 1$ schedule for M'. The reverse is just as clear: If we have a delay $= i + 1$ schedule (S'_X, S'_Y) for M' then we can obtain a delay $= i$ schedule for M by re-adding the deleted row at the beginning of S'_Y. Thus $\text{plt}(M', i + 1) \equiv \text{plt}(M, i)$.

So, now we've shown $\text{PLT}_i \preceq_{\mathcal{P}} \text{PLT}_{i+1}$ for all $i \leq 0$ and we already showed in Lemma 2.8 that it was true for $i \geq 0$. Hence:

Lemma 2.15 $\text{PLT}_i \preceq_{\mathcal{P}} \text{PLT}_{i+1}$ for all i. ▌

Similarly, we can show that

Lemma 2.16 $\text{PLT}_{i+1} \preceq_{\mathcal{P}} \text{PLT}_i$ for all i.

Proof: Start with a matrix M' and the question $\text{plt}(M', i + 1)$. Create M by adding a row of 0s. Our argument from the earlier paragraph still holds, telling us that $\text{plt}(M', i+1) \equiv \text{plt}(M, i)$ and hence $\text{PLT}_{i+1} \preceq_{\mathcal{P}} \text{PLT}_i$. ▌

Consequently all the problems PLT_i have equivalent complexity up to polynomial factors, for fixed i .

2.3 Variants

There is still much more use we can make of our dominance lemma, Lemma 2.13. We can use it to show that our original problem is equivalent to a more general variant. This general version allows $x \rightarrow x$ and $y \rightarrow y$ precedence constraints and non-unit execution times.

2.3.1 Non-unit execution times

Let's start with non-unit execution times. In this case we associate a positive integer weight or execution length with each node/job as in Figure 2.15.

If we iteratively apply the following modification to each X node and Y node then we will have an instance of our original problem: If a node is of weight k, then replace it with k 1-unit nodes, each of which has arcs to/from the same nodes as the original weighted node did. See Figure 2.16 for an example of the transformation.

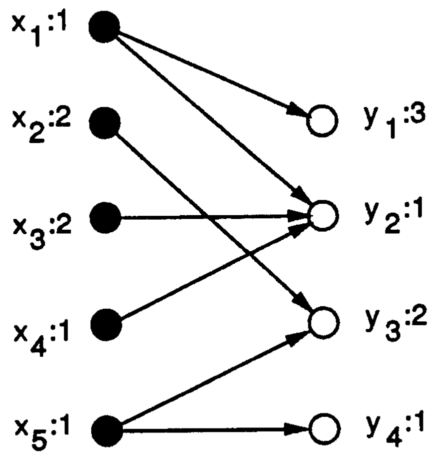


Figure 2.15: Non-unit execution times

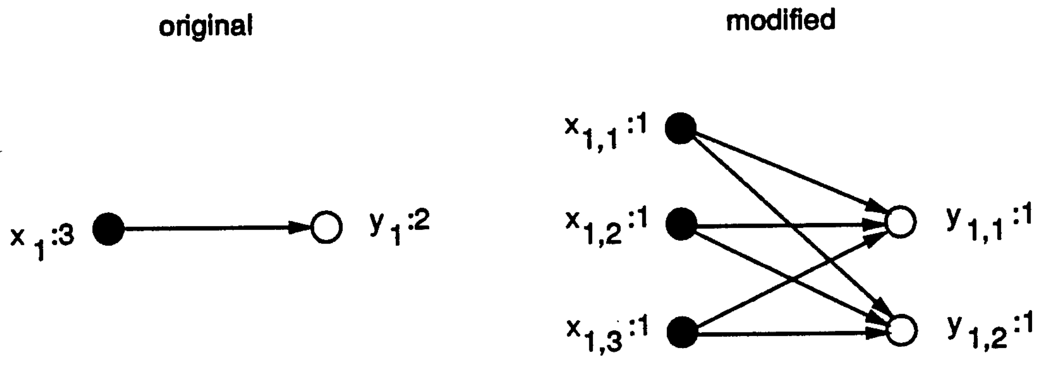


Figure 2.16: Mapping weighted times to unit execution times

We will refer to $x_{1,1}$, $x_{1,2}$ and $x_{1,3}$ as the *sub-nodes* of x_1. Obviously a delay $= i$ schedule for the weighted version yields a delay $= i$ schedule for the unit weight version. What we must show is that any delay $= i$ schedule for the unit weight graph can be modified to yield a delay $= i$ schedule for the original graph. The dominance lemma does all the work for us.

Assume we have a delay $= i$ schedule of G'. By Lemma 2.13 we can easily construct a schedule where all nodes of equal dominance are adjacent. For example, given the following schedule for the modified version of the graph in Figure 2.15

$$([x_{1,1}, x_{2,1}, x_{5,1}, x_{2,2}, x_{3,2}, x_{4,1}, x_{3,1}, \emptyset], [\emptyset, y_{1,2}, y_{1,3}, y_{4,1}, y_{1,1}, y_{2,1}, y_{3,1}, y_{3,2}])$$

dominance tells us that we can modify it to

$$([x_{1,1}, x_{5,1}, x_{2,1}, x_{2,2}, x_{3,2}, x_{4,1}, x_{3,1}, \emptyset], [\emptyset, y_{1,2}, y_{1,3}, y_{1,1}, y_{4,1}, y_{2,1}, y_{3,1}, y_{3,2}])$$

After this, the only case when a set of sub-nodes aren't adjacent is if two of the weighted nodes had exactly the same neighbors—as happened with x_3 and x_4 in the example.

But, clearly in such a case, we can re-order the nodes yielding:

$$([x_{1,1}, x_{5,1}, x_{2,1}, x_{2,2}, x_{4,1}, x_{3,2}, x_{3,1}, \emptyset], [\emptyset, y_{1,2}, y_{1,3}, y_{1,1}, y_{4,1}, y_{2,1}, y_{3,1}, y_{3,2}])$$

Since all sub-jobs are adjacent, the weighted node schedule

$$([x_1, x_5, x_2, x_2, x_4, x_3, x_3, \emptyset], [\emptyset, y_1, y_1, y_1, y_4, y_2, y_3, y_3])$$

is clearly valid. So, we can transform any delay $= i$ schedule of the unit-weight version into a delay $= i$ schedule of the weighted version.

2.3.2 Allowing $x \rightarrow x$ Constraints

Let's look at what happens to the problem when we allow $x \rightarrow x$ constraints. Our basic claim is that we can replace all the $x \rightarrow x$ links by a set of $x \rightarrow y$ links such that any solution for the modified problem can be converted into a solution for the original problem and vice-versa. Figure 2.17 has an example.

The way we perform this transformation is as follows:

1. Find the transitive closure of all the constraints and add all those edges to the graph.

2. Throw away all the $x \rightarrow x$ edges.

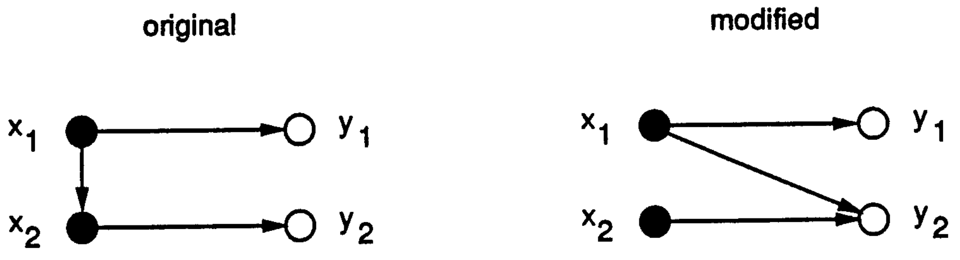


Figure 2.17: Removing $x \rightarrow x$ edges: valid

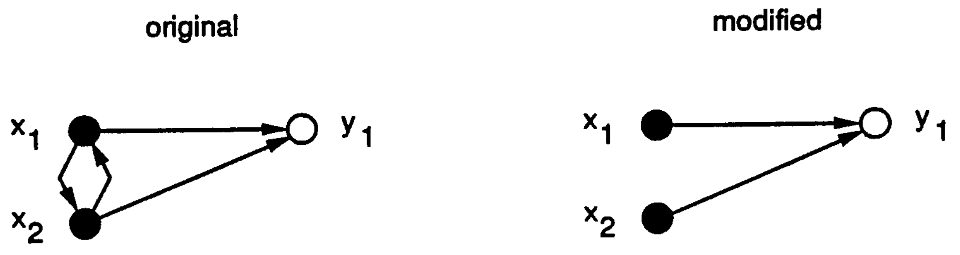


Figure 2.18: Removing $x \rightarrow x$ edges: invalid

It is clear that taking the transitive closure has no effect on the solutions: any valid schedule before is a valid schedule afterward and vice-versa.

What we must show is that it is permissible to throw away the $x \rightarrow x$ edges at this point. Our difficulty is shown in Figure 2.18. It has no valid schedule before discarding the edges, but it does have one afterwards.

Hence, we will show that *if* the original has a *valid* schedule, then any delay $= i$ schedule of the original graph can be converted into a delay $= i$ schedule of the modified graph.

Again, it is obvious that any delay $= i$ schedule of the original graph is a delay $= i$ schedule of the modified graph.

We need to show the converse. Say we are given a schedule of the modified graph. To get a valid delay $= i$ schedule of the original graph simply perform the following routine:

1. Apply the dominator method to it, yielding a schedule (S'_X, S'_Y) satisfying Lemma 2.13.
2. If any columns of the modified graph are equal, their x nodes will all be adjacent to each other. Re-order these columns in topological order according to the $x \rightarrow x$ ordering of these nodes in the original graph.

To see correctness: If x_i precedes x_j in the original graph then column x_i dominates column x_j in the modified graph. There are two cases. In one case, column x_i strictly dominates column x_j, and hence x_i will already be before x_j in S'_X. In the other case, x_i and x_j have equal columns and there again, x_i will be before x_j because of the topological ordering in step (2) of our routine.

2.3.3 All Together

What about allowing $y \rightarrow y$ arcs? Although transposition as we've proven it so far doesn't automatically imply this from the results about the $x \rightarrow x$ arcs, it can be easily extended to do so. That is

Lemma 2.17 Given an optimum schedule (S_X, S_Y) for a graph G with $x \rightarrow y$ and $x \rightarrow x$ arcs, $(\text{rev}(S_Y), \text{rev}(S_X))$ will be an optimum schedule for the graph with the directions of the edges reversed. ▮

But ultimately, we claim not merely the three individual results but rather the composition of them. We need to transform a graph with $x \rightarrow x$ edges, $y \rightarrow y$ edges and non-unit weights into an equivalent one without $x \rightarrow x$ edges, $y \rightarrow y$ edges and non-unit weights.

Our combined transformation is

Given a graph G with $x \rightarrow x$, $x \rightarrow y$ and $y \rightarrow y$ arcs and non-unit weights.

1. Find the transitive closure of G.
2. Discard all $x \rightarrow x$ and $y \rightarrow y$ edges.
3. Split up each node with weight k into k unit-weight nodes.

Assuming we have a valid schedule for the modified problem, we can convert it into a valid schedule for the original, if one exists, by:

1. Apply the dominance lemma to both rows and columns.
2. For each set of equal columns (rows):
   (a) Re-order them so that all sub-nodes of an original node are adjacent.
   (b) Combine those nodes into a single node of weight k.
   (c) Re-order the weighted nodes to obey the original topological ordering.

In going from the schedule for the modified graph to the schedule for the original graph, all the modifications are delay-non-increasing.

This, along with the fact that any schedule for the original graph can be transformed into a schedule for the modified graph with the same delay tells us that the two problems are equivalent.

2.3.4 Contractor Cash-Flow

A non-computer scenario where this problem is useful involves a contractor who is working on a large construction project. He is paid portions of his earnings for reaching certain milestones in the construction: \$$A$ when the foundation is laid, \$$B$ when the frame is up, \$$C$ when the plumbing and electrical wiring are installed. Naturally there are restrictions on the order he completes various tasks — you can't install wiring in thin air. Lastly, the contractor has to make payments for materials and labor associated with various tasks.

The contractor doesn't have a large financial reserve, and he needs to complete the project without running out of cash. How should he order the tasks so as to minimize the amount of his own money that he needs to use to pay for the intermediate expenses? He doesn't want his cash balance to go below 0.

This problem corresponds to allowing $x \rightarrow x$ constraints and weighted nodes. Every x node represents a task to be completed and its weight represents the cost in material and labor that the contractor must put up to complete that task. The $x \rightarrow x$ arcs encode the precedence relations among the tasks. There is one isolated y node with a weight equal to his initial cash balance. The other y nodes represent pay-off conditions, which may be any conjunction of tasks — if completing tasks x_1 and x_3 means that the contractor is paid \$$w$, then we create a y node with weight w and precedence arcs to it from x_1 and x_3.

The contractor runs out of cash iff overhang ≥ 0.

At this point we should make the technical note that the sizes of the payoffs must be limited by some constant or limited by some multiple of $\log n$, for the problem to be equivalent under polynomial reductions to PLT_0.

2.4 Allowing $y \rightarrow x$ Constraints

The next natural question is what happens when we allow all four types of arcs: $x \rightarrow y$, $x \rightarrow x$, $y \rightarrow y$, and $y \rightarrow x$. This turns out to be a well-studied problem.

2.4.1 Microcode Compaction for Very Long Instruction Word Computers

One approach to increasing parallelism within a processor is the use of very-long instruction words, or VLIWs [Fis79] [Veg82] [Lam87] [FERN84] [Tou84] [LDSM80]. A conventional processor will read in a machine-code instruction such as "`ADD R1, R2`" which it then decodes and maps to a series of microcode instructions. These microcode instructions are extremely low-level. Each instruction is made up of micro-operations: put the contents of R1 out onto internal bus B; read the contents of internal bus A into the ALU; and so on.

By their nature, the micro-operations for different components of the processor can be executed in parallel. Due to the coarse nature of the instructions like "`ADD R1, R2`" however, large portions of the processor may be idle at any time. The VLIW approach says: instead of the program being written as machine code, let's have it written as micro-code. Let's have the compiler translate our high-level language into a series of micro-code instructions. Since micro-code specifies so much detail, it requires much longer instruction words—and hence the name VLIW.

Obviously to make use of this new ability we don't just translate the source code into the same microcode instructions that the machine code gets translated to and then concatenate them all. This is a first step, but it doesn't yield any improvements. Instead, we want to compact these micro-ops together so that some of them will be executed in parallel. Naturally, the micro-ops will have precedence constraints between them. The micro-instructions typically have several fields and each micro-op must go into a specific field. For example, if there are exactly two fields, and we represent instructions that must go in the first field with circles and instructions that must go in the second field with squares, then we might have a precedence graph like the one in Figure 2.19.

So we can now define the micro-code compaction problem for VLIWs.

Micro-Code Compaction for VLIWs (MC-COMPACTION)

INSTANCE: Set O of unit-time micro-operations, number k of classes of operations, class number $1 \le c(o) \le k$ for each $o \in O$, partial order on O and deadline t.

QUESTION: Is there a schedule of the micro-operations that obeys the partial order, never schedules two micro-operations with the same value of $c(o)$ for the same time slot and is of length $\le t$?

This problem with $k = 2$ is equivalent to our scheduling problem allowing $x \to x$, $x \to y$,

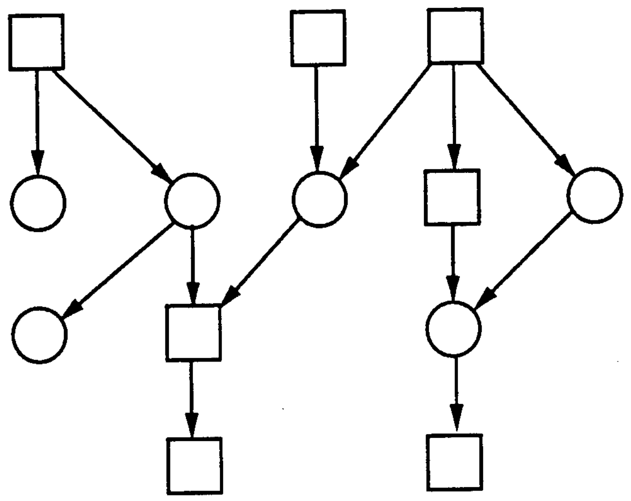


Figure 2.19: Micro-operations with dependencies

$y \rightarrow y$ and $y \rightarrow x$ arcs.

This problem is NP-complete. This was shown by Goyal [Goy76, Goy77]. Since we will want to modify his proof, let's look at it in detail.

He reduces the m-PROCESSOR SCHEDULING problem to it. The m-PROCESSOR SCHEDULING problem is known to be NP-complete [Ull76].

m-PROCESSOR SCHEDULING

INSTANCE: Set X of unit-time tasks, number m of processors, partial order on X, and deadline t.

QUESTION: Is there a schedule of the $n = |X|$ tasks that obeys the partial order, never schedules more than m tasks for the same time slot and is of length $\leq t$?

This problem can be reduced to MC-COMPACTION with $k = 2$.

We start with the precedence graph for an instance of m-PROCESSOR SCHEDULING such as that in Figure 2.20.

Split each node into two nodes—an in-node and an out-node. Replace any arc $x_i \rightarrow x_j$ with an $x_i^{\text{out}} \rightarrow x_j^{\text{in}}$ arc. Add the arcs $x_i^{\text{in}} \rightarrow x_i^{\text{out}}$. Figure 2.21 shows the result.

The out-node jobs will be required to execute on processor p_X and the in-node jobs will be required to execute on processor p_Y. Lastly, we will add an enforcing graph consisting of the filled nodes in Figure 2.22. The dashed nodes represent the slots available for execution

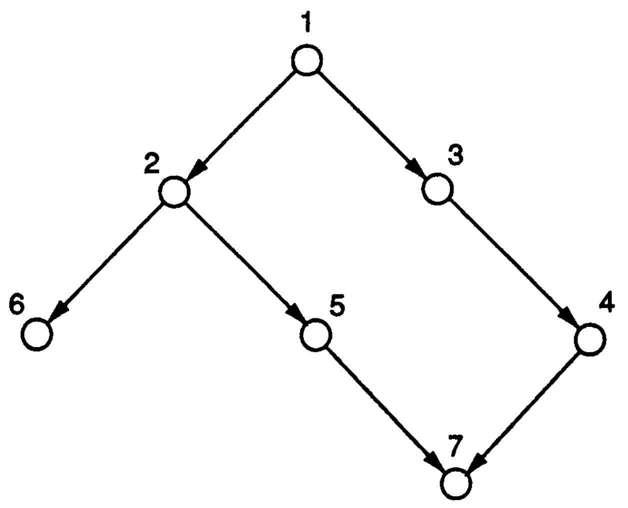


Figure 2.20: m-PROCESSOR SCHEDULING instance

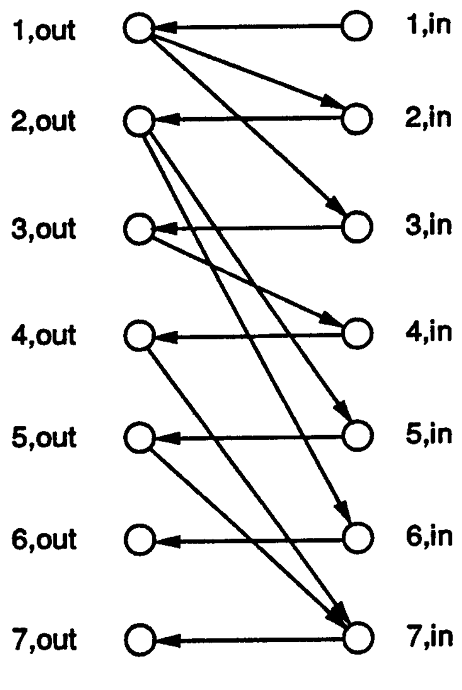


Figure 2.21: Transformed version of Figure 2.20

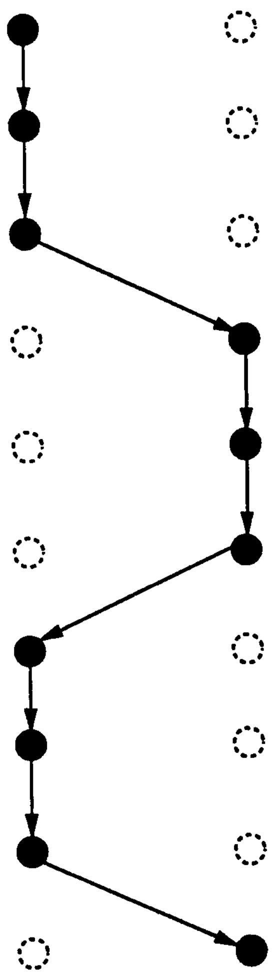


Figure 2.22: Enforcing graph for $m = 3$

of the other jobs.

We ask the question, can the transformed jobs be scheduled to complete in $2n$ time steps? The answer is yes iff the m-PROCESSOR PROBLEM could have been scheduled in n/m time steps.

This is straightforward to see. The only out-nodes that can execute at times $m + 1$ to $2m$ are those corresponding to the in-nodes that were executed from time 1 to m. This will then make available for execution at time $2m + 1$ all those jobs that are released by running the in-jobs we ran at time 1 to m. Thus, any schedule for the m-PROCESSOR PROBLEM can be transformed into a schedule for MC-COMPACTION.

To see the other direction, we merely have to note that there is no way to cheat: (1)

No out-node can be executed before its corresponding in-node; (2) if an in-node is executed during some time range $2am + 1$ to $(2a + 1)m$, then all out-nodes that (edgewise) precede it must have been executed before $2am + 1$ and hence all in-nodes of jobs that (edgewise) precede it must have been executed before $2(a - 1)m + 1$.

There is an even tighter result that shows that the MC-COMPACTION remains *NP*-complete even when the precedence graphs are limited to being chains [LLMS87]. That reduction is from 3-partition.

But we want to limit the graphs in a different direction. We want to know the minimum degree of $y \rightarrow x$ arcs we can allow while still having the problem be *NP*-complete.

Our construction shows that allowing one $y \rightarrow x$ and one $y \rightarrow y$ out of each y node and one $x \rightarrow x$ out of each X node and an unlimited number of $x \rightarrow y$ arcs makes the problem *NP*-complete. What if we didn't want to allow $y \rightarrow y$ and $x \rightarrow x$ arcs? We could try the dominance methods we tried in Section 2.3. But they don't carry over.

Alternately we could replace the enforcing graph of Figure 2.22 with that of Figure 2.23. That is, for the enforcing graph, each X node has an edge to each Y node in the following string of Y nodes and vice versa. Clearly the enforcing structure is equivalent to the original and uses no $x \rightarrow x$ and $y \rightarrow y$ arcs.

Can we reduce the number of $y \rightarrow x$ arcs? Yes, the enforcing structure in Figure 2.24 uses at most two $y \rightarrow x$ arcs per Y node.

It seems hard to get *NP*-completeness with at most one $y \rightarrow x$ arc per Y node. We will say more about this in Section 2.5.

However, it turns out that we can get *NP*-completeness with at most one $y \rightarrow x$ arc per Y node *and* one $x \rightarrow x$ arc per X node. This is done by using the enforcing structure in Figure 2.25.

It is not clear what happens when we allow one $y \rightarrow x$ arc and one $y \rightarrow y$ arc per Y node—or even k $y \rightarrow y$ arcs per Y node.

2.5 Pipeline Scheduling

A special case of allowing one $y \rightarrow x$ arc per y node is related to the problem of pipeline scheduling [Gro83] [HG83] [GM86].

Like very long instruction words, pipelining is a technique to keep less of a computer processor idle at any time, thus increasing the effective execution rate of programs.

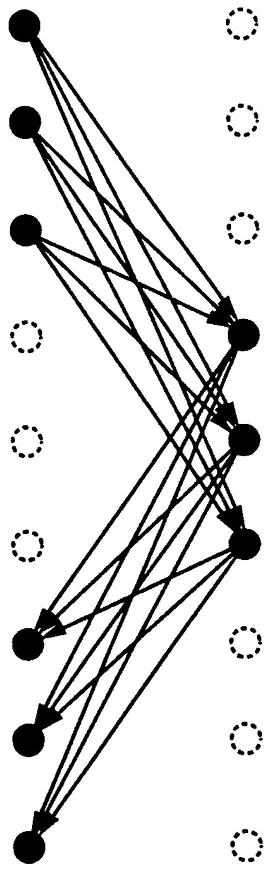


Figure 2.23: Enforcing graph for $m = 3$ with no $x \rightarrow x$ and $y \rightarrow y$ edges.

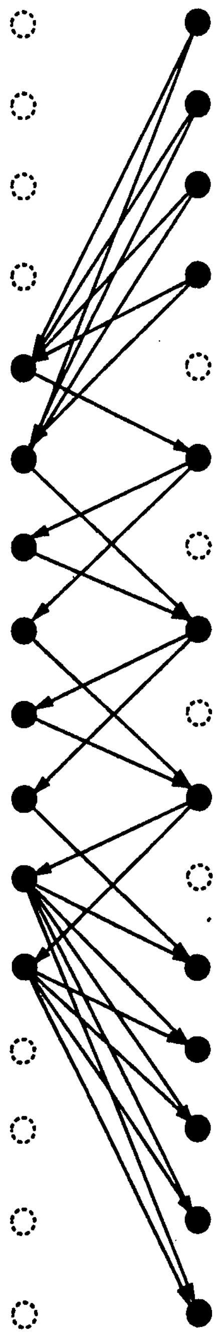


Figure 2.24: Enforcing graph for $m = 4$ with at most two $y \rightarrow x$ edges per Y node

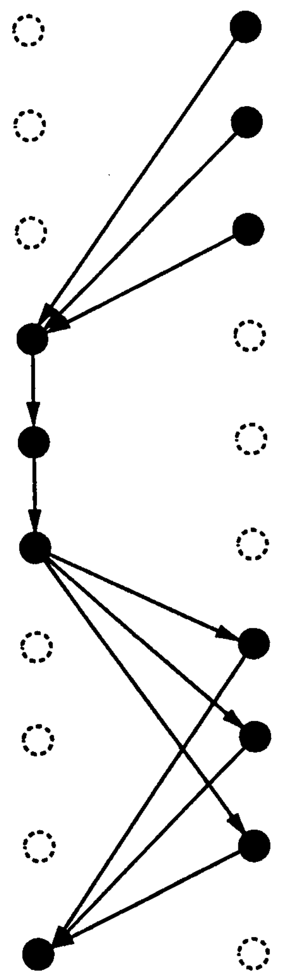


Figure 2.25: Enforcing graph for $m = 3$ with at most one $y \rightarrow x$ edge per Y node and at most one $x \rightarrow x$ edge per X node

In a pipelined computer, the processor is split up into a series of stages. During an instruction's execution, it passes from one stage to the next, freeing up the previous stage for the next instruction. The hope is that each stage of the processor can always be kept busy processing some instruction.

With pipelining, a new complication sets in. Sometimes an instruction must be prevented from starting until after some other instruction is completely precessed through all stages of the pipeline. This leaves intervening execution slots that should be filled.

Hence, one variant that arises for a k-stage pipeline is

PIPELINE-k

INSTANCE: Set X of unit-time instructions, number k of pipeline stages, partial order on X, and deadline t.

QUESTION: Is there a schedule of the $n = |X|$ tasks of length $\leq t$ such that for every pair of tasks such that x_i precedes x_j in the partial order, the instruction x_i starts at least k time units before x_j?

If $k = 1$, then we merely have the normal DAG constraints and any topological order is optimal.

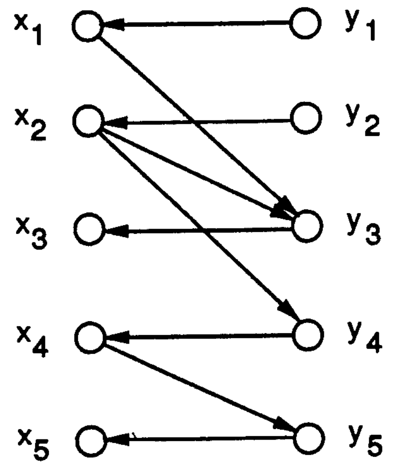
The $k = 2$ case turns out to be equivalent to the following problem:

YX-MATCH

INSTANCE: Two processors p_X and p_Y, set of unit-time jobs X, set of unit-time jobs Y with $|Y| = |X| = n$, unlimited precedence constraints of the form $x_i \rightarrow y_j$, exactly one $y \rightarrow x$ precedence constriant coming out of each Y node, exactly one $y \rightarrow x$ precedence constriant going into each X node, and a time bound t.

QUESTION: Is there a schedule of time $\leq t + 1$ satisfying the precedence constraints and such that the X jobs are only scheduled on processor p_X and the Y jobs are only scheduled on processor p_Y?

The transformations are quite simple. To go from the constraint graph of an instance of YX-MATCH (shown on the left in Figure 2.26) to the constraint graph for an instance of PIPELINE-2 (shown on the right in Figure 2.26), simply merge together nodes x_i and y_j wherever there is an edge $y_j \rightarrow x_i$.



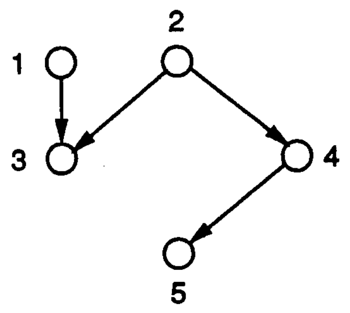


Figure 2.26: Transforming YX-MATCH to PIPELINE-2

The transformation from an instance of PIPELINE-2 to YX-MATCH is simply the reverse of this: split each node into an x^{in} and an x^{out}. Any edge $x_i \rightarrow x_j$ gets mapped to an $x_i^{\text{out}} \rightarrow x_j^{\text{in}}$ edge. Also add the edges $x_i^{\text{in}} \rightarrow x_i^{\text{out}}$. The in-nodes must execute on p_Y and the out-nodes on p_X.

Merely stating the transformations does not prove that a solution to one problem translates into a solution for the other.

Lemma 2.18 YX-MATCH and PIPELINE-2 are equivalent under polynomial-time reductions.

Proof: Assume that we are given an instance of YX-MATCH, that we transform it into an instance of PIPELINE-2 and that there is a schedule of time $\leq t$ for the PIPELINE-2 instance. Without loss of generality, let our schedule for PIPELINE-2 be $S = [x_1, \ldots, x_t]$ (x_i may refer to an idle job). Then our schedule $S' = (S'_X, S'_Y)$ for YX-MATCH is $S'_Y = [x_1^{\text{in}}, x_2^{\text{in}}, \ldots, x_t^{\text{in}}, \emptyset]$ and $S'_X = [\emptyset, x_1^{\text{out}}, x_2^{\text{out}}, \ldots, x_t^{\text{out}}]$, where x_i^{out} and x_i^{in} are the X and Y nodes that were originally collapsed together to create x_i. Since for S', x_i^{in} precedes x_i^{out}, then all $y \rightarrow x$ edges are satisfied. For any edge $x_i \rightarrow x_j$, there must be a job in S between x_i and x_j; hence there is at least one job in between x_i^{in} and x_j^{in}. Thus, x_i^{out} is executed before x_j^{in} and hence all $x \rightarrow y$ edges are satisfied. Thus we have a schedule of time $\leq t+1$ for our original instance of YX-MATCH.

Assume that we are given an instance of PIPELINE-2, that we transform it into an instance of YX-MATCH and that there is a schedule (S_X, S_Y) of time $\leq t+1$ for the

YX-MATCH instance. Let's first create an alternate schedule (S'_X, S'_Y) of the YX-MATCH instance. Let $S'_Y = S_Y$. To create S'_X, for each x_j schedule it one time unit after the y_i that has a $y_i \rightarrow x_j$ arc. Claim: $S' = (S'_X, S'_Y)$ is valid. Because of our matching of $Y \rightarrow X$ edges, no two X jobs are assigned the same execution time. Each x_j is executed no later for S'_X than it did for S_X, since in S'_X it is executed as soon as possible, given S_Y. Since X jobs are moved earlier, but Y jobs are unmoved, all $X \rightarrow Y$ constraints remain satisfied. Our specification of S'_X also satisfies all $Y \rightarrow X$ constraints. So S' is valid. To get a schedule for our original instance of PIPELINE-2, simply make $S'' = S'_Y$. The last job of S'_Y must be an idle one, so we can discard it, yielding an S'' of length $\leq t$. We just need to verify that if there is a $x_i \rightarrow x_j$ edge, these jobs will always be separated by at least one time unit. Since the YX-MATCH version had a $x_i^{\text{out}} \rightarrow x_j^{\text{in}} \rightarrow x_j^{\text{out}}$ path, then x_i^{out} and x_j^{out} must have had an intervening job. ▌

PIPELINE-2 can be solved by a polynomial time algorithm. The algorithm is based on modifications to a classic scheduling problem:

TWO PROCESSOR SCHEDULING

INSTANCE: Set X of unit-time tasks, partial order on X, and deadline t.

QUESTION: Is there a schedule of the $n = |X|$ tasks that obeys the partial order, never schedules more than two tasks for the same time slot and is of length $\leq t$?

In other words, solve the m-PROCESSOR SCHEDULING problem except that m is fixed at 2. There are a number of solutions to this problem [CG72, Gab82]. Leung, Vornberger and Witthoff [LVW84] extend these to handle the PIPELINE-2 problem — which they refer to as the "directed separation problem".

Unfortunately, it is not clear how to modify this algorithm once we start deleting the $y \rightarrow x$ arcs. This destroys an important notion used in the papers—the notion of levels—and hence the notion of a highest-level is destroyed.

Furthermore, this is about the only case of pipeline scheduling that is polynomial. Gross and Hennessy [Gro83, HG83] showed that the problem becomes NP-complete if we allow the interlocks to be mixed between lengths of 1 and 2 — where an *interlock* is the amount of intermediate time units that must separate the execution of the two instructions.

2.6 Precedence Constraints

In summary, the following are all *NP*-complete:

- $x_i \to y_j$, $x_i \to x_j$, $y_i \to y_j$, $y_i \to x_j$.
- $x_i \to y_j$, $y_i \to x_j$.
- $x_i \to y_j$, at most 2 $y_i \to x_j$ per y_i.
- $x_i \to y_j$, at most 1 $y_i \to x_j$ per y_i, at most 1 $x_i \to x_j$ per x_i.

The following are all equivalent:

- $x_i \to y_j$.
- $x_i \to y_j$, $x_i \to x_j$, $y_i \to y_j$.

The following is polynomial:

- $x_i \to y_j$, and $y_i \to x_i$ for each i.

The following is as yet unclassified:

- $x_i \to y_j$, at most 1 $y_i \to x_j$ per y_i.

Chapter 3

Well-Ordered Schedules

The relationship of triangularization to other problems is interesting in itself. However, let's turn more directly to the task of dealing with the triangularization problem proper.

3.1 Definitions

We will need some new notation before we plunge into this. The graph in Figure 3.1 will serve as an example.

Definition 3.1 The $\Gamma()$ function returns the neighbors of a node or set of nodes. So for any node z, let $\Gamma(z)$ be the set of nodes adjacent to z. Let $\Gamma(Z) = \bigcup_{z \in Z} \Gamma(z)$. In Figure 3.1, we have $\Gamma(x_2) = \{y_1, y_2, y_3\}$ and $\Gamma(\{y_1, y_2\}) = \{x_1, x_2\}$. Note that we have stopped drawing the edge directions in the drawings; in any figure, the black nodes must be executed before the adjacent white nodes are released.

Definition 3.2 For a set of nodes Z, we will sometimes want to know: if we remove the nodes in Z from the graph, what nodes become isolated—that is degree = 0. This is our function $\Gamma'(Z) = \{z \mid \Gamma(z) \subseteq Z\}$. Alternatively $\Gamma'(Z) = \overline{\Gamma(\bar{Z})}$.

So in Figure 3.1, we have $\Gamma'(x_2) = \{y_3\}$ and $\Gamma'(\{y_1, y_2\}) = \{x_1, x_5\}$.

Definition 3.3 For a schedule S of the X nodes, we will use S_k $(0 \leq k \leq m)$ to denote the subschedule consisting of the first k elements. So, if $S = [x_1, x_2, \ldots, x_m]$, then $S_k = [x_1, x_2, \ldots, x_k]$.

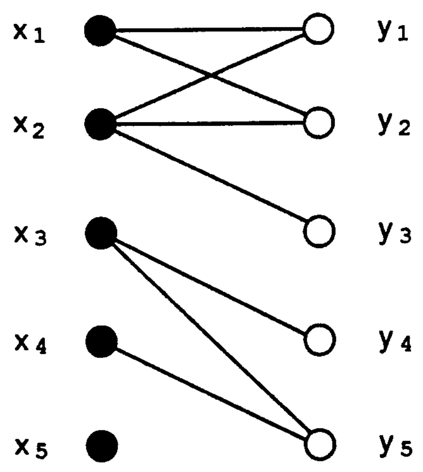


Figure 3.1: Example of constraints

We will use T_k to indicate the set of Y nodes that would be free to execute if all the jobs in S_k were executed. That is, $T_k = \Gamma'(S_k)$ for $0 \leq k \leq m$. So, if in Figure 3.1 $S = [x_1, x_2, x_3, x_4, x_5]$, then $T_3 = \{y_1, y_2, y_3, y_4\}$.

ΔT_k will be the set of Y jobs released for execution in going from S_{k-1} to S_k. That is, let $\Delta T_k = T_k \setminus T_{k-1}$ for $1 \leq k \leq m$; $\Delta T_0 = T_0$. So, continuing our example, if $S = [x_1, x_2, x_3, x_4, x_5]$, then $\Delta T_2 = \{y_1, y_2, y_3\}$.

As mentioned in Section 2.1.1 a schedule for p_X given by S_X implies a natural schedule S_Y for p_Y. Two views of this schedule were given. A third way to think of this schedule is even simpler: $[\Delta T_0, \Delta T_1, \Delta T_2, \ldots, \Delta T_m]$.

3.1.1 Charts

When a Y job isn't available yet, then p_Y must just wait for one. How long must it wait?

Definition 3.4 Let $n_k = |T_k| - |S_k|$, $0 \leq k \leq m$ and $\tilde{n}_k = |T_k| - |S_{k+1}|$, $0 \leq k < m$. Let $d_k = \min_{i=0}^{k} n_i$ and $\tilde{d}_k = \min_{i=0}^{k} \tilde{n}_k$.

The total time that p_Y must wait is $-\tilde{d}_{m-1}$. So we will call $-\tilde{d}_{m-1}$ the *delay* of S, sometimes written delay(S). If $\tilde{d}_{m-1}$ is positive, then p_Y could have started executing $\tilde{d}_{m-1}$ time units before p_X did and will not have to execute an idle job until all Y jobs are exhausted.

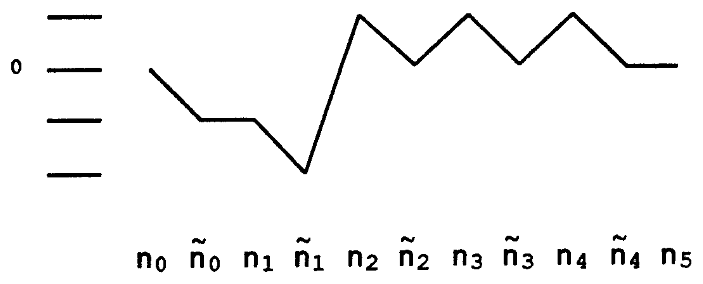


Figure 3.2: Chart corresponding to Figure 3.1

For unit time jobs, $\tilde{n}_k = n_k - 1$ and thus $\tilde{d}_{m-1} = d_{m-1} - 1$. When we generalize these definitions for non-unit time jobs, this will not necessarily be true.

If there are no isolated X nodes in the constraint graph, then $n_m \geq n_{m-1}$ and thus $d_m = d_{m-1}$. Hence if we are dealing with only unit time jobs and there are no isolated X nodes in the constraint graph, then $d_m = \tilde{d}_{m-1} + 1$. Thus, sometimes when we really want to show that an operation doesn't change the delay of a graph, instead of showing that $\tilde{d}_{m-1}$ doesn't change, we may show that d_m doesn't change. Since we can always throw away all isolated X nodes and put them at the end of any schedule that we ultimately find, the simplification of using d_m doesn't cause any problems unless we are dealing with lemmas about square matrices.

For any schedule S of the X nodes, we will draw a *chart* of the sequence differences $[n_0, \tilde{n}_0, n_1, \ldots, \tilde{n}_{m-1}, n_m]$. Figure 3.2 illustrates the chart for our example from Figure 3.1 with the schedule $S = [x_1, \ldots, x_5]$. We can read the delay of the schedule right off the chart. The delay is the negation of the lowest value that the chart dips to. In this case the chart dips to -2, so the delay is equal to 2.

Before we start it is important to note: Since the delay(S) is determined by the worst dip in the chart $(-d_{m-1})$, and since local modifications to S result in only local modifications to the chart, then all we need to do is to make sure that our modification doesn't worsen the dip of the chart in the neighborhood of the modification and that will be enough to ensure that we haven't worsened the delay. To be more specific. partition a schedule into three parts $S = [X_1, X_2, X_3]$. If all we do is modify the order of nodes within X_2, yielding $S' = [X_1, X_2', X_3]$, then the chart for X_1 clearly remains the same, since X_1 occurs before X_2. Since no modifications to the order of the jobs within X_2 can allow X_2 to relase a job previously released by X_3, and since at the end of X_2' we will have released exactly the same jobs as released at the end of X_2, then the chart for X_3 remains unchanged.

3.2 Two 1s Per Row

Let's start with a special case. Let's consider the case where our matrix has no more than two 1s per row. That is, each Y node has degree at most 2, as in Figure 3.1.

3.2.1 Component Types

The first step in understanding the case with two 1s per row is to classify the possible connected subgraphs of this class of graphs into 7 types. Each type will have a characteristic chart as shown in Figure 3.3.

Type $(-1,-1)$: an isolated X node. Looking at the chart for it gives an explanation of its name. Its worst (lowest) dip is -1 and its final value is -1.

Type $(0,1)$: an isolated Y node. Since there are no X nodes, the chart instantly reaches its final value of 1. It never dips, which we will choose to define as 0; we will see that this choice works best, but there is no choice that really suffices in all that follows. Fortunately, these nodes and the $(-1,-1)$s can be and will be discarded early.

Type $(-1,0)$: these are the connected components with the same number of X nodes and Y nodes and with at least one Y node having degree $= 1$. It follows that there is exactly one such Y node. For this type of component, there is always a way to remove an X node such that it releases a Y node and leaves behind only $(-1,0)$ subpieces. Hence the optimum chart will be a saw-tooth, bouncing back and forth between 0 and -1, ending at 0.

Type $(-1,+k)$, where $k > 0$: these are the components with $k > 0$ more Y nodes than X nodes and with at least one Y node having degree $= 1$. For this type of component, there is always a way to remove an X node such that it releases a Y node and leaves behind only subpieces of types $(0,1)$ and $(-1,l)$, where $0 \leq l \leq k$.

Type $(-2,0)$: these are the components with the same number of X nodes and Y nodes, and such that every Y node has degree $= 2$. Clearly two X nodes must be removed before a Y node can be released. There is always a choice of two X nodes whose removal releases one Y node. If the component has more than two X nodes, this leaves behind a $(-1,+1)$ piece and possibly some $(-1,0)$ pieces.

Type $(-2,+k)$, where $k > 0$: these are the components with $k > 0$ more Y nodes than X nodes and with all Y nodes having degree $= 2$. Clearly two X nodes must be removed before a Y node can be released. There is always a choice of two X nodes whose removal

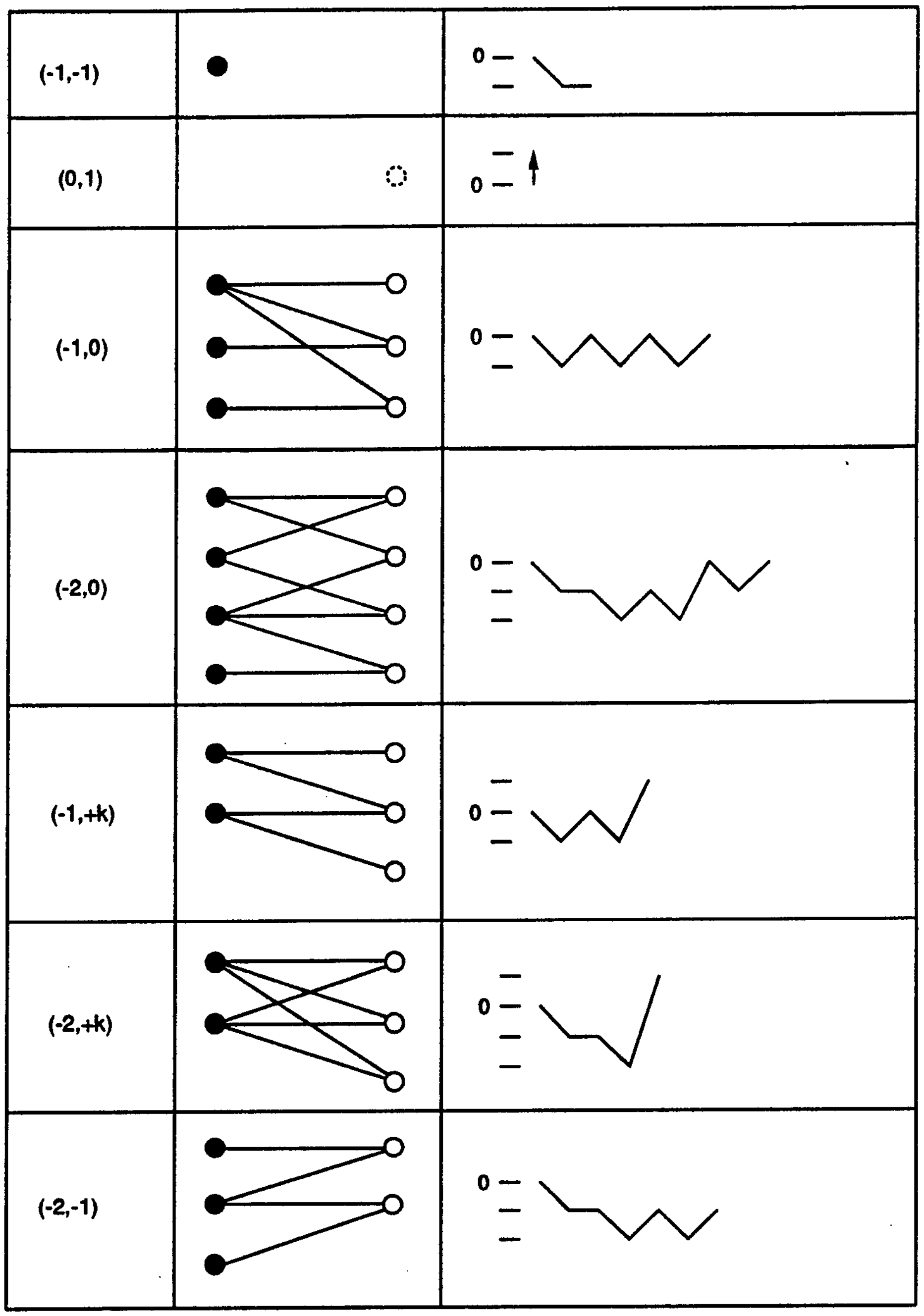


Figure 3.3: Seven types of components.

releases one Y node. Other than the just released Y nodes, the remaining graph will consist of pieces of types $(-1,0)$ and $(-1,l)$, where $0 \leq l \leq k+1$.

Type $(-2,-1)$: these are the components which have one less Y node than X nodes, yet all Y nodes have degree $= 2$. That is, they are trees whose leaves are all X nodes. Removing any leaf turns it into a $(-1,0)$ piece.

3.2.2 Ordering the Components

The intuition for finding an optimum schedule is that we want to quickly force the chart values to get high enough so that any subsequent dips in the chart won't dip down very far.

It turns out that it is always best to schedule the components in a particular order based on their types: $(0,1)$, $(-1,+k)$, $(-2,+k)$, $(-1,0)$, $(-2,0)$, $(-2,-1)$, $(-1,-1)$.

We already know why it is best to put $(0,1)$s first and the $(-1,-1)$s last—dominance. But how can we see the rest of this?

Definition 3.5 The *borrowing* of a subschedule S_k is like the delay of a schedule, except that we don't negate it, that is, it equals $\tilde{d}_{k-1}$. The *profit* of a subschedule S_k is simply $|T_k| - |S_k| = n_k$. Thus the type (b,p) of a component reflects its borrowing b and profit p.

Choosing to put the $(-1,+k)$s next after the $(0,1)$s is an example of what we call "greedy use of minimum-borrowing non-negative profit subschedules". This terminology may cause some confusion: the more that is borrowed, the more negative the borrowing. So the minimum borrowing is actually the borrowing with the largest numerical value.

For the moment we treat each connected component as an indivisible piece. In the general case this is not allowable and we will show later why this is valid for our current case of two 1s per row. First let's show why our schedule is optimal under this assumption.

We will come up with an ordering $\lhd$ on pieces of the type $t = (b,p)$ where b is the amount of borrowing we did, and p is the profit that we realized.

Assume that we have two tuples $t_1 = (b_1, p_1)$, $t_2 = (b_2, p_2)$. The $\lhd$-ordering is as follows.

- If p_1 and p_2 are positive, we have these cases: $b_1 < b_2 \Rightarrow t_1 \rhd t_2$, $b_1 = b_2 \Rightarrow t_1 \equiv t_2$, and $b_1 > b_2 \Rightarrow t_1 \lhd t_2$.

- If p_1 and p_2 are negative, we have these cases: $b_1 - p_1 < b_2 - p_2 \Rightarrow t_1 \lhd t_2$, $b_1 - p_1 = b_2 - p_2 \Rightarrow t_1 \equiv t_2$, and $b_1 - p_1 > b_2 - p_2 \Rightarrow t_1 \rhd t_2$.

- If p_1 is positive and p_2 is nonpositive or if $p_1 = 0$ and p_2 is negative, then $t_1 \lhd t_2$.

- If p_1 is nonpositive and p_2 is positive or if p_1 is negative and $p_2 = 0$, then $t_1 \rhd t_2$.
- If $p_1 = p_2 = 0$, then $t_1 \equiv t_2$.

One property of the $\lhd$-ordering that we can immediately observe is:

Lemma 3.1 $(b_1, p_1) \lhd (b_2, p_2) \iff (b_2, p_2) \rhd (b_1, p_1)$. ▮

Lemma 3.2 Given a schedule with $t_1 = (b_1, p_1)$ directly following $t_2 = (b_2, p_2)$. If $t_1 \lhd t_2$ or $t_1 \equiv t_2$, then it is never worse to swap the order and schedule t_1 directly before t_2.

Proof: The swapping operation has no global effects. So all we need to look at is the worst borrowing for each of the two orders within their local portion of the schedule. Without swapping, that is if t_2 is before t_1, then the worst borrowing is $b = \min(b_2, b_1 + p_2)$. If we swap so that t_1 is before t_2, then the worst borrowing is $b' = \min(b_1, b_2 + p_1)$. So we must show that if $t_1 \lhd t_2$ then $b' \geq b$.

Let's follow the case structure of the definition of $\lhd$.

- p_1 and p_2 are positive. So we know that $b_1 \geq b_2$. Hence, $b_1 + p_2 > b_2$ and $b = b_2$. So, $b' \geq b$ regardless of whether $b' = b_1$ or $b' = b_2 + p_1$.
- p_1 and p_2 are negative. Since $t_1 \lhd t_2$ or $t_1 \equiv t_2$, we know that $b_2 - p_2 \geq b_1 - p_1$. Hence, $b_2 > b_1 + p_2$ and $b = b_1 + p_2$. So, $b' \geq b$ regardless of whether $b' = b_1$ or $b' = b_2 + p_1$.
- p_1 is positive and p_2 is nonpositive or $p_1 = 0$ and p_2 is negative. This implies that $b_1 \geq b_1 + p_2$ and $b_2 + p_1 \geq b_2$. Hence, $b' \geq b$.
- $p_1 = p_2 = 0$. This implies that $b_1 = b_1 + p_2$ and $b_2 + p_1 = b_2$. Hence, $b' = b$.

▮

Lemma 3.3 For any set of independent (borrowing,profit) tuples, there is an optimal ordering that obeys the $\lhd$-ordering.

Proof: Assume that we have some schedule S that is $k > 0$ inversions away from the nearest $\lhd$-valid ordering. Clearly there is some place where two adjacent (b, p) tuples are out of order. By Lemma 3.2 we can swap these two tuples without degrading the schedule. The post swap schedule S' is $k - 1$ inversions away.

We can clearly repeat this until we get a $k = 0$ schedule that is at least as good as S.
∎

We have previously stated that the order should be: $(0,1)$, $(-1,+k)$, $(-2,+k)$, $(-1,0)$, $(-2,0)$, $(-2,-1)$, $(-1,-1)$. Now we can see that that was a simplification. Components of types $(-2,0)$ and $(-1,0)$ are equivalent and can be interchanged without degrading the schedule.

3.2.3 Transposition of a Chart

With what we have now defined, we can take a detour and look at what happens to a chart for a matrix G when we transpose its matrix, yielding G'. When we are looking at the chart of G, the schedule of the Y nodes is determined by the schedule of the X nodes; when we are looking at the chart of G', the schedule of the X nodes is determined by the schedule of the Y nodes. Hence, when looking at the chart of the transpose, we aren't necessarily interested in the reverse of the original schedule of the X nodes, rather we are interested in the schedule of the X nodes that is determined by the reverse of the original schedule of the Y nodes in G. This difference requires us to explore a few new lemmas.

Lemma 3.4 For a graph G, we are given a schedule $S = [x_1, \ldots, x_n]$ with profit p and maximum borrowing b. If we transpose the graph yielding G', then the schedule $S' = [y_n, \ldots, y_1]$ will have profit $p' = -p$ and maximum borrowing $b' \geq b - p$.

Proof: Since the original profit is $p = |Y| - |X|$ and the profit of the transpose is $p' = |X| - |Y|$, we can see that $p = -p$.

We can think of schedule S' as $[\Delta T_m, \Delta T_{m-1}, \ldots, \Delta T_0]$. Let's compute the borrowing at any point during the scheduling of some ΔT_i. By this point we could have scheduled at most the Y nodes $Y \setminus T_{i-1}$. We will have released at least the X nodes $x_m, x_{m-1}, \ldots, x_{i+1}$. So we have;

$$\begin{aligned} \tilde{n}' &\geq |X \setminus S_i| - |Y \setminus T_{i-1}| \\ &= |T_{i-1}| - S_i - (|Y| - |X|) \\ &= \tilde{n}_i - p \end{aligned}$$

Since b is the minimum over all $\tilde{n}_i$ and b' is the minimum over all $\tilde{n}'$, then $b' \geq b - p$. ∎

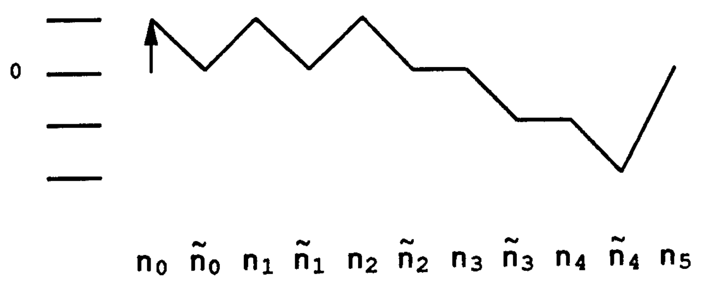


Figure 3.4: Chart corresponding to transpose of the graph in Figure 3.1

Lemma 3.5 For a graph G, we are given an *optimum* schedule $S = [x_1, \ldots, x_n]$ with profit p and maximum borrowing b. If we transpose the graph yielding G', then the schedule $S' = [y_n, \ldots y_1]$ will have profit $p' = -p$ and maximum borrowing $b' = b - p$.

Proof: By Lemma 3.4, $b' \geq b - p$. Take G' and transpose it again to yield G''. By Lemma 3.4, $b'' \geq b' - p'$. Since the original schedule S is optimal, $b'' = b$. So we have $b = b'' \geq b' - p' \geq b - p - p' = b$. Hence $b = b' - p'$ and $b' = b - p$. ▮

This extends to individual (b, p) pieces, provided that each X node in the (b, p) piece is adjacent to some Y node in the piece. Hence, to get the chart of the transpose of the graph in Figure 3.1 we first take each (b, p) of the original graph and replace it with $(b - p, -p)$ and then reverse the order of the sequence of (b, p) pairs. So for our example we start with $[(-2, 1), (-1, 0), (-1, -1)]$ as shown in Figure 3.2. This gets transformed first into $[(-3, -1), (-1, 0), (0, 1)]$ and then into $[(0, 1), (-1, 0), (-3, -1)]$.

Figure 3.4 shows the chart of the transpose of Figure 3.1 using the schedule $S_X = [y_5, y_4, y_3, y_2, y_1]$. We can see that the chart for Figure 3.4 does precisely follow the behavior $[(0, 1), (-1, 0), (-3, -1)]$, as expected.

A few other things to note. (1) Since $(0, 1)$s and $(-1, -1)$s are transposes of each other, this $(b, p) \rightarrow (b - p, -p)$ transformation implies that we chose the right name for the $(0, 1)$ pieces. (2) This transformation of sequences of (b, p) pairs is its own inverse. (3) If the original sequence of (b, p) pairs obeys the $\lhd$ ordering, then so does the transformed one, since:

Lemma 3.6 $(b_1, p_1) \lhd (b_2, p_2) \iff (b_2 - p_2, -p_2) \lhd (b_1 - p_1, -p_1)$. $(b_1, p_1) \equiv (b_2, p_2) \iff (b_2 - p_2, -p_2) \equiv (b_1 - p_1, -p_1)$. $(b_1, p_1) \rhd (b_2, p_2) \iff (b_2 - p_2, -p_2) \rhd (b_1 - p_1, -p_1)$. ▮

Proof: Let's follow the case structure of the definition of $\lhd$.

- p_1 and p_2 are positive. There are three cases. Case 1: $(b_1,p_1) \lhd (b_2,p_2) \iff b_1 > b_2 \iff (b_1-p_1)-(-p_1) > (b_2-p_2)-(-p_2) \iff (b_2-p_2,-p_2) \lhd (b_1-p_1,-p_1)$. Case 2: $(b_1,p_1) \equiv (b_2,p_2) \iff b_1 = b_2 \iff (b_1-p_1)-(-p_1) = (b_2-p_2)-(-p_2) \iff (b_2-p_2,-p_2) \equiv (b_1-p_1,-p_1)$. Case 3: $(b_1,p_1) \rhd (b_2,p_2) \iff b_1 < b_2 \iff (b_1-p_1)-(-p_1) < (b_2-p_2)-(-p_2) \iff (b_2-p_2,-p_2) \rhd (b_1-p_1,-p_1)$.

- p_1 and p_2 are negative. There are three cases. Case 1: $(b_1,p_1) \lhd (b_2,p_2) \iff b_1-p_1 < b_2-p_1 \iff (b_1-p_1) < (b_2-p_2) \iff (b_2-p_2,-p_2) \lhd (b_1-p_1,-p_1)$. Case 2: $(b_1,p_1) \equiv (b_2,p_2) \iff b_1-p_1 = b_2-p_2 \iff (b_1-p_1) = (b_2-p_2) \iff (b_2-p_2,-p_2) \equiv (b_1-p_1,-p_1)$. Case 3: $(b_1,p_1) \rhd (b_2,p_2) \iff b_1-p_1 > b_2-p_2 \iff (b_1-p_1) > (b_2-p_2) \iff (b_2-p_2,-p_2) \rhd (b_1-p_1,-p_1)$.

- p_1 is positive and p_2 is nonpositive or $p_1 = 0$ and p_2 is negative. This implies that either $-p_1$ is negative and $-p_2$ is nonnegative or $-p_1 = 0$ and $-p_2$ positive. This is equivalent to saying that either $-p_2$ is positive and $-p_1$ is nonpositive or $-p_2 = 0$ and $-p_1$ is negative. Hence both $(b_1,p_1) \lhd (b_2,p_2)$ and $(b_2-p_2,-p_2) \lhd (b_1-p_1,-p_1)$.

- p_1 is nonpositive and p_2 is positive or p_1 is negative and $p_2 = 0$. This implies that either $-p_1$ is nonnegative and $-p_2$ is negative or $-p_1$ is positive and $-p_2 = 0$. This is equivalent to saying that either $-p_2$ is nonpositive and $-p_1$ is positive or $-p_2$ is negative and $-p_1 = 0$. Hence both $(b_1,p_1) \rhd (b_2,p_2)$ and $(b_2-p_2,-p_2) \rhd (b_1-p_1,-p_1)$.

- $p_1 = p_2 = 0$. Then $-p_2 = -p_1 = 0$. Hence both $(b_1,p_1) \equiv (b_2,p_2)$ and $(b_2-p_2,-p_2) \equiv (b_1-p_1,-p_1)$.

▮

3.2.4 Indivisible Components?

Returning to our proof for two 1s per row, we still have to show that we can treat the components as indivisible. That is, we have to show that we cannot get a better schedule by allowing a schedule to start with one component and switch to a second component and then back to the first.

This is not true in the general case, when we have more than two 1s per row, as Figure 3.5 shows. The reason that intermixing schedules is an improvement in Figure 3.5 is because a positive point in the chart for one component preceded the chart's lowest point.

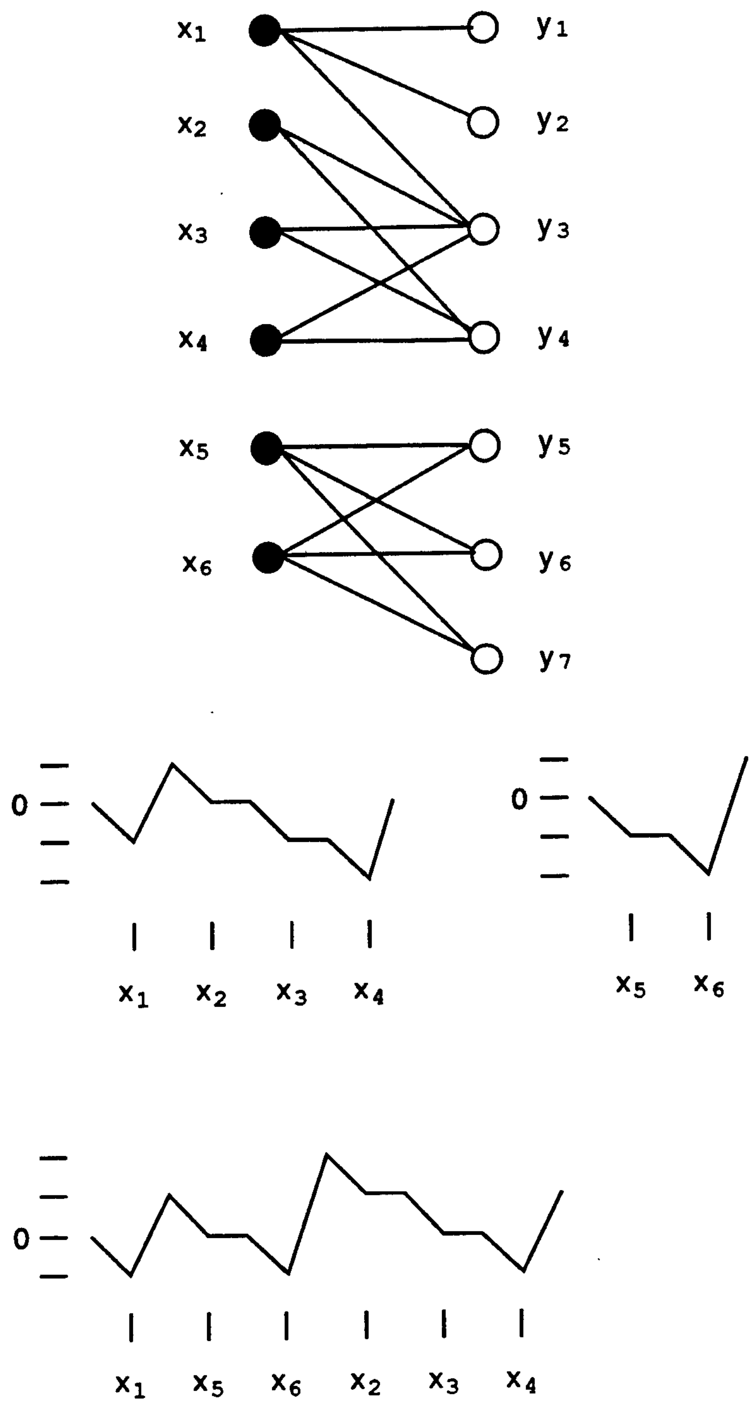


Figure 3.5: Intermixing can improve the solution.

Because of dominance, for this section we can ignore connected components with only one node.

The following lemma will help us in proving that we can treat the components as indivisible.

Lemma 3.7 Given a connected bipartite graph $G = (X, Y, E)$ in which the maximum degree of the Y nodes is ≤ 2. For every non-empty subset $\emptyset \subset X' \subseteq X$, we have $|\Gamma'(X')| - |X'| \leq |\Gamma'(X)| - |X| = n - m$.

Proof: As this will be a proof by contradiction, assume that we have a set $X' \neq \emptyset$ such that $|\Gamma'(X')| - |X'| > |\Gamma'(X)| - |X|$. Let $Y' = \Gamma'(X')$, $X'' = X \setminus X'$ and $Y'' = Y \setminus Y'$.

Consider the graph induced by the nodes $X'' \cup Y''$. It has some number, c, of connected components. Thus there must be at least $|X''| + |Y''| - c$ edges between X'' and Y''. By definition of $\Gamma'(X')$, there are no edges from X'' to $Y' = \Gamma'(X')$. Since our bipartite graph is connected, there must be at least c edges between Y'' and X'. Thus there at least $|X''|+|Y''|$ edges that have one endpoint in Y''. Since $|X''| = |X| - |X'| > |\Gamma'(X)| - |\Gamma'(X')| = |Y''|$, then there are $> 2 \cdot |Y''|$ edges that have one endpoint in $|Y''|$. But every node in $|Y''|$ is of degree ≤ 2. This is a contradiction. ▮

Corollary 3.8 Given any schedule S for a connected bipartite graph $G = (X, Y, E)$ in which the maximum degree of the Y nodes is ≤ 2. It follows that $\max_{i=1}^{m} n_i = n_m$. ▮

Corollary 3.9 Given any schedule S for a connected bipartite graph $G = (X, Y, E)$ in which the maximum degree of the Y nodes is ≤ 2. Either $\max_{i=0}^{m} n_i = n_m$ or $\max_{i=0}^{m} n_i = n_0$. ▮

Definition 3.6 For any specified schedule, we will refer to n_i as the *balance* at time i. And $\max_{i=0}^{m} n_i$ will be referred to as the *peak balance*.

So for every connected component (in the two 1s per row case), the peak balance occurs at the beginning and/or the end of the schedule. Also note for any schedule for any of the (b, p) pieces we are dealing with, the balance must dip to $b + 1$ before the balance becomes positive—this is a more complicated, and more useful, way of saying that $(-2, \pm k)$ components must schedule two X nodes before they release any Y nodes.

What happens to the peak balance when we transpose a component?

Lemma 3.10 Given a connected graph G such that for any schedule S of it $\max_{i=0}^{m} n_i = P$ and such that for any schedule S' for G', the transpose of G, we have that $\max_{i=0}^{m} n'_i \geq |X| - |Y| + P$. Then given any schedule S' for G', we have that $\max_{i=0}^{m} n'_i = |X| - |Y| + P$.

Proof: Assume to the contrary that we have a schedule S' for G' with $\max_{i=0}^{m} n'_i = n'_j > |X| - |Y| + P$. That is, $|\Gamma'(S'_j)| - |S'_j| > |X| - |Y| + P$. Then in the original graph, if we start a schedule with $X \setminus \Gamma'(S'_j)$, we release $Y \setminus S'_j$. Thus the balance at this point is:

$$\begin{aligned} |Y \setminus S'_j| - |X \setminus \Gamma'(S'_j)| &= |Y| - |X| + (|\Gamma'(S'_j)| - |S'_j|) \\ &> |Y| - |X| + (|X| - |Y| + P) \\ &= P \end{aligned}$$

This is a contradiction. ▮

Corollary 3.11 Given a connected graph G such that for any schedule S of it $\max_{i=0}^{m} n_i = n_m = |Y| - |X|$. Given any schedule S' for G', the transpose of G, we have that $\max_{i=0}^{m} n'_i = n_0 = 0$.

Corollary 3.12 Given a connected graph G such that for any schedule S of it $\max_{i=0}^{m} n_i = n_0 = 0$. Given any schedule S' for G', the transpose of G, we have that $\max_{i=0}^{m} n'_i = n'_m = |X| - |Y|$.

Corollary 3.13 If a connected graph has its peak balance at the end (beginning) of every schedule of it, then the transposed version of the graph has its peak balance at the beginning (end) of every schedule of it.

In talking about intermixing of multiple components, complicated notation seems unavoidable. We will have a number of components $C_1, C_2, \ldots, C_k$. Each one will have a schedule $S^1, S^2, \ldots, S^k$. Let $n^i_j = |\Gamma'(S^i_j)| - |S^i_j|$. There will be a schedule S which is an intermixing of the schedules $S^1, S^2, \ldots, S^k$. Let $u(i,j) =$ the number of X nodes from S^i that are scheduled before or during time j in schedule S. Hence, $n_j = \sum_{i=1}^{k} n^i_{u(i,j)}$.

Lemma 3.14 Given k components $C_1, \ldots, C_k$ such that (1) every component either has its peak balance at the end of every possible schedule for it or it has its peak balance at the beginning of every possible schedule for it (2) at least one component has its peak balance at the end of every possible schedule for it and (3) any schedule of any (b_i, p_i) component C_i dips to a balance of $b_i + 1$ before the balance becomes positive; given schedules $S^1, \ldots, S^k$ for the components and given an intermixed schedule S of the components. There exists a schedule S' that has only $k-1$ of the components intermixed and the other component scheduled first, such that S' has a delay no worse than S.

Proof: Without loss of generality, assume that C_1 is a minimum component with respect to the $\lhd$ ordering. Let S' be a schedule with C_1 scheduled first and scheduled optimally, followed by $C_2, \ldots, C_k$ intermixed in the same order as they are in S.

Let m_1 = the number of X nodes in C_1. If the j-th X node in S is not part of C_1, then in S' it will be scheduled in position $j + m_1 - u(1,j)$. In S', for $j' > m_1$, $n'_{j'} = \sum_{i=1}^{k} n'^{i}_{j'} = n'^{1}_{m_1} + \sum_{i=2}^{k} n^{i}_{u(i,j)}$, where j is such that $j' = j + m_1 - u(1,j)$. We claim that $n'^{1}_{m_1} = n^{1}_{m_1} \geq n^{1}_{u(1,j)}$ and hence $n'_{j'} \geq n_j$. We can see that $n'^{1}_{m_1} = n^{1}_{m_1}$, since they are both the balance at the end of schedules of C_1. Claim: Since C_1 is minimum with respect to $\lhd$ and at least one component of $C_1, \ldots, C_k$ has its peak balance at the end of every possible schedule for it, then C_1 is one such component and hence $n^{1}_{m_1} \geq n^{1}_{u(1,j)}$ for all j. To see this, note that for all components that satisfy condition (1) in the statement of the lemma: (a) a component has its peak balance at the end of every possible schedule for it iff the component has a nonnegative profit (b) a component has its peak balance at the beginning of every possible schedule for it iff the component has a nonpositive profit. Hence, some component has a nonnegative profit. From the definition of $\lhd$, it follows that C_1 has a nonnegative profit. Hence, C_1 has its peak balance at the end of every possible schedule for it.

Now all we need to show is that $\min_{j=1}^{m_1} n'_j \geq \min_{j=1}^{m} n_j$. To see this, let's look at S. In particular, let's look at the smallest value of l such that $n^{i'}_{u(i',l)} = b_{i'} + 1$ for some $S^{i'}$ and $p_{i'} \geq 0$. Since every component reaches a balance of $b_i + 1 \leq 0$ before its balance goes positive, then $n^{i}_{u(i,l)} \leq 0$ for all $i \neq i'$. Hence $\min_{j=1}^{m} n_j \leq n_l = \sum_{i=1}^{k} n^{i}_{u(i,l)} \leq n^{i'}_{u(i',l)}$. Since we have scheduled C_1 optimally in S' and since C_1 is minimum with respect to $\lhd$, $n^{i'}_{u(i',l)} = b_{i'} + 1 \leq b_1 + 1 = \min_{j=1}^{m_1} n'_j$ and hence $\min_{j=1}^{m_1} n'_j \geq \min_{j=1}^{m} n_j$. ▮

Because of Corollary 3.13 and Lemma 3.6, transposition gives us:

Corollary 3.15 Given k components $C_1, \ldots, C_k$ such that (1) every component either has its peak balance at the end of every possible schedule for it or it has its peak balance at the beginning of every possible schedule for it (2) at least one component has its peak balance at the *beginning* of every possible schedule for it and (3) any schedule of the transpose of any (b_i, p_i) component C_i dips to a balance of $b_i - p_i + 1$ before the balance becomes positive; given schedules $S^1, \ldots, S^k$ for the components and given an intermixed schedule S of the components. There exists a schedule S' that has only $k-1$ of the components intermixed and the other component scheduled *last*, such that S' has a delay no worse than S. ∎

Lemma 3.16 Given k components, each with less than two 1s per row. There is an optimal schedule that does not intermix the components.

Proof: This is trivially true for the base case of the induction, when $k = 1$, since there is no intermixing possible.

Otherwise, assume that there is an optimal schedule S that has intermixing. For $k > 1$, either there exists a component which has its peak balance at the end of every possible schedule for it, or there doesn't. If such a component exists, then Lemma 3.14 tells us that we can find a schedule S' that begins with one component C separated out from the others and has $\leq k-1$ of the components intermixed and such that delay(S') $\leq$ delay(S). Hence S' is also optimal. By induction on $G \setminus C$, we obtain an optimal schedule S'' of $G \setminus C$ that does not intermix the $\leq k-1$ intermixed components of $G \setminus C$. We can append S'' to the optimal schedule of C, yielding an optimal schedule of the whole graph that does not intermix components.

If there does not exists a component which has its peak balance at the end of every possible schedule for it, then for every component, every schedule has its peak balance at the beginning. Thus we must have only $(-2,-1)$ components. Any schedule of the transpose of a $(-2,-1)$ reaches a balance of $-2-(-1)+1 = 0$ before the balance becomes positive. Thus all the preconditions of Corollary 3.15 hold. Once again, this corollary gives us a schedule S' that has one less intermixing than S and then we use induction on the remaining intermixed portion of S' to yield an optimal schedule with no intermixing. ∎

3.3 Merging

We have seen that when there are only two 1s per row, we can independently solve the separate connected components and then merely append the solutions in an appropriate order to yield the overall solution. As Figure 3.5 showed, this does not apply for the more general case. Thus a question arises: In the general case, can we solve separate pieces independently and then perform some simple method of combining the solutions? Or must we view all separate pieces at the same time, as we try to solve the problem?

3.3.1 Peak Balance

Our first intuitive observation is that the $\lhd$-ordering works because it gets the balance as high as possible so that subsequent dips will not go very far negative.

Let's apply this intuition of goodness to a single component. What is the largest possible peak balance? Can it be achieved by an optimum schedule?

Lemma 3.17 For a graph with n rows and m columns and with a maximum matching M of the graph, the peak balance is at most $n - |M|$.

Proof: Claim: for any set $Y' \subseteq Y$, $|Y'| - |\Gamma(Y')| \leq |Y| - |M|$. To see this: Consider, $Y' \cap M$ and $\Gamma(Y') \cap M$. Clearly for each distinct node in the first there is a distinct node in the second that is adjacent to it. That is, $|Y' \cap M| \leq |\Gamma(Y') \cap M|$. Hence, $|Y'| - |\Gamma(Y')| \leq (|Y'| - |Y' \cap M|) - (|\Gamma(Y')| - |\Gamma(Y') \cap M|) \leq |Y'| - |Y' \cap M| \leq |Y| - |M|$.

Claim: for any set $X' \subseteq X$, $|\Gamma'(X')| - |X'| \leq |Y| - |M|$. To see this: first note that $\Gamma(\Gamma'(X')) \subseteq X'$. Thus, $|\Gamma'(X')| - |X'| \leq |\Gamma'(X')| - |\Gamma(\Gamma'(X'))| \leq |Y| - |M|$.

Since the balance of a schedule S is $n_k = |\Gamma'(S_k)| - |S_k|$ and since $n = |Y|$, we see that $n_k \leq n - |M|$ for any k and any schedule S. ∎

The more interesting claim is that for every graph, there is a minimum delay schedule that has a peak balance of $n - |M|$. To show this we need a more complicated variant of our "inversion" strategy used in Lemma 3.3. The added complication comes from the fact that the pieces that we will be shifting around will no longer be independent. Instead, they will have edges to common nodes. Thus, once again, we will need to get more mathematical in order to derive a few useful lemmas.

3.3.2 Basic Lemmas

For the rest of this chapter, assume that all nodes in X have at least one neighbor, since any nodes in X with no neighbors are optimally placed at the end of the schedule. As noted earlier, this sometimes allows us to concentrate on d_m instead of $\tilde{d}_{m-1}$.

The first lemma tells us the obvious: if we enlarge (shrink) the set of nodes removed from one side, we enlarge (shrink) the set of isolated nodes on the other. It also gives us an explicit formula relating these sets of isolated nodes.

Lemma 3.18 For any two sets of nodes such that $Z_1 \subseteq Z_2$, we have $\Gamma'(Z_1) \subseteq \Gamma'(Z_2)$ and $\Gamma'(Z_1) = \Gamma'(Z_2) \setminus \Gamma(Z_2 \setminus Z_1)$.

Proof: If $z \in \Gamma'(Z_1)$ then $\Gamma(z) \subseteq Z_1 \subseteq Z_2$. Hence, $z \in \Gamma'(Z_2)$ and $\Gamma'(Z_1) \subseteq \Gamma'(Z_2)$.

If $z \in \Gamma'(Z_2) \setminus \Gamma(Z_2 \setminus Z_1)$ then it is not adjacent to any node in $Z_2 \setminus Z_1$. Since $\Gamma(z) \subseteq Z_2$, then $\Gamma(z) \subseteq Z_1$ and $z \in \Gamma'(Z_1)$.

If $z \in \Gamma'(Z_1)$ then $\Gamma(z) \subseteq Z_1$. Since we've already shown that $\Gamma'(Z_1) \subseteq \Gamma'(Z_2)$, we have $z \in \Gamma'(Z_2)$. Since z is not adjacent to $Z_2 \setminus Z_1$, we must have $z \in \Gamma'(Z_2) \setminus \Gamma(Z_2 \setminus Z_1)$. Hence, $\Gamma'(Z_1) = \Gamma'(Z_2) \setminus \Gamma(Z_2 \setminus Z_1)$. ▮

The next three lemmas help us nail down a specific formula for ΔT_i.

Lemma 3.19 Given a schedule $S = [x_1, x_2, \ldots, x_m]$ and a node $y \in Y$, we have $y \in \Delta T_i$ if and only if i is the largest integer such that $x_i \in \Gamma(y)$.

Proof: Let i be the largest integer such that $x_i \in \Gamma(y)$. Clearly $\Gamma(y) \subseteq S_i$, hence $y \in \Gamma'(S_i) = T_i$. Since $x_i \in \Gamma(y)$, $\Gamma(y) \not\subseteq S_{i-1}$. Thus, $y \notin \Gamma'(S_{i-1}) = T_{i-1}$. So $y \in \Delta T_i$.

Since each node $y \in Y$ is in exactly one ΔT_i, the other direction of the implication is immediate. ▮

The function $\Gamma'(X')$ tells us what Y jobs will be released by the jobs in X' if they are the only jobs executed. What happens when X' is intermixed with other X jobs in some schedule? This causes the set of Y jobs released by the jobs in X' to vary. Nevertheless, this set can be bounded on both sides.

Lemma 3.20 For any schedule $S = [x_1, \ldots, x_m]$, and any set $X' \subseteq X$, we have

$$\Gamma'(X') \subseteq \bigcup_{x_i \in X'} \Delta T_i \subseteq \Gamma(X').$$

Proof: Choose any $y \in \Gamma'(X')$. Let i be the largest integer such that $x_i \in \Gamma(y)$. By Lemma 3.19, $y \in \Delta T_i$. Since $\Gamma(y) \subseteq X'$, we have $x_i \in X'$ and hence $y \in \Delta T_i \subseteq \bigcup_{x_i \in X'} \Delta T_i$.

Choose any $y \in \Delta T_i$, with $x_i \in X'$. By Lemma 3.19, $x_i \in \Gamma(y)$. Hence $y \in \Gamma(x_i) \subseteq \Gamma(X')$. ▮

Lemma 3.21 For any schedule $S = [x_1, \ldots, x_m]$, we have $\Delta T_i = \Gamma'(S_i) \bigcap \Gamma(x_i)$.

Proof: Since $T_i = \Gamma'(S_i)$, $\Delta T_i \subseteq \Gamma'(S_i)$. By Lemma 3.20, $\Delta T_i \subseteq \Gamma(x_i)$. Hence, $\Delta T_i \subseteq \Gamma'(S_i) \bigcap \Gamma(x_i)$.

If $y_i \in \Gamma'(S_i)$ then $y_i \in T_i$. If $y_i \in \Gamma(x_i)$ then $y_i \notin T_{i-1}$. Hence, $y_i \in \Delta T_i$ and $\Delta T_i \supseteq \Gamma'(S_i) \bigcap \Gamma(x_i)$. ▌

Lemma 3.22 For any two sets of nodes such that $X_1 \subseteq X_2$ and any schedule $S = [x_1, x_2, \ldots, x_m]$ of X_2, we have

$$\Gamma'(X_1) \subseteq \Gamma'(X_2) \setminus \bigcup_{x_i \in X_2 \setminus X_1} \Delta T_i \subseteq \Gamma'(X_2) \setminus \Gamma'(X_2 \setminus X_1).$$

Proof: By Lemma 3.18, $\Gamma'(X_1) \subseteq \Gamma'(X_2) \setminus \Gamma(X_2 \setminus X_1)$. By Lemma 3.20, $\bigcup_{x_i \in X_2 \setminus X_1} \Delta T_i \subseteq \Gamma(X_2 \setminus X_1)$. Hence, $\Gamma'(X_1) \subseteq \Gamma'(X_2) \setminus \bigcup_{x_i \in X_2 \setminus X_1} \Delta T_i$. By Lemma 3.20, $\Gamma'(X_2 \setminus X_1) \subseteq \bigcup_{x_i \in X_2 \setminus X_1} \Delta T_i$. Hence, $\Gamma'(X_2) \setminus \bigcup_{x_i \in X_2 \setminus X_1} \Delta T_i \subseteq \Gamma'(X_2) \setminus \Gamma'(X_2 \setminus X_1)$. ▌

3.3.3 Segregation

To go further we also need a new tool. Our new tool is called *segregation*. From a schedule S, say $S_X = [x_1, x_3, x_5, x_4, x_6, x_2]$, we will modify it to form a segregated schedule S'. The way this works is this: First, we assign each node x_i a label from a set of labels; for this example, let our label set be $\{\texttt{one}, \texttt{two}, \texttt{three}\}$ and let $L(x_1) = \texttt{one}$, $L(x_2) = \texttt{one}$, $L(x_3) = \texttt{two}$, $L(x_4) = \texttt{three}$, $L(x_5) = \texttt{three}$, $L(x_6) = \texttt{two}$. The set of labellings has an ordering, say $[\texttt{one}, \texttt{two}, \texttt{three}]$.

To form the segregated schedule we first use all the nodes of the first label, then all the nodes of the next label, and so on, until all the nodes are scheduled. All the nodes of a particular label will remain in the relative order that they have in the schedule S. So in our example, the result will be $S'_X = [x_1, x_2, x_3, x_6, x_5, x_4]$.

Like our use of inversion in the two 1s per row case, we will perform these segregations in ways that won't degrade the schedule. So, sometimes we wish to impose two conditions:

- The segregated schedule always has a delay no larger than the original schedule.
- The labels can be assigned efficiently without knowing an optimal schedule.

Then we can use the labels to divide our problem into smaller scheduling subproblems, one per label. Optimal solutions to the subproblems can then just be concatenated to yield an optimal solution to our original problem.

Let $S = [x_1, \ldots, x_m]$ be a schedule. Applying segregation to it will result in some other schedule $S' = [x'_1, \ldots, x'_m]$. A specific segregation of a specific schedule S can be viewed as a permutation π; that is, $x_i = x'_{\pi(i)}$.

Lemma 3.23 If $S_i \subseteq S'_{\pi(i)}$ then $\Delta T_i \subseteq \Delta T'_{\pi(i)}$. If $S_i \supseteq S'_{\pi(i)}$ then $\Delta T_i \supseteq \Delta T'_{\pi(i)}$.

Proof: Since $\Gamma(x_i) = \Gamma(x'_{\pi(i)})$ and since Lemma 3.18 tells us that $S_i \subseteq S'_{\pi(i)}$ implies $\Gamma'(S_i) \subseteq \Gamma'(S'_{\pi(i)})$, we know that $\Gamma'(S_i) \bigcap \Gamma(x_i) \subseteq \Gamma'(S'_{\pi(i)}) \bigcap \Gamma(x'_{\pi(i)})$. By Lemma 3.21, this tells us that $\Delta T_i \subseteq \Delta T'_{\pi(i)}$. ∎

We can always think of more complex segregations as a composition of two-label segregations, that is segregations with $|L| = 2$. Let that label set be $L = \{\texttt{one}, \texttt{two}\}$; we will use the segregation $[\texttt{one}, \texttt{two}]$. Lemma 3.23 tells us that for two-label segregations, if x_i has label two then $\Delta T_i \subseteq \Delta T'_{\pi(i)}$ and if x_i has label one then $\Delta T_i \supseteq \Delta T'_{\pi(i)}$. The following three lemmas give us a set of conditions when we can be sure that applying a two-label segregation doesn't increase the delay.

Lemma 3.24 Let f be the number of X nodes labelled one. If applying a two-label segregation with the order $[\texttt{one}, \texttt{two}]$ to a schedule S yields a segregated schedule S' with $n'_f = \max_{i=0}^{f} n'_i$, then $n'_{\pi(i)} \geq n_i$ for all x_i with label two.

Proof: Consider any x_i node labeled two. Clearly, $S_i = S'_{\pi(i)} \setminus (S'_f \setminus S'_{\pi(j)})$, where x_j is the one node most recently preceding x_i in the schedule S. (If x_j is non-existent, then using $\pi(j) = 0$ makes the following reasoning still applicable.) Since $S_i \subseteq S'_{\pi(i)}$, applying Lemma 3.22 with schedule S' tells us that

$$\begin{aligned} T_i &= \Gamma(S_i) \\ &\subseteq \Gamma'(S'_{\pi(i)}) \setminus \cup_{x'_i \in (S'_f \setminus S'_{\pi(j)})} \Delta T'_i \\ &= \Gamma'(S'_{\pi(i)}) \setminus (\Gamma'(S'_f) \setminus \Gamma'(S'_{\pi(j)})) \\ &= T'_{\pi(i)} \setminus (T'_f \setminus T'_{\pi(j)}). \end{aligned}$$

Hence,

$$\begin{aligned} n_i &= |T_i| - |S_i| \\ &\leq |T'_{\pi(i)} \setminus (T'_f \setminus T'_{\pi(j)})| - |S'_{\pi(i)} \setminus (S'_f \setminus S'_{\pi(j)})| \\ &= (|T'_{\pi(i)}| - |T'_f| + |T'_{\pi(j)}|) - (|S'_{\pi(i)}| - |S'_f| + |S'_{\pi(j)}|) \end{aligned}$$

$$
\begin{aligned}
&= n'_{\pi(i)} - n'_f + n'_{\pi(j)} \\
&\leq n'_{\pi(i)}.
\end{aligned}
$$

▮

Lemma 3.25 Let f be the number of X nodes labelled one. If applying a two-label segregation with the order [one, two] to a schedule S yields a segregated schedule S' with $n'_f = \max_{i=f}^{m} n'_i$, then $n'_{\pi(i)} \geq n_i$ for all x_i with label one.

Proof: Consider any x_i node labeled one. Clearly, $S_i = S'_{\pi(j)} \setminus (S'_f \setminus S'_{\pi(i)})$, where x_j is the two node most recently preceding x_i in the schedule S. (If x_j is non-existent, then $S_i = S'_{\pi(i)}$ and $T_i = T'_{\pi(i)}$, and hence $n_i = n'_{\pi(i)}$.) Since $S_i \subseteq S'_{\pi(j)}$, applying Lemma 3.22 with schedule S' tells us that $T_i \subseteq T'_{\pi(j)} \setminus (T'_f \setminus T'_{\pi(i)})$. Hence,

$$
\begin{aligned}
n_i &= |T_i| - |S_i| \\
&\leq |T'_{\pi(j)} \setminus (T'_f \setminus T'_{\pi(i)})| - |S'_{\pi(j)} \setminus (S'_f \setminus S'_{\pi(i)})| \\
&= (|T'_{\pi(j)}| - |T'_f| + |T'_{\pi(i)}|) - (|S'_{\pi(j)}| - |S'_f| + |S'_{\pi(i)}|) \\
&= n'_{\pi(j)} - n'_f + n'_{\pi(i)} \\
&\leq n'_{\pi(i)}.
\end{aligned}
$$

▮

Lemma 3.26 Let f be the number of X nodes labelled one. If applying a two-label segregation with the order [one, two] to a schedule S yields a segregated schedule S' with $n'_f = \max_{i=0}^{m} n'_i$, then $d'_m \geq d_m$.

Proof: Lemmas 3.24 and 3.25 both apply and hence we know that $n_i \leq n'_{\pi(i)}$ for all i. It is thus clear that $d_m \leq d'_m$. ▮

We will explore some of the power of this lemma later. A simpler, but still useful segregation lemma is:

Lemma 3.27 Applying a two-label segregation to a schedule S that assigns the label one to a single node $x_k \in X$ such that $\Gamma(x_k) \supseteq T_k$ and the label two to all the other nodes in X results in a schedule with $d'_m \geq d_m$.

Proof: If $T_k = \emptyset$, $d_k = -k = d'_k$. If $T_k \neq \emptyset$, then $T_{k-1} = \emptyset$ by hypothesis, hence $d_k = 1 - k$ and $d'_k \geq 1 - k$. Hence, $d'_k \geq d_k$. For all $i > k$, $x_i = x'_{\pi(i)}$. Hence, $S_i = S'_{\pi(i)}$, $T_i = T'_{\pi(i)}$ and $n_i = n'_i$. Thus $d'_m \geq d_m$. ∎

Lemma 3.27 helps us reaffirm something that we already knew because of dominance:

Corollary 3.28 Applying to a schedule S a two-label segregation that assigns the label one to a set of nodes $x_k \in X$ such that $\Gamma(x_k) = Y$ and the label two to all the other nodes in X results in a schedule with $d'_m \geq d_m$. ∎

Also, Lemma 3.27 helps us to validate the legitimacy of making certain local modifications to schedules.

Lemma 3.29 Given a schedule $S = [x_1, \ldots, x_m]$. If $\Delta T_{i+1} = \Delta T_{i+2} = \ldots = \Delta T_{k-1} = \emptyset$ and $\Delta T_k \neq \emptyset$, then the schedule

$$S' = [x_1, \ldots, x_i, x_k, x_{i+1}, \ldots, x_{k-1}, x_{k+1}, \ldots, x_m]$$

will have $d'_m \geq d_m$.

Proof: Apply Lemma 3.27 to the graph $G \setminus (S_i \cup T_i)$ and the schedule $S \setminus S_i$. ∎

There is another use for segregations. Sometimes they allow us to find an optimum schedule for a subgraph, given an optimum schedule for the whole graph.

Lemma 3.30 Let f be the number of X nodes labelled one. Let S be an optimal schedule, and let the two-label segregation yield a schedule S'. Let $\bar{S}'_f = [x'_{f+1}, \ldots, x'_m]$. If $\tilde{d}'_{m-1} = \tilde{d}_{m-1}$ and $\tilde{d}'_{f-1} > \tilde{d}'_{m-1}$, then $\bar{S}'_f$ is an optimal schedule for the graph G' induced by the nodes $(S'_m \setminus S'_f) \bigcup (T'_m \setminus T'_f)$.

Proof: Assume that $\bar{S}'_f$ is not optimal for G'. This means that there is a schedule $\bar{S}''_f$ that is better. Then the schedule $[S'_f, \bar{S}''_f]$ would be a better schedule than S' and S. This contradicts our assumptions on S and thus $\bar{S}'_f$ is optimal for G'. ∎

3.3.4 Decompose

Let's show a method to assign labels from the label set $L = \{\texttt{ascent}, \texttt{plateau}, \texttt{descent}\}$ to the x_i nodes *without* knowing an optimum schedule, and in such a way that given any optimal schedule, the corresponding segregated schedule will not have a larger delay. For this purpose we assume that a maximum matching of the graph $G(X,Y)$ has been given.

Definition 3.7 An *alternating path* is a path that alternates between edges in a matching and edges not in the matching. That is, it never consecutively uses two edges not in the matching.

To simplify talking about this decomposition, we will modify the bipartite graph. The edges are normally all oriented from X to Y. Whenever an edge is in the matching, we will reverse the orientation so that the edge will point from Y to X. Thus alternating paths now correspond to directed paths in the modified graph.

First find any maximum matching, M, of the bipartite graph. There are $|X| - |M|$ unmatched x_is and $|Y| - |M|$ unmatched y_is. The ascent is composed of the $|Y| - |M|$ unmatched y_is and any nodes that can reach them via directed paths in the modified graph. The descent is composed of the $|X| - |M|$ unmatched x_is and any nodes reachable from them via directed paths in the modified graph. The plateau is composed of any nodes that aren't reachable from and can't reach to any unmatched node via a directed path in the modified graph. (Note that there cannot be a directed path between unmatched vertices, since M is maximum.)

This seems like a non-deterministic labeling, since we say "find *any* maximum matching." Actually, the labeling is unique.

Lemma 3.31 The node labelings are invariant under different maximum matchings.

Proof: By *inverting* a path, we mean changing the matching edges on it into non-matching edges and vice-versa. In our modified graph, this corresponds to changing the orientations of every edge on the path. It's a well known result in graph theory that any maximum matching can be converted into any other by a sequence of operations of the form: either invert an even length alternating path starting or ending at an unmatched node, or invert an alternating cycle.

If we invert a directed cycle we haven't changed the set of unmatched nodes, nor have we changed the reachability from any node to any other node. So clearly the labelings don't change.

If we invert a directed even length path, then we do change the set of unmatched nodes. There are two mutually exclusive cases: either the first node of our directed path is an unmatched X node or the last node of our directed path is an unmatched Y node.

Say that the first node is an unmatched X node. Then after inversion, this previously unmatched node will become matched and we will instead have a new unmatched X node at the end of the original directed path. Thus this new unmatched X node is at the beginning of the inverted directed path and all nodes on the path are now reachable from this new unmatched X node. Hence the labeling will remain the same.

Similarly, if there originally was an unmatched Y node at the end of the original directed path, then after inversion there will be an unmatched Y node at the end of the inverted directed path, and hence the labeling will remain unchanged. ▮

3.3.5 APD-Segregated Schedule

Let $S = [x_1, \ldots, x_m]$ be an optimum schedule. Segregate the schedule in the order: [`ascent`, `plateau`, `descent`]. We will call such orderings *APD-segregated schedulings.*

The following lemma, while almost equivalent to Lemma 3.17, is more constructive.

Lemma 3.32 Given a bipartite graph $G = (X, Y, E)$ with a maximum matching M, we have $\min_{X' \subseteq X}(|X'| - |\Gamma'(X')|) = |M| - |Y|$.

Proof: Let V be the set of unmatched nodes of Y plus all nodes from which they can be reached by alternating paths. Let $X' = V \bigcap X$. Then $\Gamma'(X')$ consists of the unmatched nodes of Y plus the nodes matched with X'; hence $|X'| - |\Gamma'(X')| = |M| - |Y|$.

If X' is any subset of X, if $y \in Y$ is in $\Gamma'(X')$ and if $(x, y) \in M$, then $x \in X'$. Hence $|\Gamma'(X')| - |X'| \leq |Y| - |M|$. So, $|M| - |Y|$ is the smallest possible value. ▮

Lemma 3.33 For any schedule $S = [x_1, \ldots, x_m]$, the APD-segregated schedule $S' = [x'_1, \ldots, x'_m]$ corresponding to S has $d'_m \geq d_m$.

Proof: First, let us add the label p-or-d to every node with the label plateau or descent. Let us show that for any schedule, the segregated schedule using the order [ascent, p-or-d] is at least as efficient as the original schedule S.

Let f be the number of X nodes labeled with ascent. By Lemma 3.32, $\max_{i=0}^{m} n'_i = \max_{i=0}^{m}(|\Gamma'(S'_i)| - |S'_i|) \leq |Y| - |M|$. Since $n'_f = |Y| - |M|$, we can see that the maximum is achieved at $\max_{i=0}^{m} n'_i = n'_f$. Hence Lemma 3.26 tells us that $d'_m \geq d_m$.

Instead, let us add the label a-or-p to every node with the label ascent or plateau. Again we will show that for any given schedule, if we segregated schedule using the order [a-or-p, descent] we get a new schedule at least as efficient as the original schedule S.

Let f be the number of X nodes labeled with a-or-p. By Lemma 3.32, $\max_{i=0}^{m} n'_i = \max_{i=0}^{m}(|\Gamma'(S'_i)| - |S'_i|) \leq |Y| - |M|$. Since $n'_f = |Y| - |M|$, we can see that the maximum is achieved at $\max_{i=0}^{m} n'_i = n'_f$. Hence Lemma 3.26 tells us that $d'_m \geq d_m$.

Since both of the above segregations work for any initial schedule S, they can be composed. By composing the two segregations, we see that a [ascent, plateau, descent] segregation of a schedule S yields a schedule S' with $d'_m \geq d_m$. ▮

By segregating, we have reduced our problem to a few more specialized ones. If we solve each of them optimally, we will have solved our original problem optimally.

ASCENT

INSTANCE: A time bound t and a bipartite graph $G = (X, Y, E)$ such that $|X| < |Y|$ and such that it has a maximum matching of size $|X|$ and such that every node is reachable from one of the $|Y| - |X|$ unmatched nodes by an alternating path.

QUESTION: Is there a valid PLT schedule of length $\leq t$?

DESCENT

INSTANCE: A time bound t and a bipartite graph $G = (X, Y, E)$ such that $|Y| < |X|$ and such that it has a maximum matching of size $|Y|$ and such that every node is reachable from one of the $|X| - |Y|$ unmatched nodes by an alternating path.

QUESTION: Is there a valid PLT schedule of length $\leq t$?

PLATEAU

INSTANCE: A time bound t and a bipartite graph $G = (X, Y, E)$ such that $|X| = |Y|$ and such that it has a perfect matching.
QUESTION: Is there a valid PLT schedule of length $\leq t$?

The plateau can be further segregated. To see this consider the modified version of the graph which has the edges from a maximum matching oriented from Y to X. Decompose the plateau into a DAG of strongly-connected components (SCCs) and topologically number the components. Like our APD-decomposition, this decomposition also won't vary with different maximum matchings. To see this, we again need to consider what happens when we invert paths and cycles. Any directed even length path that starts at an unmatched X node or ends at an unmatched Y node cannot extend into the plateau, so inverting such a path doesn't affect the plateau at all. Any directed cycle is either wholly outside the plateau or wholly inside a single strongly-connected component of the plateau, and thus inverting such a cycle won't affect the DAG/SCC structure of the plateau.

If the decomposition gives k components, then we can do the following $k-1$ segregations: for $i = 1$ to $k-1$, label components 1 through i with one and $i+1$ through k with two. These segregations will satisfy Lemma 3.26 and hence we have not harmed anything.

Figure 3.6 shows a sample decomposition of a graph. Note that one of the two "SCC"s of the plateau is degenerate: the plateau piece $\{x_{14}, y_{19}\}$ has only two nodes and one edge.

PLATEAU-SCC

INSTANCE: A time bound t and a bipartite graph $G = (X, Y, E)$ such that $|X| = |Y|$ and such that it has a perfect matching and such that for any subset $\emptyset \subset X' \subset X$, $|\Gamma'(X')| - |X'| < 0$.
QUESTION: Is there a valid PLT schedule of length $\leq t$?

Lemma 3.34 PLATEAU $\equiv$ PLATEAU-SCC.

Proof: We have already described how to use segregation to decompose a single instance of PLATEAU into at most a linear number of instances of PLATEAU-SCC that be can solved independently. Hence PLATEAU $\preceq_{\mathcal{P}}$ PLATEAU-SCC.

Since the PLATEAU problem is the same as the PLATEAU-SCC problem except that the allowable instances of PLATEAU are a superset of the allowable instance of

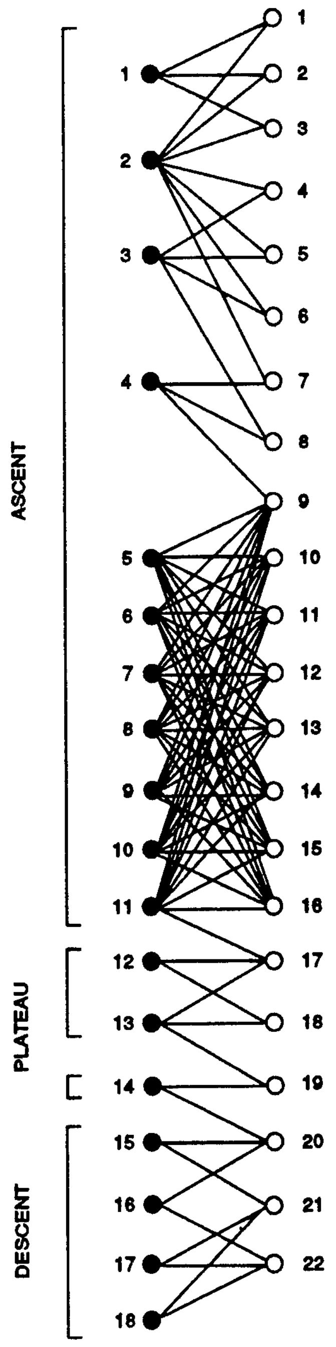


Figure 3.6: APD-decomposition

PLATEAU-SCC, then PLATEAU-SCC $\preceq_{\mathcal{P}}$ PLATEAU. ▮

Lemma 3.35 ASCENT $\equiv$ DESCENT.

Proof: Because of Corollary 2.6 we can start with an instance of ASCENT, let $t' = t+|Y|-|X|$, transpose its adjacency matrix and solve the problem as a DESCENT instance. Similarly, we can start with an instance of DESCENT, let $t' = t+|Y|-|X|$, transpose its adjacency matrix and solve the problem as an ASCENT instance. ▮

Lemma 3.36 ASCENT $\preceq_{\mathcal{P}}$ PLATEAU.

Proof: Given an instance of ASCENT. Add $|Y|-|X|$ new nodes to X. Connect each one of these to each node of Y. Solve this instance of PLATEAU yielding an optimal schedule S. Give the label two to all the original nodes of X and give the label one to the others. By Corollary 3.28, the segregated schedule S' will have $d'_m \geq d_m$. Since S was optimal, $d'_m = d_m$ and hence $\tilde{d}'_{m-1} = \tilde{d}_{m-1}$. Clearly, $\tilde{d}'_{f-1} > \tilde{d}'_{m-1}$, so Lemma 3.30 tells us that $\bar{S}'_f = [x'_{d+1}, \ldots, x'_m]$ is an optimal schedule for the graph induced by $(S'_m \setminus S'_f) \bigcup (T'_m \setminus T'_f)$, which is just our original instance of ASCENT. ▮

Lemma 3.37 PLATEAU $\preceq_{\mathcal{P}}$ ASCENT.

Proof: Given an instance of PLATEAU. Add one new node y_{m+1} to Y. Connect it to every node $x_i \in X$. Any optimal ASCENT solution is clearly an optimal PLATEAU solution. ▮

Corollary 3.38 PLATEAU $\equiv$ ASCENT $\equiv$ DESCENT $\equiv$ PLT. ▮

3.3.6 Well-Ordered Optimum

These segregations have given us a good start on defining what we will call well-ordered optimum schedules. To actually define them, we need to introduce more terminology. All examples use the graph in Figure 3.6.

Definition 3.8 A *nonnegative-profit* subschedule (NPS) is a schedule $[x_1, \ldots, x_k]$, $k \leq m$, such that $n_k \geq 0$ and $n_i \leq 0$ for all $0 \leq i < k$. For example in Figure 3.6, the subschedule $[x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}]$ is an NPS whose chart drops all the way down to -7 before bouncing back up to 0; thus, this is a $(-7, 0)$ NPS. A *minimum-borrowing* nonnegative-profit subschedule (MBNPS) is a NPS whose $\tilde{d}_{k-1}$ is no smaller than that of any other NPS. The charts for the NPSs $[x_4, x_3, x_2]$ and $[x_2, x_1]$ only drop to -2 and there are no NPSs whose chart only drops to -1; thus these are $(-2, +2)$ and $(-2, +1)$ MBNPSs. A *minimal* minimum-borrowing nonnegative-profit subschedule (MMBNPS) is a MBNPS which is minimal in the sense that there is no strict subset of the X nodes of the subschedule that can be re-ordered to yield a nonnegative-profit subschedule with the same or better borrowing then the original subschedule. The $(-2, +2)$ MBNPS $[x_4, x_3, x_2]$ is not minimal, since it contains the $(-2, +1)$ MBNPS $[x_2, x_3]$; the subschedules $[x_2, x_3]$ and $[x_1, x_2]$ are MMBNPSs. Finally, a *minimal NPS with borrowing b* is an NPS that is minimal for its borrowing b, even though it may not be a MBNPS. The subschedule $[x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}]$ is a minimal NPS with borrowing -7.

AMMBNPS

INSTANCE: A time bound t and a bipartite graph $G = (X, Y, E)$ such that $|X| < |Y|$ and such that it has a maximum matching of size $|X|$ and such that every node is reachable from one of the $|Y| - |X|$ unmatched nodes by an alternating path.

FIND: An *arbitrary* MMBNPS.

It turns out that MMBNPSs must be connected:

Lemma 3.39 Let $S = [x_1, \ldots, x_h]$ be a (b, p) MMBNPS for a graph G. The graph induced by the nodes $S \cup \Gamma'(S)$ is a connected graph.

Proof: Assume to the contrary that $S \cup \Gamma'(S)$ is made up of $k > 1$ connected components $C_1, \ldots, C_k$. The schedule S can be viewed as a merge of k separate schedule $S^1, \ldots, S^k$ such that S^i only uses nodes from C_i. We will borrow the notations $u(i, j)$ and n^i_j that were used in the proof of Lemma 3.14.

Choose j' to be the smallest number such that there exists an i with $n^i_{j'} \geq 0$ and $u(i, j') > 0$. For any j there is at most one i such that $n^i_j \neq n^i_{j-1}$, hence the i corresponding to our choice of $j = j'$ is unique, and we will assume without loss of generality that this i is equal to 1.

Since $b \leq n_j$ and $\sum_{i=2}^{k} n_j^i \leq 0$ for $0 \leq j \leq j'$, then $b \leq \sum_{i=1}^{k} n_j^i = n_j^1 + \sum_{i=2}^{k} n_j^i \leq n_j^1$ for $0 \leq j \leq j'$.

What this means is that the subschedule consisting of the first $u(i, j')$ nodes of the schedule S^1 is an NPS with borrowing $\geq b$. Thus S is not minimal and we have our contradiction. ▮

Definition 3.9 We call a schedule a *well-ordered schedule* if it satisfies the four constraints:

1. The schedule is APD-segregated.
2. The ascent is a *greedy* sequence of MMBNPS—that is, we find an arbitrary MMB-NPS, schedule it first, and then iterate on the remaining portion of the ascent.
3. The plateau is segregated into SCCs, each of which is optimally scheduled. The SCCs appear in a valid topological order.
4. If one looks at the transpose of the descent, it is a greedy sequence of MMBNPS.

In defining a well-ordered optimum, we have used a greedy sequence of MMBNPS's. Before we show that all well-ordered schedules are optimum schedules, we need some helping lemmas.

Lemma 3.40 Given any minimal NPS of borrowing b then there is schedule with borrowing $= \min(b, -\text{overhang}(M))$ that begins with the NPS.

Proof: Start with an arbitrary optimum schedule $S = [x_1, \ldots, x_m]$. As a first step, label the X nodes that appear in the minimal NPS with one, label the other X nodes with a two and then segregate in the order [one, two].

Let f be the number of X nodes labelled one. Since the one nodes form a minimal NPS, $n'_i \leq 0$ for $0 \leq i < f$. Also $n'_f = p \geq 0$ where p is the profit of the NPS. Hence $\max_{i=0}^{f} n_i = n'_f$ and Lemma 3.24 tells us that $n'_{\pi(i)} \geq n_i$ for all x_i with label two.

The initial nodes comprising the NPS may not be optimally scheduled, so we do so now. This has no effect on $n'_{\pi(i)}$ for all x_i with label two.

The worst borrowing for the schedule either: (1) occurs during the scheduling of the one nodes, and hence is equal to b; (2) occurs during the scheduling of the two nodes, and hence is equal to $-\text{overhang}(M)$. ∎

Lemma 3.41 Given any MMBNPS. There is optimal schedule of the ascent that begins with the MMBNPS.

Proof: Let b be the borrowing of the MMBNPS. Claim: $b \geq -\text{overhang}(M)$, and hence Lemma 3.40 gives us our result.

To see that $b \geq -\text{overhang}(M)$, start with any schedule S of the ascent. The balance of this schedule must go positive, so the schedule has some non-empty initial NPS. Say that this NPS has a borrowing of b'. We know that $b' \leq b$. Also, $\text{delay}(S) \geq -b' \geq -b$. Furthermore, $\text{overhang}(M) = \text{delay}(S)$ for any optimum schedule S. Hence $\text{overhang}(M) \geq -b$, or equivalently $b \geq -\text{overhang}(M)$. ∎

Corollary 3.42 Greedy use of AMMBNPS is optimal. ∎

Corollary 3.43 ASCENT $\preceq_{\mathcal{P}}$ AMMBNPS. ∎

3.3.7 Fundamental Charts

Two different well-ordered schedules may have different charts. They may even have different sequences of (b,p) pairs for their ASCENTS. Yet there is a certain fundamental essence that can be abstracted from their charts that will be identical.

To find the *fundamental chart* of a chart first break it up into the standard three pieces, the ASCENT, the PLATEAU and the DESCENT. Modify them as follows.

REPLACE: For the ascent, replace each MBNPS with a single (b,p) pair. For each plateau SCC, replace it with a single (b,p) pair. For the descent, replace each MBNPS of the transpose with a single (b,p) pair. Thus since the upper chart in Figure 3.7 is a $[(-2,+1),(-1,+2),(-1,+1),(-7,+1),(-2,0),(-1,0),(-2,-1),(-2,0)]$, then after replacement we get the chart in the lower left corner.

COMBINE: For the ascent, combine adjacent (b,p) pairs as much as possible: If the pair (b_2,p_2) follows the pair (b_1,p_1) and $p_1 - b_1 \geq -b_2$ then we can combine the two pairs to yield (b_1,p_1+p_2). Do the same thing for the transpose of the descent. For the plateau,

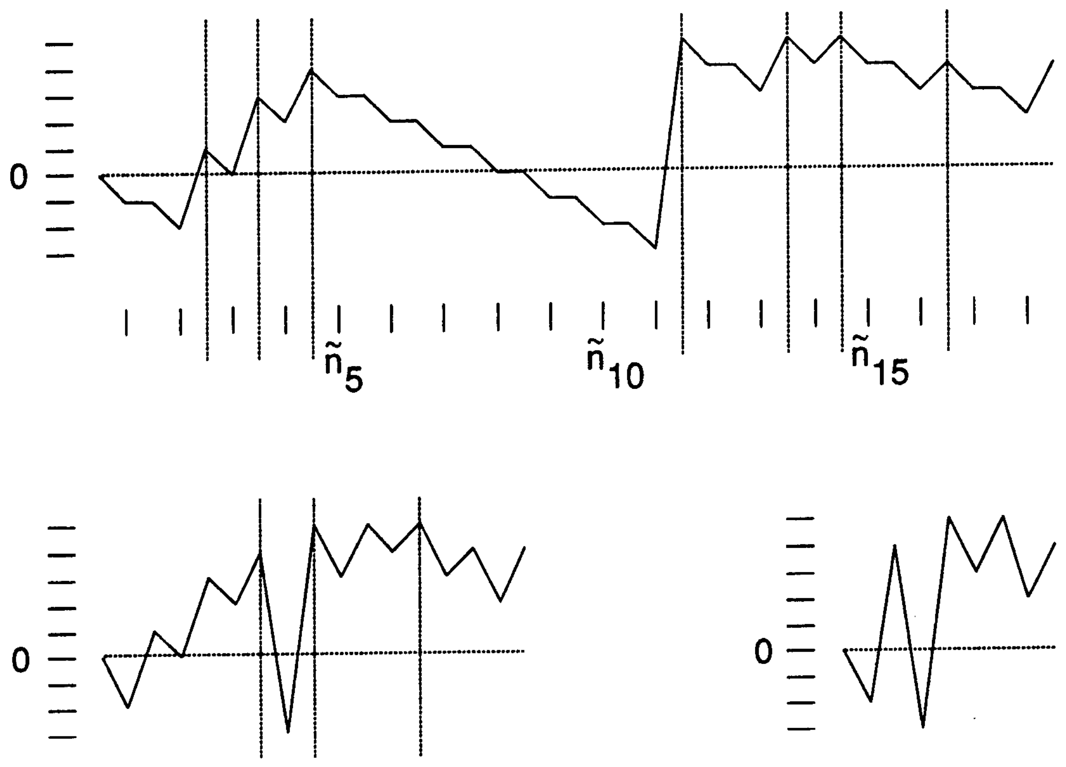


Figure 3.7: Chart and Fundamental Chart.

combine all pairs into a single $(b, 0)$ pair, with b being the minimum b over the all the pairs in the plateau. These leaves us with $[(-2, +4), (-7, +1), (-2, 0), (-2, -1)]$ as shown by the chart in the lower right corner of Figure 3.7. This is the fundamental chart corresponding to the original schedule.

If we restrict our consideration to the ascent, the Fundamental Chart will swing up or down only when the original chart of the schedule hits a new high or a new low.

Lemma 3.44 Any two well-ordered optimum schedules for the same graph will have the same fundamental chart.

Proof: Suppose we have two well-ordered optimum schedules S_1 and S_2. Let a *fragment* be a section of the schedule corresponding to a single (b, p) pair in the fundamental chart. Let F_1 and F_2 be the first fragments in the ascents of S_1 and S_2, respectively. Since the borrowing of a fragment equals the borrowing of its first MMBNPS, the two fragments have the same borrowings $b_1 = b_2$. Let p_1 and p_2 be the profit associated with fragments F_1 and F_2, respectively. If we can just show that they use the exact same sets of X nodes, then it will be clear that $p_1 = p_2$. Also, if they use the exact same sets of X nodes, they must leave the same remaining nodes. Thus we can just iterate this analysis on the remaining nodes, and hence the entire ascents have identical fundamental charts.

Transposition implies that the descents of the schedules also have identical fundamental charts. Lastly, it is obvious that the plateaus have the same fundmental charts and hence the entire fundamental charts are identical.

So all we need to do is show that F_1 and F_2 use the same sets of X nodes. Without loss of generality, assume that $p_1 \leq p_2$. In S_2, label all X nodes used by F_1 with the label one and label all other X nodes with the label two. Segregate S_2 in [one, two] order. Let f equal the number of X nodes labelled with one, that is $f = |F_1|$. Let $g = |F_1 \cup F_2|$. Since $n'_f = p_1$ and $n'_g = p_2$, then $n'_f \leq n'_g$ and hence $S_3 = F_2 \setminus F_1$ is an NPS in $G \setminus (F_1 \cup \Gamma'(F_1))$. Since F_1 is comprised of MMBNPSs, $\max_{i=0}^{f} n'_i = n'_f$. Thus, Lemma 3.24 tells us that $n'_{\pi(i)} \geq n_i$ for all x_i with label two. Hence if $S_3 \neq \emptyset$, the borrowing of S_3 is $\geq b + p_1$ and S_3 must contain an MMBNPS with borrowing $\geq b + p_1$. This is impossible, since such a MMBNPS would have been included in F_1, and hence $S_3 = \emptyset$. Since $S_3 = \emptyset$, $F_2 \subseteq F_1$ and $n'_f = n'_g$. Hence $p_1 = p_2$. Since $p_1 = p_2$, we can now use a symmetric argument to show that $F_1 \subseteq F_2$. ∎

3.3.8 Nearly Well-Ordered Optimum Schedules

It turns out that there are also non-well-ordered schedules for a graph that have the same fundamental chart as all the well-ordered schedules. We will define a *nearly well-ordered* schedule as any schedule, well-ordered or non-well-ordered, that has the same fundamental chart as all the well-ordered schedules.

We will show that the following two problems are equivalent under polynomial-time reductions.

Find an arbitrary optimum schedule (FIND-ARB-OPT)

INSTANCE: A bipartite constraint graph.

FIND: An arbitrary optimum schedule.

Find a nearly well-ordered optimum schedule (FIND-NEAR-WELL-OPT)

INSTANCE: A bipartite constraint graph.

FIND: A nearly well-ordered optimum schedule.

Since the set of nearly well-ordered optimum schedules are a subset of the arbitrary optimum schedules:

Lemma 3.45 FIND-ARB-OPT $\preceq_{\mathcal{P}}$ FIND-NEAR-WELL-OPT. ▌

To see the other direction, let's first examine the problem:

Determine the fundamental chart (FUND-CHART)

INSTANCE: A bipartite constraint graph.

FIND: The fundamental chart.

Lemma 3.46 FUND-CHART $\preceq_{\mathcal{P}}$ FIND-ARB-OPT.

Proof: We are given a graph and we want to determine its fundamental chart.

First we will find the fundamental chart of the `ascent`. Do the following with k running from 1 to $n+1$. Add a new component $K_{k,n+1}$ to the original graph, with the smaller side of the new complete graph in X. Find an arbitrary optimum solution to the new graph; that is, call a polynomial time FIND-ARB-OPT routine. Plot the resulting overhangs as a function of k; call this plot the *scan* of the graph. Figure 3.8 shows the scan for the `ascent` of the graph in Figure 3.6.

It turns out that we can read the fundamental chart right off the scan. Look at the horizontal sections. For example, look at the horizontal section from $k = 2$ to $k = 6$. This means that when adding $K_{3,n+1}$, $K_{4,n+1}$, $K_{5,n+1}$ or $K_{6,n+1}$, there must be a NPS made from nodes in the original graph that preceded the $K_{k,n+1}$ components in the schedule. This NPS must have had a borrowing of 2 and a profit of 4. This generalizes. If there is a horizontal section from k_1 to k_2, then the fundamental chart has a $(b,p) = (-k_1, k_2 - k_1)$ component. Thus the fundamental chart for the example in Figure 3.8 is $[(-2,4)(-7,1)]$.

This carries over to finding the fundamental chart for the transpose of the `descent`. For the `plateau`, there will be no horizontal sections in the scan. This can be fixed by adding a new Y node and connecting it to all the original X nodes. When we find that this modified graph is of type $(b,1)$ for some b, then we know that the original `plateau` is of type $(b,0)$.
▌

Let the first horizontal section be from k_1 to k_2. Let's look at the schedule that was generated when we added the $K_{k_2,n+1}$ piece and called the FIND-ARB-OPT routine. Let's

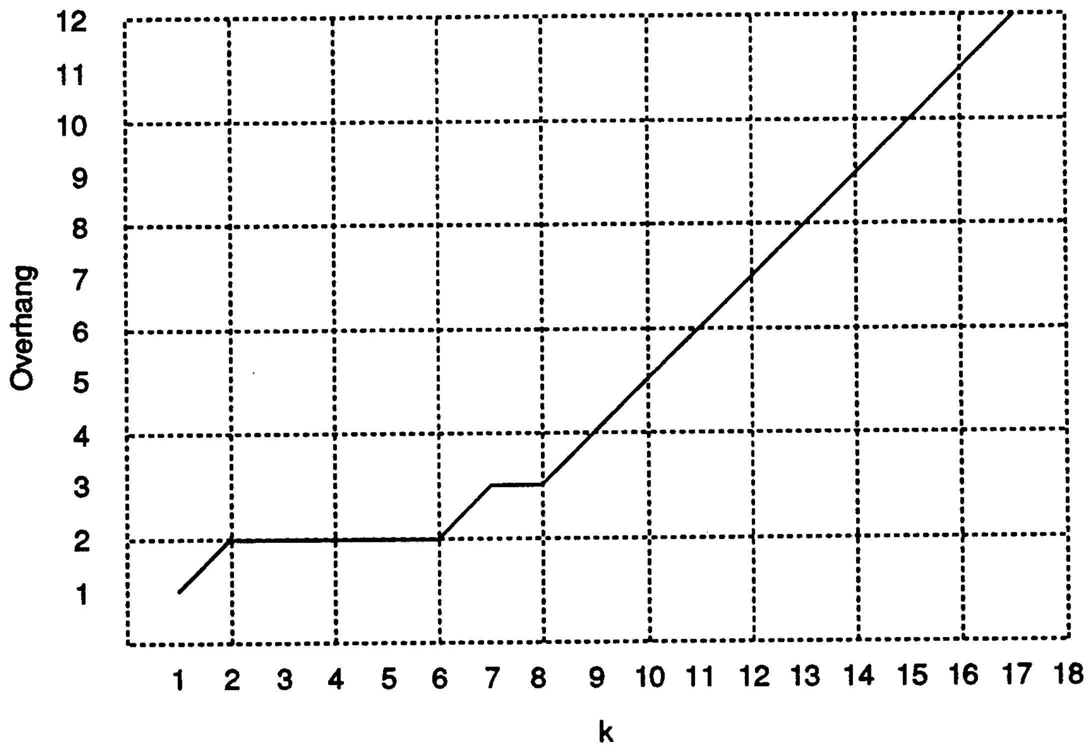


Figure 3.8: Scan of a graph.

focus on the part of the schedule just up to when the $n+1$ Y nodes of the $K_{k_2,n+1}$ are released. Label the $K_{k_2,n+1}$ piece with two and label all the X nodes that occur before the last X node of the $K_{k_2,n+1}$ piece with one. Segregate in [one, two] order. The result is a $(-k_1, k_2 - k_1)$ piece followed by a $(-k_2, n+1-k_2)$ piece. How do we know that the first piece is a $(-k_1, k_2 - k_1)$ piece? The piece can't have borrowing $> -k_1$, because then the first horizontal section of the scan would have started earlier. The piece can't have borrowing $< -k_1$, because the $(-k_2, n+1-k_2)$ piece doesn't contribute a positive amount to the chart value until the end, and the overall borrowing is only k_1. If the piece had profit $> k_2 - k_1$, then the first horizontal section of the scan would continue past k_2. Lastly, if the profit were $< k_2 - k_1$, then the $(-k_2, n+1-k_2)$ piece would cause a borrowing $< -k_1$.

The $(-k_1, k_2 - k_1)$ piece gives us the first fragment of a nearly well-ordered optimum schedule. Then we can simply delete all those nodes from the graph and repeat. Hence:

Corollary 3.47 FIND-NEAR-WELL-OPT $\preceq_{\mathcal{P}}$ FIND-ARB-OPT. ▌

3.3.9 Merge

With what we now know about well-ordered schedules we can return to the question that started this section. Is there a simple method of combining independent optimum schedules of disconnected subgraphs into an optimum schedule for the whole graph? We will show that well-ordered schedules are easier to merge than arbitrary optimum schedules. We will show this by considering the following three problems.

MERGE-ARB

INSTANCE: Two separate constraint graphs and *arbitrary* optimal schedules for them.

FIND: An arbitrary optimal schedule for the combined graph.

MERGE-NEAR

INSTANCE: Two separate constraint graphs and *nearly well-ordered* optimal schedules for them.

FIND: A nearly well-ordered optimal schedule for the combined graph.

MERGE-WELL

INSTANCE: Two separate constraint graphs and *well-ordered* optimal schedules for them.

FIND: A well-ordered optimal schedule for the combined graph.

We will show that MERGE-WELL and MERGE-NEAR are quite easy. However, MERGE-ARB is as hard as the problem of finding a nearly well-ordered schedule and hence can only be solved easily if PLT can.

Lemma 3.48 FIND-NEAR-WELL-OPT $\preceq_{\mathcal{P}}$ MERGE-ARB

Proof: The proof runs exactly like Lemma 3.46 and Corollary 3.47, except that we replace calls to FIND-ARB-OPT with calls to MERGE-ARB. Everything else goes through the same way. ▌

Corollary 3.49 MERGE-ARB $\equiv$ PLT. ▌

What about MERGE-WELL? Once again, the problem can be partitioned into **ascent**, **plateau** and **descent** pieces. The **ascent** of the merged schedules will be a merge of the two **ascents**. The **plateau** of the merged schedules will be a merge of the two **plateaus**. To merge the two **descents** we transpose them into **ascents**, merge them and then transpose them back.

How do we merge two well-ordered **ascents**? Corollary 3.42 tells us that greedy use of AMMBNPS is optimal. Fortunately, Lemma 3.39 tells us that a MMBNPS must be a connected piece and hence any MMBNPS of the combined graph is an MMBNPS of one of the **ascents**. Look at the first MMBNPS of each of the two schedules. They are easy to find since they are already right at the beginning of the schedules. They are of some types, say (b_1, p_1) and (b_2, p_2). If $b_1 \geq b_2$ then by greedy use of AMMBNPS we can choose the (b_1, p_1) piece as the first MMBNPS of our new merged schedule; otherwise choose the (b_2, p_2) piece as the first MMBNPS. After removing the chosen MMBNPS, iterate on the remaining portion of the graph.

The optimal merge of two well-ordered **plateaus** is even simpler. One only needs to concatenate the two schedules without interweaving. This is because all MMBNPSs have profit $= 0$ and each $(b, 0)$ MMBNPS will dip to the same chart level relative to the peak balance regardless of the order of the MMBNPSs.

For example, given the following two well-ordered schedules:

$$[(-2,0),(-1,2),(-2,0),(-2,0),(-5,-1),(-5,-3)]$$
$$[(-1,1),(-3,1),(-1,0),(-3,0),(-4,-1),(-4,-3)].$$

The (b,p) pairs up to and including the last one with a positive p comprise the ASCENT. The (b,p) pairs starting from and including the first one with a negative p comprise the DESCENT. The $(b,0)$ pairs left in the middle comprise the PLATEAU. So, one possible well-ordered merge is:

$$\begin{aligned}&[(-1,1),(-2,0),(-1,2),(-3,1),(-2,0),(-2,0),(-1,0),(-3,0),\\&\quad(-5,-1),(-4,-1),(-5,-3),(-4,-3)]\end{aligned}$$

What about MERGE-NEAR? It turns out that we can use the same algorithm used for MERGE-WELL.

3.4 Weighted Chains

We can generalize "charts" to handle weighted nodes directly. The reduction in Section 2.3 was mainly to show equivalence. It is unlikely that someone faced with the weighted problem would wish to convert it to the unit-weight version before actually solving it.

We generalize the definitions of n_i and $\bar{n}_i$ in Section 3.2 to:

$$\begin{aligned}n_i &= (\sum_{y\in T_i} w(y)) - (\sum_{x\in S_i} w(x))\\ \bar{n}_i &= (\sum_{y\in T_i} w(y)) - (\sum_{x\in S_{i+1}} w(x))\end{aligned}$$

So the graph in Figure 2.15 with the schedule $S_X = [x_1, x_5, x_2, x_4, x_3]$ will have the chart shown in Figure 3.9.

The algorithm for merging schedules based on their charts remains the same, so we can merge well-ordered schedules of weighted graphs in polynomial time. This serves as the basis for a polynomial time algorithm to schedule weighted chains.

A chain is a string of nodes as shown in Figure 3.10.

The algorithm to schedule chains is based on dynamic programming. We will compute the optimum schedule for every connected subgraph of the chain. These correspond to intervals of the chain. There are $O((m+n)^2)$ such intervals.

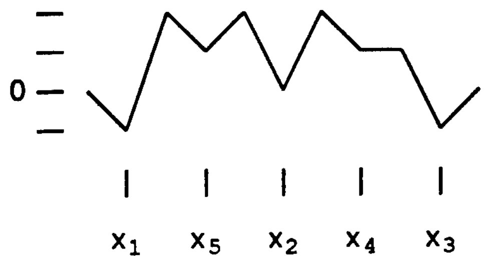


Figure 3.9: Chart for a weighted graph.

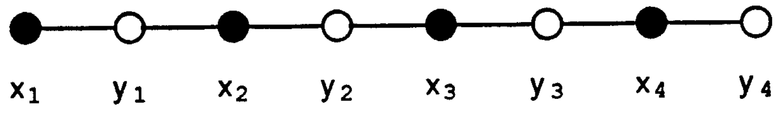


Figure 3.10: A chain.

We compute these schedules in a particular order and by a particular method: We first compute the schedule for all connected subgraphs of size 1, then size 2, then size 3, and so on. For a particular graph with k X nodes, we compute its schedule as follows: We will compute k different schedules one of which is guaranteed to be a nearly well-ordered optimum; then, we pick out the nearly well-ordered optimum from this set.

The k different schedules correspond to the fact that the schedule must start with some one of the X nodes. We can easily compute the best schedule given that a particular node x is first: After removing x, what is left is one or two smaller connected pieces. We have already computed nearly well-ordered optimums for these pieces. By merging them, we get a nearly well-ordered optimum for the whole graph piece minus the node x.

By the suffix law of Corollary 3.51, if some nearly well-optimum for the piece begins with x, then the suffix of the optimum schedule can be any well-ordered optimum schedule of the remaining graph. Thus by concatenating the merge of the pieces to x, we will get a well-ordered optimum for the piece, assuming there is a well-ordered optimum beginning with x. That is:

Lemma 3.50 Given that there is a well-ordered optimum schedule starting with the node x. Let $S = (S_X, S_Y)$ be an arbitrary well-ordered optimum schedule for $G \setminus x$. Then $S = ([x, S_X], S_Y)$ is a well-ordered optimum schedule for G. ▌

Corollary 3.51 Given that there is a nearly well-ordered optimum schedule starting with the node x. Let $S = (S_X, S_Y)$ be an arbitrary nearly well-ordered optimum schedule for

$G \setminus x$. Then $S = ([x, S_X], S_Y)$ is a nearly well-ordered optimum schedule for G. ∎

Since we construct one such schedule for each x, and some x must be first in the nearly well-ordered optimum, at least one of the k schedules is our nearly well-ordered optimum schedule. All we need to do is pick it out and we are done.

So this brings up the new sub-problem:

Select a nearly well-ordered optimum (SELECT-NWO-OPT)

INSTANCE: A set of schedules that is guaranteed to contain a nearly well-ordered optimum schedule.

FIND: A member of the set that is a nearly well-ordered optimum schedule..

Since we just have to pick an optimum out from a set of schedules which is *guaranteed* to contain one, we only need to be able to make comparative evaluations: "this schedule is *more nearly well-ordered* than this other schedule."

So how do we pick out a nearly well-ordered optimum schedule from amidst a line-up of fakers? For example, say that our set of schedules is:

$$\begin{aligned}
S_1 &= [(-2,2),(-3,2),(-2,0),(-1,0),(-3,-2)] \\
S_2 &= [(-2,2),(-3,2),(-3,0),(-1,0),(-3,-2)] \\
S_3 &= [(-3,2),(-2,2),(-2,0),(-1,0),(-3,-2)] \\
S_4 &= [(-2,2),(-3,1),(-2,0),(-1,0),(-3,-1)] \\
S_5 &= [(-2,2),(-3,2),(-1,0),(-2,0),(-3,-2)]
\end{aligned}$$

Divide the schedules into `ascent`, `plateau` and `descent`. We can discard a schedule if its `plateau` dips further negative relative to the peak balance than some other schedule. So in the example, we can discard S_2.

Now we just have to check the `ascents` (and the transposed `descents`). Assume that we have two alternate schedules S and S'. Assuming they are nearly well-ordered we can easily determine their fundamental charts and break them up into their fragments: $S = [F_1, F_2, F_3, \ldots]$ and $S' = [F'_1, F'_2, F'_3, \ldots]$. Corresponding to each fragment is a (b_i, p_i) or (b'_i, p'_i) pair. In our example, we now have:

$$S_1 = [(-2,4),(-2,0),(-3,-2)]$$

$$\begin{aligned} S_3 &= [(-3,4),(-2,0),(-3,-2)] \\ S_4 &= [(-2,3),(-2,0),(-3,-1)] \\ S_5 &= [(-2,4),(-2,0),(-3,-2)] \end{aligned}$$

For i increasing from 1 perform the following comparison procedure: If $b_i > b'_i$ then we can discard S' and stop. If $b'_i > b_i$ then we can discard S and stop. So $b_i = b'_i$. If $p_i > p'_i$ then we can discard S' and stop. If $p'_i > p_i$ then we can discard S and stop.

Lemma 3.52 If both schedules survive the comparison procedure, and either one is a nearly well-ordered schedule, then the other is too.

Proof: Easy. Both survive if and only if they have the same fundamental chart. ▮

In our example, only schedules S_1 and S_5 survive.

Since all nearly well-ordered schedules will survive, since there exists at least one well-ordered schedule, and since the surviving schedules are either all nearly well-ordered or all not nearly well-ordered, then all the surviving schedules are nearly well-ordered. Thus we can arbitrarily pick any one of them.

Chapter 4

One, Two, Three, Approximation

4.1 Nodes with degree = 1

There is a simple rule for any Y node with degree $= 1$, that is, for any node y with $\Gamma(y) = \{x\}$. We can schedule the node x immediately. We can do this because scheduling x has a borrowing of 1 and a non-negative profit; it is $(-1, 0)$ and hence it is an MMBNPS and we can schedule it next by Corollary 3.42. If this creates more Y nodes of degree one, then we can repeatedly apply this rule. By transposition, we can apply a similar rule when we have an X node of degree 1.

4.1.1 Scheduling trees in polynomial time

A consequence of this is that we can handle graphs whose underlying constraint graph, ignoring the directions on the precedence arcs, is a tree. We can schedule a tree as follows: As long as there is a leaf node that is a Y node, schedule its X-neighbor next. This may completely schedule the tree, or it may leave us a tree in which all the leaves are X nodes. In the latter case, we will have a partial schedule (S_X, S_Y). So for the example in Figure 4.1, we may get a schedule $([x_1, x_2, x_6][\emptyset, y_1, y_2, y_3, y_6])$. If our remaining tree has only X nodes as leaves, then we treat the remainder as a separate problem by transposing it and recursing. This will return us a schedule (S'_Y, S'_X) for the transposed tree. So for the example in Figure 4.1, we may get $([y_5, y_4], [\emptyset, x_5, x_4, x_3])$. Easily construct a schedule for the overall graph as $([S_X, \text{rev}(S'_X)], [S_Y, \text{rev}(S'_Y)])$, so in our example, we get: $([x_1, x_2, x_6, x_3, x_4, x_5, \emptyset], [\emptyset, y_1, y_2, y_3, y_6, y_4, y_5])$.

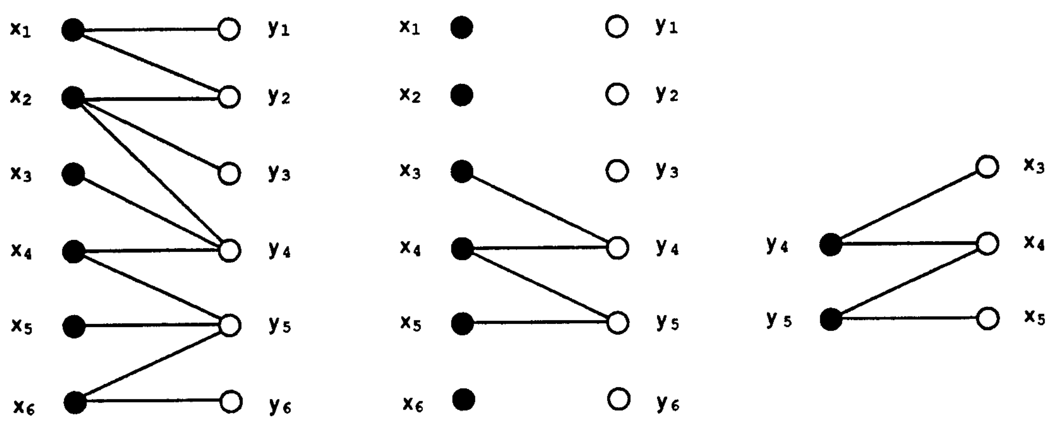


Figure 4.1: Scheduling a tree in polynomial time

4.2 Nodes with degree = 2

The degree = 1 rule that we just used suggests a degree = 2 rule. Say that we have a node y with degree 2 and $\Gamma(y) = \{x_1, x_2\}$. Once we schedule one of these two nodes — without loss of generality say x_1 — then y becomes a degree 1 node. Once x_1 has been scheduled and removed from the graph, scheduling x_2 next becomes a $(-1, 0)$ MMBNPS and hence is optimal by Corollary 3.42.

So the general rule is:

Lemma 4.1 Given a schedule $S_X = [x_1, \ldots, x_n]$ and a node y such that $\Gamma(y) = \{x_i, x_j\}$, $i < j$, transforming the schedule into $S_X = [x_1, \ldots, x_i, x_j, x_{i+1}, \ldots, x_{j-1}, x_{j+1}, \ldots x_m]$ is delay non-increasing. ▮

4.2.1 Collapsing Transformation

The lemma leads us to consider a transformation where we collapse together the two X nodes and discard the Y node as shown in Figure 4.2. Define the Collapsing Transformation as:

- Replace a degree 2 node y and its two neighbors x_1 and x_2 with a single node x' such that $\Gamma(x') = (\Gamma(x_1) \cup \Gamma(x_2)) \setminus y$.

Under the right conditions, this transformation will not affect the overhang of the graph. Under the wrong conditions, such as in the two graphs in Figure 4.3, the Collapsing Transformation does affect the overhang.

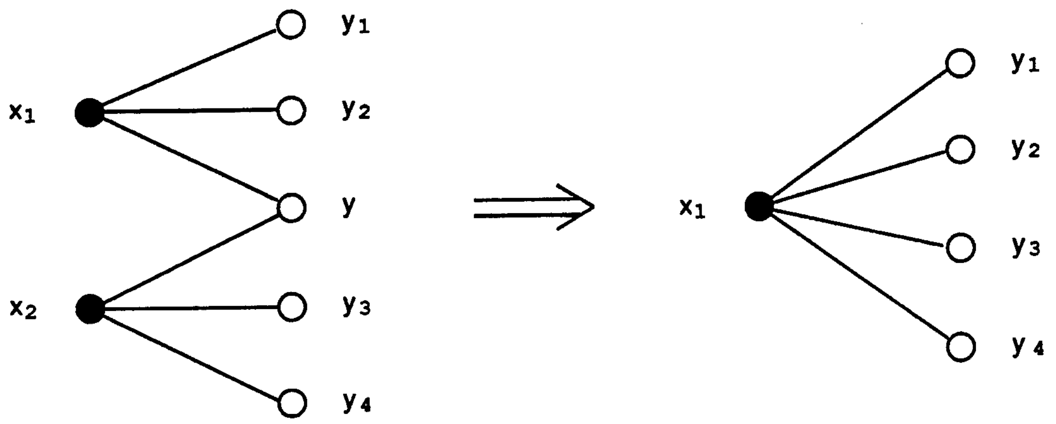


Figure 4.2: Collapsing Transformation applied to y

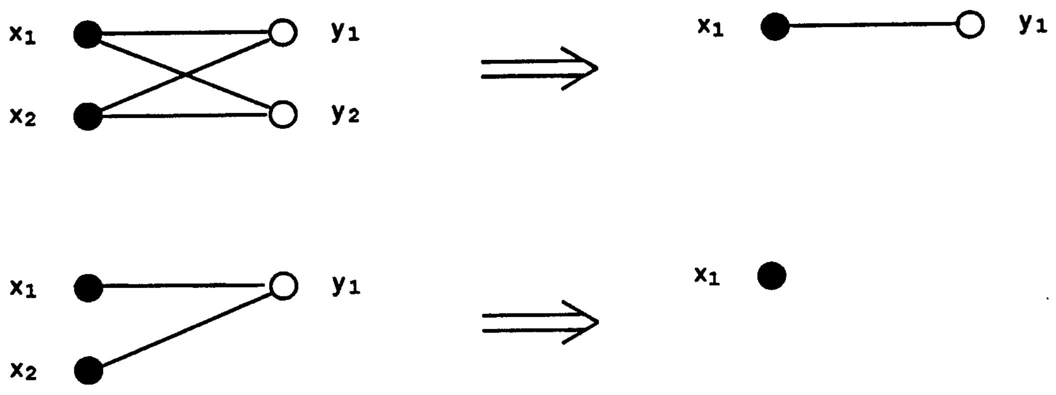


Figure 4.3: Bad Collapsing Transformations

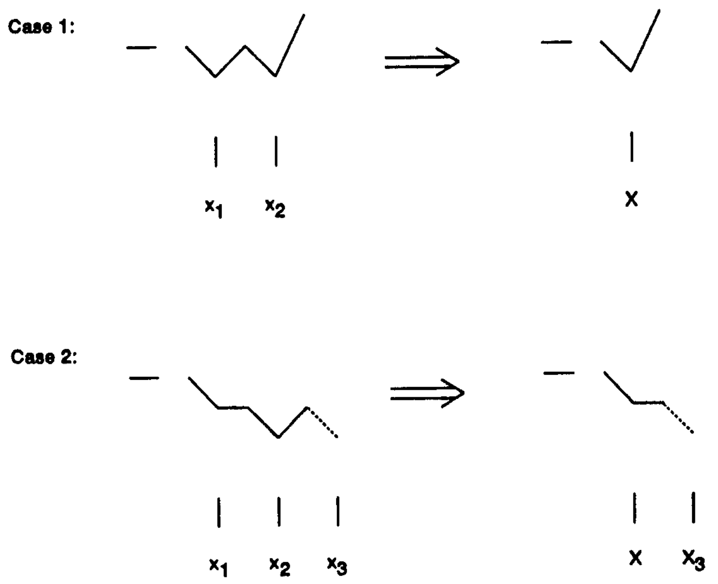


Figure 4.4: Effect of constrained Collapsing Transformations

The following lemma spells out the proper constraints to ensure that our transformation doesn't change the overhang of the graph.

Lemma 4.2 Under the constraints that $\Gamma(x_1) \cap \Gamma(x_2) = \{y\}$ and $|\Gamma(x_1) \cup \Gamma(x_2)| \geq 2$, the Collapsing Transformation leaves the overhang of the graph unchanged.

Proof: By Lemma 4.1 there is an optimum schedule of the original graph that has x_1 and x_2 consecutive. Create an ordering for the new post-collapsing graph that schedules x in the place of where x_1 and x_2 were. We claim that this ordering has the same delay as the optimal schedule of the original graph. To see this, we just need to look at the two cases shown in Figure 4.4:

- Case 1: Due to their position in the schedule, either x_1 or x_2 releases at least one job other than y, say y_2. Since $\Gamma(x_1) \cap \Gamma(x_2) = \{y\}$, y_2 can be released by executing the appropriate one of the two x jobs first. Without loss of generality assume that it is x_1. In this case the original schedule could have scheduled x_1 before x_2 and thus the minimum borrowing of the chart is clearly not affected by the transformation.
- Case 2: Due to their position in the schedule, neither x_1 nor x_2 releases a job other than y. Since $|\Gamma(x_1) \cup \Gamma(x_2)| \geq 2$, x_2 and x_1 cannot be the last X jobs scheduled; they

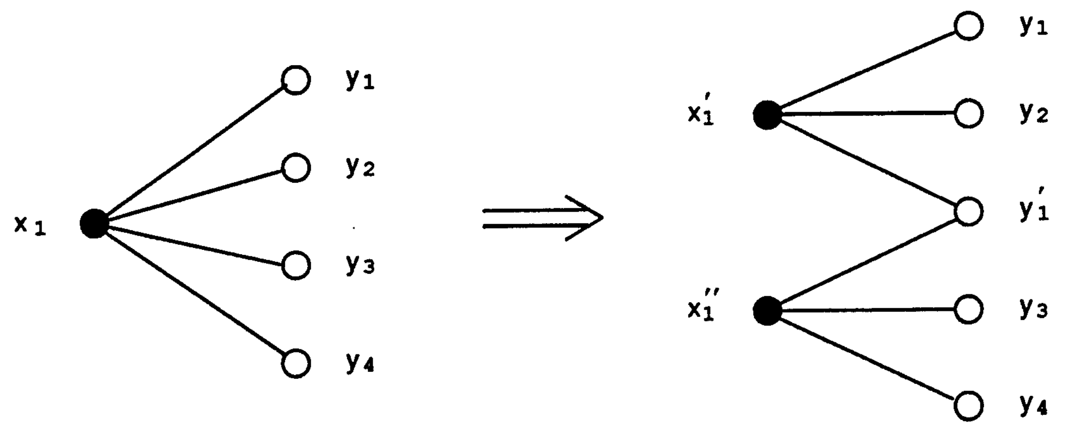


Figure 4.5: Expanding nodes into degree 3 nodes

must be followed by some other job, say x_3. Again, the chart is clearly not affected by the transformation.

This shows that the collapsed version has a delay no better than the original. To see that it is no worse note that we could just replace x by x_1 and x_2 and yield a schedule for the original graph that is no worse than the one that we already know is optimal. ∎

So, naturally, repeating this constrained Collapsing Transformation as many times as we want will not change the overhang of the graph.

4.3 Nodes with degree ≤ 3

There is no apparent way to extend this to collapsing degree 3 nodes. Instead what we will do in this section is look at a reverse of the Collapse Transformation from the previous section, and we will simultaneously explore the question: how hard is it to optimally schedule graphs whose nodes all have degree ≤ 3?

4.3.1 Expanding Transformation

What does the reverse of the Collapse Transformation look like? An example is shown in Figure 4.5. Say in a graph G we have a node x_1 of degree 4 with $\Gamma(x_1) = \{y_1, y_2, y_3, y_4\}$. To yield the graph G', we remove the node x_1 and replace it with the two nodes x_1' and x_1'', and we create a new node y_1'. We add edges so that $\Gamma(x_1') = \{y_1, y_2, y_1'\}$ and $\Gamma(x_1'') = \{y_3, y_4, y_1'\}$.

Since $\Gamma(y_1') = \{x_1', x_1''\}$ and $\Gamma(x_1') \cap \Gamma(x_1'') = \{y_1'\}$ and $|\Gamma(x_1') \cup \Gamma(x_1'')| \geq 2$, then we could clearly apply the Collapsing Transformation to y' and G' yielding the original graph G. Thus the two graphs G and G' have the same overhang by Lemma 4.2. The only constraints for the Expanding Transformation are that $|\Gamma(x_1)| \geq 1$ and that we ensure that $\Gamma(x_1') \cap \Gamma(x_1'') = \{y'\}$ and $\Gamma(x_1') \cup \Gamma(x_1'') = \Gamma(x) \cup \{y'\}$.

Lemma 4.3 Under the constraints that $\Gamma(x_1') \cap \Gamma(x_1'') = \{y'\}$ and $\Gamma(x_1') \cup \Gamma(x_1'') = \Gamma(x) \cup \{y'\}$, the Expanding Transformation leaves the overhang of the graph unchanged.

Proof: By Lemma 4.2, if we start with G' and collapse y', yielding G, we will not have changed the overhang of the graph. Thus G and G' have the same overhang. ▮

4.3.2 Three ones per row

We have seen that it is easy to solve the PLT problem when we have two 1s per row. What about threes 1s per row? four 1s per row? In this section we will look at the special case of having at most three 1s per row. We will show that this special case is as hard as the original problem. Even the restriction "at most three 1s per row and at most three 1s per column" is as hard as the original problem.

Lemma 4.4 Any $n \times n$ matrix M with $\leq k$ 1s per column, can be reduced to an equivalent $2n \times 2n$ matrix M' with $\leq (\lceil k/2 \rceil + 1)$ 1s per column. By being "equivalent" we mean that they have the same overhang.

Proof: Choose any X node (that is, a column) with degree $> \lceil k/2 \rceil + 1$. Apply the Expanding Transformation to it, creating new nodes x' and x'' such that $|\Gamma(x')| \leq \lceil k/2 \rceil + 1$ and $|\Gamma(x'')| \leq \lceil k/2 \rceil + 1$. We can always do this, since $|\Gamma(x')| + |\Gamma(x'')| = |\Gamma(x)| + 1 \leq k + 1$.

Each transformation adds one row and one column to the matrix and preserves the overhang. There need be at most n transformations. ▮

Corollary 4.5 Any $n \times n$ matrix with $\leq n$ 1s per column, can be reduced to an equivalent $n^2 \times n^2$ matrix with ≤ 3 1s per column. ▮

Corollary 4.6 Any $n \times n$ matrix with $\leq n$ 1s per row and $\leq n$ 1s per column can be reduced to an equivalent $n^3 \times n^3$ matrix with ≤ 3 1s per row and ≤ 3 1s per column. ▮

Figure 4.6 shows an example where we go from 4 1s per row to 3 1s per row.

$$
\begin{array}{ccccc}
 & 0 & 1 & 2 & 3 \\
0 & \times & \times & \cdot & \cdot \\
1 & \times & \times & \cdot & \cdot \\
2 & \times & \times & \times & \times \\
3 & \times & \times & \times & \times
\end{array}
$$

$$
\begin{array}{ccccccccc}
 & 0 & 0' & 1 & 1' & 2 & 2' & 3 & 3' \\
0' & \times & \times & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
0'' & \cdot & \times & \times & \cdot & \cdot & \cdot & \cdot & \cdot \\
1' & \times & \cdot & \cdot & \times & \cdot & \cdot & \cdot & \cdot \\
1'' & \cdot & \cdot & \times & \times & \cdot & \cdot & \cdot & \cdot \\
2' & \times & \cdot & \times & \cdot & \cdot & \times & \cdot & \cdot \\
2'' & \cdot & \cdot & \cdot & \cdot & \times & \times & \times & \cdot \\
3' & \times & \cdot & \times & \cdot & \cdot & \cdot & \cdot & \times \\
3'' & \cdot & \cdot & \cdot & \cdot & \times & \cdot & \times & \times
\end{array}
$$

Figure 4.6: From 4$\times$ to 3$\times$ per row and column

4.4 Polynomial Solutions to Overhang-Limited PLATEAUs

In this section we describe an algorithm that can solve PLATEAU-SCC with a fixed bound of $t = k$ and consequently AMMBNPS with borrowing $\leq k$ in time $O(n^{k+3})$. The work was inspired by work on the related problem known as Bandwidth Minimization [GGJK78] [Sax80] [GS84]. One interesting consequence of this result is that we can solve the PLT$_3$ problem in polynomial time for any graph whose nodes are all of degree *exactly* equal to three.

Since we will deal with PLATEAU-SCC, the chart will never go above 0. Since we are only interested in schedules with a bound of $t = k$, the chart can never go below k. It is this tight constraint on chart values that allows us to have a polynomial time algorithm.

Note that we are only interested in finding *some* schedule with delay $\leq k$, not necessarily an optimum one. An optimum one can be found by running this algorithm multiple times with different values for k.

Here is how it works:

At phase i, the set of active schedules can be depicted as a rooted tree of depth i. The edges of the search tree will have the names of X nodes attached to them. Every "active"

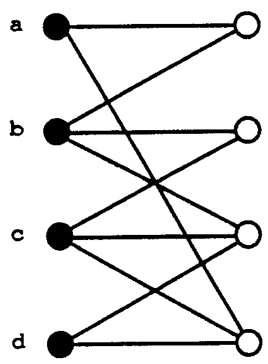


Figure 4.7: The Input Graph

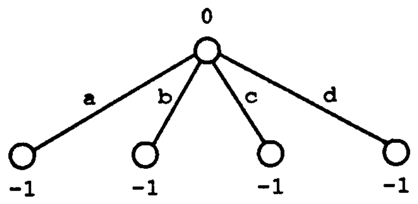


Figure 4.8: Phase 1

leaf of the graph represents an active alternative sub-schedule for the graph. That sub-schedule is simply the sequence of edge labels along the path from the root of the tree to the leaf. The nodes will have numbers, representing chart levels, attached to them.

For example, let the graph in Figure 4.7 be our input. The search tree is expanded breadth first. So the search tree after phase 1 is shown in Figure 4.8. This means that our four active sub-schedules are $[A]$, $[B]$, $[C]$ and $[D]$. Each has a resulting balance of -1.

At the next phase, each active leaf is expanded in all possible ways to yield sub-schedules that are one node longer. Thus at phase 2, we have the tree in Figure 4.9. Fortunately, at this point we can prune the tree. Without any pruning, the algorithm would require $O(n!)$ time.

Here is how the pruning works: Since the leaf corresponding to $S = [A, B]$ rises back up to a chart level that was achieved by a prefix of it, namely $S' = [A]$, then $S \setminus S'$ is a nonnegative profit sub-schedule of $G \setminus S'$. Indeed, it is a ***shortest*** nonnegative profit sub-schedule of $G \setminus S'$ and hence a minimal NPS. Let b be the lowest chart level reached by S and b' be the final chart level reached by S'. Lemma 3.40 tells us that there is some schedule of $G \setminus S'$ that starts with $S \setminus S'$ and has borrowing of $\min(b - b', -\text{overhang}(G \setminus S'))$. Thus, except for the path from S' to S in the search tree, we can prune all other descendants of

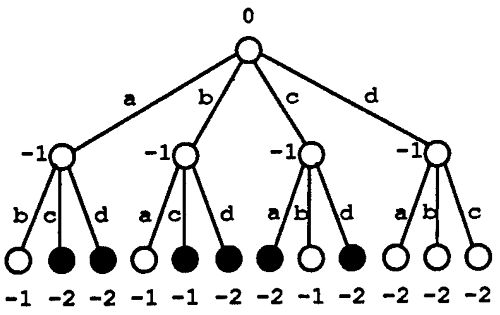


Figure 4.9: Phase 2

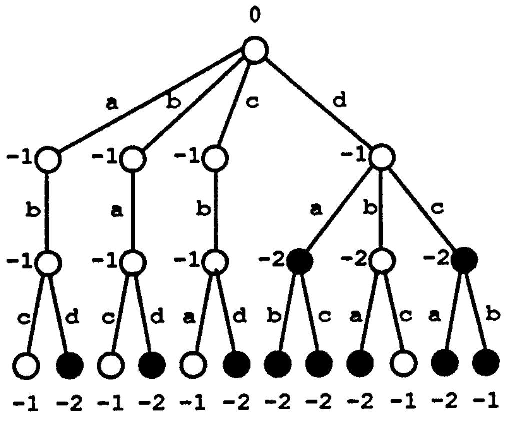


Figure 4.10: Phase 3

the node corresponding to S'. This is why the nodes corresponding to $[A, C]$ and $[A, D]$ are filled in in black. Note that Lemma 3.40 tells us that the choice between saving $[B, A]$ and $[B, C]$ can be made arbitrarily.

The next phase is shown in Figure 4.10. We have again expanded all active leaves. This time we have a larger example of pruning. $S = [D, B, C]$ climbs back up to the chart level achieved by $S' = [D]$. This allows us to prune 5 leaves and 2 interior nodes.

The algorithm continues in this manner until completion (only one more phase for our example). Because of the pruning described above, if a leaf node is labelled with borrowing b, then only $|b|$ of its ancestors can currently have multiple children. Let's add one more pruning rule: If a leaf is labeled with $-k$, then mark that leaf as inactive. We can do this because extending the sub-schedule any further will violate the given bound on borrowing.

The result of all this is that there are never more than n^{k-1} active leaves at the end of any phase. This implies that no more than n^{k+1} nodes are ever generated. Allowing $O(n^2)$ time to generate each node, our overall time is $O(n^{k+3})$.

4.4.1 3-Regular Graphs

As we mentioned at the beginning of the section, a consequence of this result is that if the nodes of a graph are all of degree *exactly* equal to three, then we can solve the PLT_3 problem in polynomial time.

This is because of the following lemma:

Lemma 4.7 Each connected component of a k-regular graph is a single PLATEAU-SCC problem.

Proof: There is a perfect matching of the graph, and hence every component is part of the plateau. but we have made an even stronger claim: that every connected component is a *single* PLATEAU-SCC piece. To see this, assume without loss of generality that we have two PLATEAU-SCCs $G_1 = \{X_1, Y_1\}$ and $G_2 = \{X_2, Y_2\}$ that are connected by an edge e_1 from X_1 to Y_2. Since G_1 and G_2 are PLATEAU-SCC pieces, $|X_1| = |Y_1|$ and $|X_2| = |Y_2|$. There are $k \cdot |Y_1|$ edges from nodes in Y_1. Because of the edge from X_1 to Y_2, there can be at most $k \cdot |X_1| - 1 = k \cdot |Y_1| - 1$ edges between Y_1 and X_1. Thus there must be an edge e_2 from Y_1 to X_2. Since neither e_1 nor e_2 is a matching edge, G_1 and G_2 are strongly connected together. ∎

At first it might seem like we have just solved our overall problem. We haven't. The deficiency isn't that our algorithm only works for 3-regular graphs — we only used the 3-regularity of the graphs to show that we had a PLATEAU-SCC problem. The algorithm will solve any PLATEAU problem with $t = 3$. We can also use it to find MMBNPS with borrowing $= 3$.

The shortcoming is this: while being able to solve PLATEAUs and MMBNPS implies that we can solve PLT, it is not true that solving PLATEAU with $t = 3$ and MMBNPS with borrowing $= 3$ implies that we can solve PLT_3. Indeed, solving a PLT_3 problem may involve solving PLATEAU problems with arbitrarily large t.

4.5 Approximations

We have seen that given a PLATEAU-SCC piece, we can find an optimal schedule in polynomial time, provided that the delay of the optimal schedule is below some fixed k of our choosing. If the actual delay is large, this is not useful. In this case, we would still like to be able to get an approximation to the delay, and to get a schedule that is approximately optimum.

Consider the simplest approximation algorithm: While $|Y| > 0$, choose the node $y \in Y$ of minimum degree and schedule the nodes $\Gamma(y)$. If there is a tie for minimum degree, pick randomly.

What can we say about this approximation algorithm? Let's concentrate on graphs with degree ≤ 3. Also let's assume we have a single PLATEAU piece with $m = m_X = m_Y$.

The first thing to note is that: after our algorithm picks the first Y node and schedules $\Gamma(y)$, the graph will have $|X| < |Y|$. This will hold true until all the Y nodes are exhausted. What does this tell us?

Lemma 4.8 If we have a graph with X nodes of degree $\leq c$, then some Y node must have degree $\leq c \cdot |X|/|Y|$.

Proof: This is a simple pigeon hole argument. There are $\leq c \cdot |X|$ edges. If every Y node had degree $> c \cdot |X|/|Y|$, then we would have $> c \cdot |X|$ edges. Contradiction. ∎.

Corollary 4.9 If we have a graph with X nodes of degree ≤ 3, and $|X| < |Y|$ then some Y node must have degree ≤ 2. ∎

This gives us our first bound on the overall length of the schedule that the approximation algorithm will give us.

Lemma 4.10 The approximation algorithm will always return a schedule with a length of at most $1.5m + 2$ for a graph with max Y degree $= 3$.

Proof: The way to determine the length is to find a limit on the number of idle Y jobs that can be needed. When we choose a Y job and schedule $\Gamma(y)$, we need $|\Gamma(y)| - 1$ idle Y jobs. We also need one additional idle Y job at time 1.

In the worst case the first Y node has $|\Gamma(y)| = 3$. Thereafter, Corollary 4.9 tells us that $|\Gamma(y)| \leq 2$, since $|X| < |Y|$. We can have $|\Gamma(y)| = 2$ for at most $(m-3)/2$ y-jobs. So we have a total of at most $(m-3)/2 + 3$ idle jobs. Thus our overall length is $\leq 1.5m + 2$. ▮

But we can do even better than that. This is because if the schedule is really using one idle Y job per Y job released, then we will get to a point where $|Y| > 1.5 \cdot |X|$. If that happens, then Lemma 4.8 tells us that there is a Y node of degree $= 1$. Thus it would seem that we wouldn't need any more idle y-slots after that time. There is one complication. The X node may release more than one Y node. What happens if the balance of X's and Y's shifts back so that $|Y| = 1.5 \cdot |X|$ or $|Y| < 1.5 \cdot |X|$? By careful counting this can be shown to not be a problem.

Lemma 4.11 Given a graph with degree ≤ 3 for the X nodes, and with $m_Y > 1.5 \cdot m_X$, where $m_X = |X|$ and $m_Y = |Y|$. The approximation algorithm will schedule this with only one idle Y job.

Proof: The idle Y job will obviously be at time 1. What we need to show is that if we run into a situation where all the Y nodes have degree ≥ 2, then we will have accumulated enough excess released Y jobs that we can execute them instead of having an idle Y job. To do this we will show that the invariant

$$|Y| + excess - 1 \geq |X| + m_X/2$$

always holds as $|X|$ and $|Y|$ vary and that *excess*, the number of excess available y jobs, is always ≥ 0.

At the start of our execution, $|X| = m_X$ and $|Y| = m_Y$. Since $m_Y > 1.5 \cdot m_X$, then $|Y| - 1 \geq 1.5 \cdot |X|$. Since $excess = 0$, the invariant holds at the beginning of the execution.

How does it change as we go along?

CASE 1: We can schedule one X job which releases $k \geq 1$ Y jobs. Since $|Y|$ decreases by k, *excess* increases by $k-1$ and $|X|$ decreases by 1, the invariant remains satisfied.

CASE 2: We need to schedule two X jobs in order to release $k \geq 1$ Y jobs. First *excess* decreases by 1, then $|Y|$ decreases by k, *excess* increases by $k-1$ and $|X|$ decreases by 2. At the end the invariant is satisfied, but we need to verify that $excess \geq 1$ at the beginning. Since we needed two X jobs, we know that initially $|Y| \leq 1.5 \cdot |X|$. Since $|Y| + excess - 1 \geq |X| + m_X/2 \geq 1.5 \cdot |X|$, then $excess \geq 1.5 \cdot |X| - |Y| + 1 \geq 1$.

CASE 3: We need to schedule three X jobs in order to release $k \geq 1$ Y jobs. First *excess* decreases by 2, then $|Y|$ decreases by k, *excess* increases by $k-1$ and $|X|$ decreases by 3. At the end the invariant is satisfied, but we need to verify that *excess* ≥ 2 at the beginning. Since we needed three X jobs, we know that initially $|Y| \leq |X|$. Since $|Y| + excess - 1 \geq |X| + m_X/2$, then $excess \geq m_X/2 + |X| - |Y| + 1 \geq m_X/2 + 1$. For case 3 to be possible, we must have $m_X \geq 3$, so everything is OK.

▮

So what is the largest number of idle jobs we can need before we hit the first crossover point where $|Y| > 1.5 \cdot |X|$?

For the portion of the schedule before we hit $|Y| > 1.5 \cdot |X|$, let j_i be the number of Y jobs that had $|\Gamma(y)| = i$ when they were picked by the algorithm. We will have used $2j_3 + j_2 + 1$ idle jobs. When we hit the crossover point, $|Y| = m_Y - j_3 - j_2 - j_1$ and $|X| = m_X - 3j_3 - 2j_2 - j_1$. At crossover $|Y| \leq 1.5 \cdot |X| + 1$, so $m_Y - j_3 - j_2 - j_1 < 1.5m_X - 4.5j_3 - 3j_2 - 1.5j_1 + 1$. Simplifying, and plugging in $j_3 = 1$, gives $2j_2 + j_1/2 < (m_X - 5)/2$. We want to choose j_2 and j_1 to maximize $j_2 + 3$, which means $j_1 = 0$ and $j_2 = (m_X - 5)/4$. Hence the overall idle time is $(m_X + 7)/4$.

Thus we can replace Lemma 4.10 with

Lemma 4.12 The approximation algorithm will always return a schedule with a length of at most $(5m+7)/4 + 1$ for a graph with max Y degree $= 3$. ▮

Figure 4.11 shows a near worst case example where this bound is almost achieved because of extreme unluckiness by the algorithm when it chooses between nodes of equal degree. The $6/8m$ unshown edges from A go to the nodes in B' and C'; each node in B' and C' touches exactly one of these edges. The $3/8m$ unshown edges from B go to the nodes in C'; each node in C' touches exactly one of these edges. We'll specify more detail in a moment. This detail is enough to see that the schedule length is $5m/4 + 1$ without the dashed edge and $(5m+4)/4 + 1$ with the dashed edge.

How does this compare to the optimum for this graph? Not very well. By filling in the unspecified edges in the appropriate pattern, we can repeatedly alternate scheduling A, B and C nodes in the pattern shown by Figure 4.12. There will be some initial start-up delay. But once we get over that constant amount of borrowing, we won't need any more.

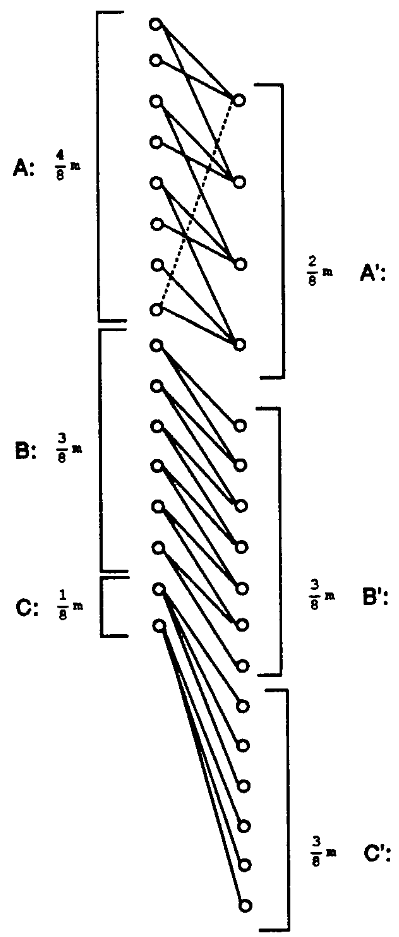


Figure 4.11: Worst Case for Approximation

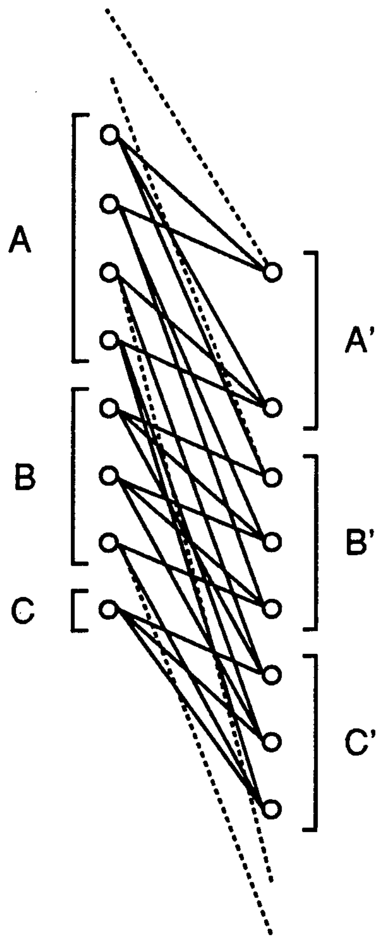


Figure 4.12: A constant borrowing schedule

Hence,

$$\rho = \frac{approx}{opt} = 1.25 + O(1/n)$$

So that's as well as *this* approximation scheme does. What about other approximation algorithms? Can we find an algorithm that will approximate to within a constant? This turns out not to be possible unless we could solve PLT itself.

Lemma 4.13 If, using a polynomial amount of time, we could approximate the schedule for an arbitrary graph such that

$$approx - opt \leq c$$

then PLT $\in P$.

Proof: Take an instance of PLT. Multiply the weights by $c + 1$. If we can approximate the new problem to within a constant c, then this gives us an optimum for the original. ▮

What about $approx - opt \leq \log n$ or $approx - opt \leq n^{1/2}$?

Assume that we can approximate so that $approx - opt \leq f(n)$ for some function $f(n)$. We will multiply the weights of the graph by another function $g(n) + 1$. If $f(n \cdot (g(n) + 1)) < g(n) + 1$ then we can use the approximation to the weighted version to solve the original.

Assume that $f(n) = n^{\epsilon}$ for some ϵ. Let's choose $g(n) = n^{(1+\delta)\epsilon} - 1$. For what values of ϵ can we find a δ such that $f(n \cdot (g(n) + 1)) < g(n) + 1$?

We want $(n(n^{(1+\delta)\epsilon}))^{\epsilon} < n^{(1+\delta)\epsilon}$. So

$$\begin{array}{rcl} \epsilon + (1+\delta)\epsilon^2 & < & (1+\delta)\epsilon \\ (1+\delta)\epsilon^2 & < & \delta\epsilon \\ (1+\delta)\epsilon & < & \delta \\ \epsilon & < & (1-\epsilon)\delta \\ \epsilon/(1-\epsilon) & < & \delta \end{array}$$

That is, for every constant $\epsilon < 1$, there is a constant δ such that $f(n \cdot (g(n) + 1)) < g(n) + 1$. Hence,

Lemma 4.14 Any polynomial time approximation algorithm with a bound of $approx - opt \leq f(n) = n^{\epsilon}$ for $\epsilon < 1$ implies a polynomial solution to PLT itself. ▮

Chapter 5

Lower Bound Techniques

The type of approximation algorithms that we were just considering give upper bounds on what the actual overhang is. It is also interesting to be able to obtain lower bounds on the overhang.

5.1 Simple Methods

A few lower bound techniques are easy to validate.

Lemma 5.1 If M contains a bipartite clique $K_{c,d}$ as a subgraph and $c, d > 0$, then the overhang of M is $\geq c + d - |Y|$.

Proof: Even if the clique is stuck in the lower-left-hand corner, as shown in Figure 5.1, its upper-right corner will protrude enough to ensure that the overhang is $\geq c + d - |Y|$. ▮

A clique which maximizes $c + d$ can be found in polynomial time by complementing the edge set ($E' = \{(x, y) \mid x \in X, y \in Y, (x, y) \notin E\}$) and then finding the maximum independent set. We can find the maximum independent set in polynomial time since our graph is bipartite.

Another obvious lower bounding technique is less computationally feasible.

Lemma 5.2 If every subset $Y' \subseteq Y$ of size c has $|\Gamma(Y')| \geq d$, then the overhang is $\geq d - c + 1$.

Proof: Let Y' be the first c nodes released by a schedule and let $X' = \Gamma(Y')$. Segregate X' to the front of S_X. Now, Y' and X' are in the upper left corner as shown in Figure 5.2.

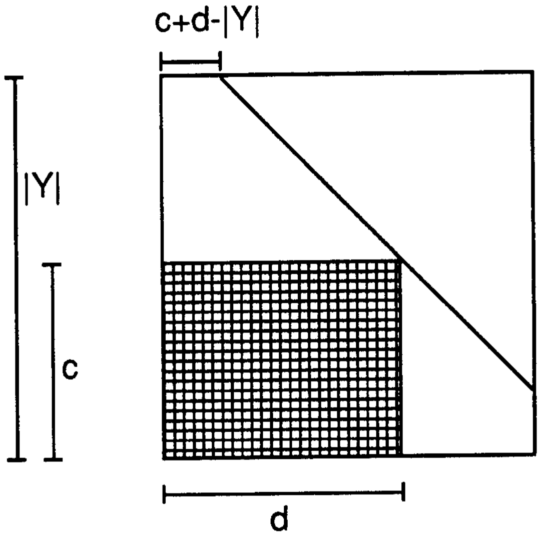


Figure 5.1: overhang $\geq c + d - |Y|$

There must be an $\times$ in each column of the sub-rectangle. If the lower-right corner of this sub-rectangle is occupied by an edge, then the overhang is $\geq d - c + 1$. If the $\times$ in that column where in any other row, then the overhang would be worse: $> d - c + 1$. Similarly, if X' isn't restricted to the first d columns, then the overhang would be $> d - c + 1$. ▮

Corollary 5.3 If every subset of X nodes of size i has $|\Gamma'(X)| \leq j$, then the overhang is $\geq i - j + 1$. ▮

This is a more general version of a previous result:

Lemma 5.4 [Too87] Given an $n \times n$ binary matrix M. plt$(M, 1) \Rightarrow$ there is no $2 \leq i \leq n$ such that i columns of the matrix each have $\geq n - i + 2$ 1s. ▮

As an example, consider the graph in Figure 5.3. Every set of Y nodes of size 2 is adjacent to 5 X nodes, so by Lemma 5.2 the overhang $\geq 5 - 2 + 1 = 4$. Equivalently, every set of 5 X nodes releases no more than 2 Y nodes, so by Corollary 5.3 the overhang $\geq 5 - 2 + 1 = 4$.

This technique is potentially exponential, and is not guaranteed to yield the correct result. For example, Figure 5.4 shows a graph (X nodes are filled in in black) with overhang 7 for which Lemma 5.2 can only show a lower bound of 6 (the dashed splines enclose a Y

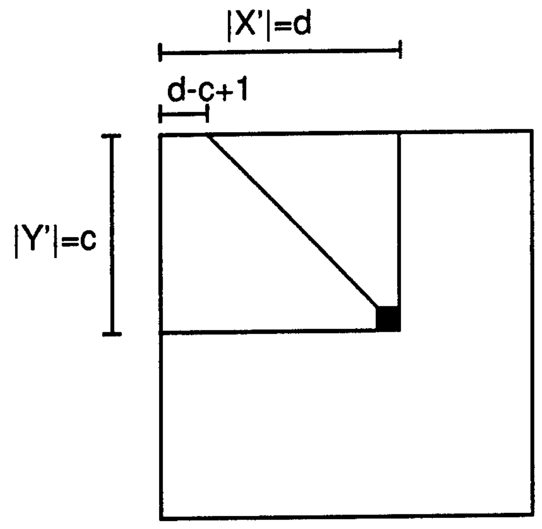


Figure 5.2: overhang $\geq d - c + 1$

$$\begin{array}{c|ccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
0 & \times & \cdot & \times & \times & \cdot & \cdot & \cdot \\
1 & \cdot & \times & \cdot & \times & \times & \cdot & \cdot \\
2 & \cdot & \cdot & \times & \cdot & \times & \times & \cdot \\
3 & \cdot & \cdot & \cdot & \times & \cdot & \times & \times \\
4 & \times & \cdot & \cdot & \cdot & \times & \cdot & \times \\
5 & \times & \times & \cdot & \cdot & \cdot & \times & \cdot \\
6 & \cdot & \times & \times & \cdot & \cdot & \cdot & \times
\end{array}$$

Figure 5.3: 3-regular graph

that minimizes $|\Gamma(Y)|$ over all Y of a fixed $|Y|$). To see that it has overhang 7, delete the dashed edge and the two nodes dangling from it. In the resulting graph every set of 7 Y nodes has 13 X nodes as neighbors, so Lemma 5.2 tells us that the overhang is 7. Adding the two nodes back to the graph can not reduce the overhang.

5.2 Permanents and Perfect Matchings

Matchings also shed some light on the overhang.

When a maximum matching touches every node in the graph, we say that it is a *perfect matching.* One useful function of an adjacency matrix M of a bipartite graph with $|X| = |Y| = n$ is the *permanent* of the matrix, which we will write as per(M). The permanent is equal to the number of different perfect matchings of the graph.

Another previous result is:

Lemma 5.5 [Too87] Given an $n \times n$ binary matrix M. plt$(M, 1) \Rightarrow$ the permanent of M is 0 or 1. ▮

This is straightforward once it is stated. Another straightforward result is that:

Lemma 5.6 Given an $n \times n$ binary matrix M. The permanent of M is 1 $\Rightarrow$ plt$(M, 1)$.

Proof: Since per$(M) = 1$, there is exactly one maximum matching of the graph. As we did in Section 2, let's reverse the directions of the edges in the maximum matching. The resulting graph is a directed-acyclic graph. Give the Y nodes a topological numbering. For each edge $y \rightarrow x$, give x the same number as y. If we schedule the X nodes and the Y nodes in the order of this numbering, the resulting matrix will be lower-triangular with 1s down the diagonal. ▮

Furthermore, Lemma 5.5 can be generalized to:

Lemma 5.7 If the $n \times n$ matrix M has overhang $k > 0$, then per$(M) \leq k^{n-k}k!$.

Proof: The question here is: what is the largest possible permanent for an $n \times n$ matrix with overhang k? We will call this $P(n, k)$. Clearly adding more 1s to the matrix can only increase the permanent. Hence what we need to figure out is the permanent of the matrix with all 1s below the k-diagonal. Figure 5.5 shows the case where $n = 6$ and $k = 3$. If

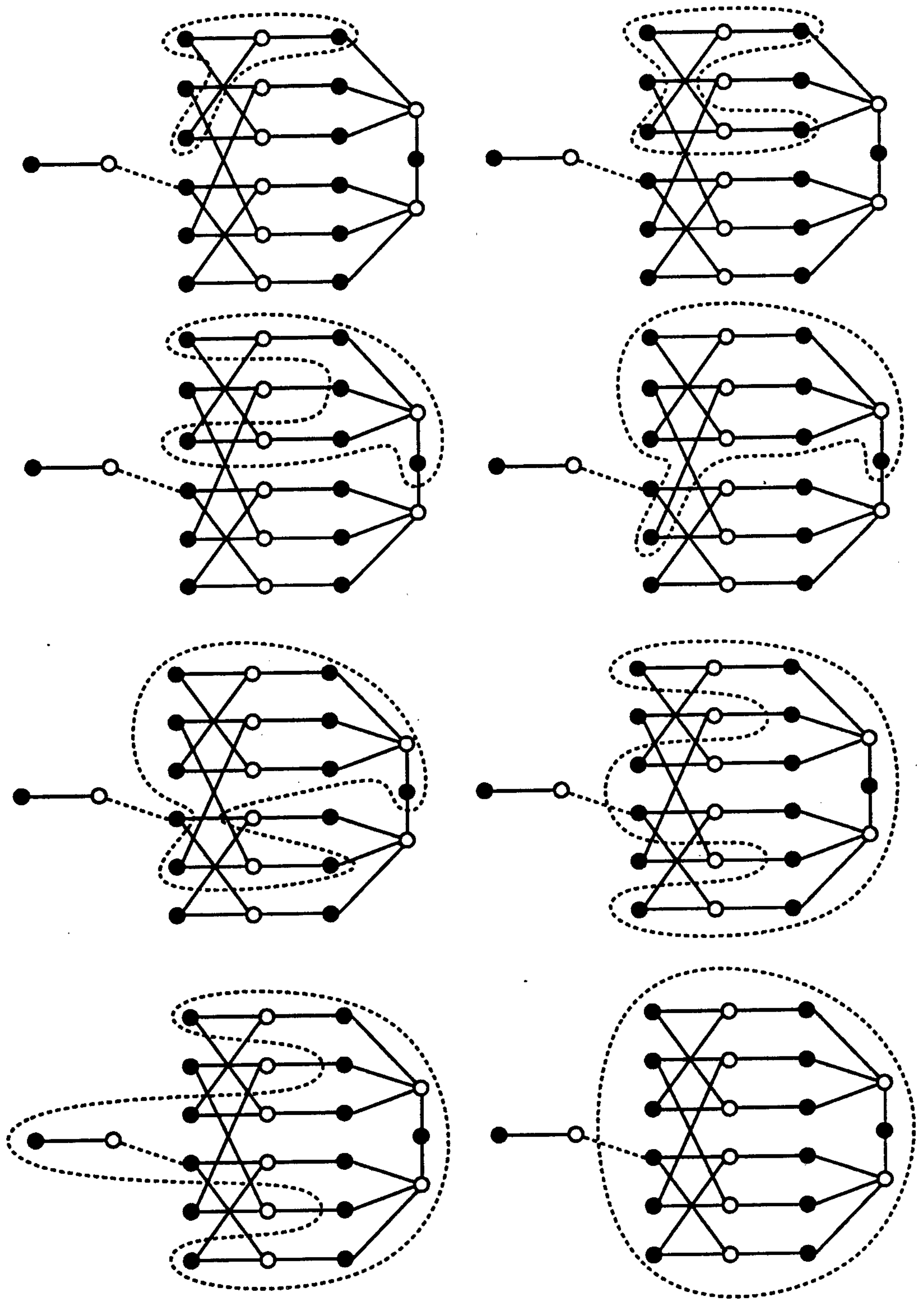


Figure 5.4: Lower bound from Lemma 5.2 is not tight.

$$\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
0 & \times & \times & \times & \cdot & \cdot & \cdot \\
1 & \times & \times & \times & \times & \cdot & \cdot \\
2 & \times & \times & \times & \times & \times & \cdot \\
3 & \times & \times & \times & \times & \times & \times \\
4 & \times & \times & \times & \times & \times & \times \\
5 & \times & \times & \times & \times & \times & \times
\end{array}$$

Figure 5.5: Matrix for $P(6,3)$

$k = n$, clearly the permanent is $n!$. That is, $P(n,n) = n!$. If $k < n$, then look at the column with k 1s in it. There are k choices for which row to match to that column. After we have made that choice, the remaining matrix is an $(n-1) \times (n-1)$ matrix with overhang k and with all 1s below the k-diagonal. That is, if $k < n$, $P(n,k) = k \cdot P(n-1,k)$. Hence, $P(n,k) = k^{n-k}k!$. ▮

The non-monotonicity of the implications of the permanent is interesting: $\text{per}(M) \geq 2 \Rightarrow$ 'no'; $\text{per}(M) = 1 \Rightarrow$ 'yes'; $\text{per}(M) = 0 \Rightarrow$ 'no information'.

5.2.1 Subarrays

Checking if the permanent $= 0$ or $= 1$ is easy. However, the problem of computing the permanent is #P-Complete [Val79]. That is, it is hard to compute. For the moment, we will ignore this and see what light the permanent sheds on our problem.

In particular, what peaks our interest are the following simple lemmas.

Lemma 5.8 If the matrix M has overhang k ($k \geq 1$) then it has an $(n-k+1) \times (n-k+1)$ subarray M' of overhang 1. ▮

Lemma 5.9 If the matrix M has a $n' \times n'$ subarray M' with overhang k ($k \geq 0$) then $\text{plt}(M, n-n'+k)$. ▮

Hence, if the matrix M has an $n' \times n'$ subarray M' with $\text{per}(M') = 1$, then $\text{plt}(M, n-n'+1)$, since M' has overhang 1.

By Lemma 5.5 the matrix M' in Lemma 5.8 must have $\text{per}(M') = 0$ or $\text{per}(M') = 1$. In the case where $\text{per}(M') = 0$, is there some *other* submatrix M'' of the same size such that $\text{per}(M'') = 1$?

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ 0 & \times & \times & \cdot & \cdot \\ 1 & \times & \times & \cdot & \cdot \\ 2 & \times & \times & \times & \times \\ 3 & \times & \times & \times & \times \end{array}$$

Figure 5.6: Counterexample

If so, then there always exists a maximum submatrix with $\text{per}(M'') \leq 1$, which actually has $\text{per}(M) = 1$. The following lemma almost gives us what we want.

Lemma 5.10 For a matrix M let M' be a largest submatrix with $\text{per}(M') \leq 1$. If $\text{per}(M') = 0$, then $\text{overhang}(M) \geq n - n' + 1$. If $\text{per}(M') = 1$, then $\text{overhang}(M) = n - n' + 1$.

Proof: Assume to the contrary that M has overhang $k < n - n' + 1$. Hence there is an $(n - k + 1) \times (n - k + 1)$ subarray M'' of overhang 1 (by Lemma 5.8). By Lemma 5.5, M'' has $\text{per}(M'') \leq 1$. But by assumption $k < n - n' + 1$ and hence $n - k + 1 > n'$. This contradicts our assumption that M' was the largest subarray with $\text{per}(M') \leq 1$. Hence, $\text{overhang}(M) \geq n - n' + 1$.

Furthermore, in the case where $\text{per}(M') = 1$, we know that $\text{overhang}(M') = 1$. Then Lemma 5.9 tells us that M has overhang $\leq n - n' + 1$. Hence, $\textit{overhang}(M) = n - n' + 1$.
▮

But unfortunately, our desired lemma is not true:

False Lemma 5.11 Let M' be the largest subarray of M with $\text{per}(M') = 1$. Then $\text{overhang}(M) = n - n' + 1$. ▮

The counterexample in Figure 5.6 has numerous 2×2 subarrays with permanent 1, no 3×3 subarrays with permanent 1, and yet has an overhang of only 2.

Another tempting but false lemma is:

False Lemma 5.12 Let the maximum subarray M' with $\text{per}(M') \leq 1$ have overhang k. Then the array M has overhang $n - n' + k$. ▮

As the maximum such subarray is $[y_0, y_1, y_2, y_3, y_4][x_4, x_5, x_6, x_7, x_8]$, the lemma would imply that the array in Figure 5.7 has overhang 7, when it actually has overhang 6. To

$$\begin{array}{c|ccccccccc}
 & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\
y_0 & \times & \times & \times & \times & \cdot & \cdot & \cdot & \cdot & \cdot \\
y_1 & \times & \times & \times & \times & \times & \times & \times & \times & \cdot \\
y_2 & \times & \times & \times & \times & \times & \times & \times & \times & \cdot \\
y_3 & \times & \times & \times & \times & \times & \times & \times & \times & \cdot \\
y_4 & \times & \times & \times & \times & \times & \times & \times & \times & \cdot \\
y_5 & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
y_6 & \cdot & \times & \times & \times & \times & \times & \times & \times & \times \\
y_7 & \cdot & \cdot & \times & \times & \times & \times & \times & \times & \times \\
y_8 & \cdot & \cdot & \cdot & \times & \times & \times & \times & \times & \times
\end{array}$$

Figure 5.7: Counterexample

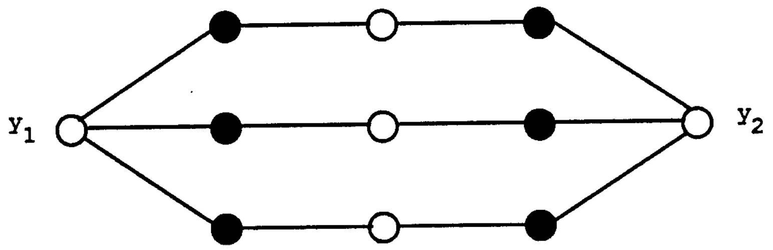


Figure 5.8: Flow of 3 from y_1 to y_2

see that it has overhang 6, simply transpose the matrix; that is, the following is a valid schedule:

$$\begin{aligned}
S_X &= [x_8, x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0] \\
S_Y &= [\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, y_8, y_7, y_6, y_5, y_4, y_3, y_2, y_1, y_0]
\end{aligned}$$

5.3 Network Flow

We can also use network flow techniques to obtain lower bounds on the overhang for a graph.

We will use sets of source nodes and sets of sink nodes. The sources and sinks will either all be X nodes or all Y nodes. We will set the capacities of the nodes and the edges to 1. Thus the flow is actually the number of vertex disjoint paths between the two sets of nodes.

Let's start out with the simplest case. Consider the graph in Figure 5.8.

Lemma 5.13 Given a unit-capacity graph and a single source node y_1 and a single sink node y_2. Let the maximum flow from source to sink be k and assume that the set of k disjoint paths uses every node in the graph. Then, the overhang of the graph is $\geq k$.

Proof: The basic idea is that there is no way to release either the source or the sink without borrowing k. Hence there is no way to schedule the whole graph without borrowing k.

Let the first source/sink released be in ΔT_j. Let's look at S_j and T_{j-1}. Claim $\tilde{n}_{j-1} = |T_{j-1}| - |S_j| \leq -k$.

To see this claim, we partition each of the sets S_j and T_{j-1} into k pieces. Since the graph has a flow of k between the source and sink, there are k vertex disjoint paths between y_1 and y_2. Number these paths from 1 to k. Let U_i be the nodes of S_j that are on path number i. Let V_i be the nodes of T_{j-1} that are on path number i. For all i, $|U_i| \geq |V_i| + 1$. To see this, look at two cases. If $|V_i| > 0$, then U_i must contain all the X nodes on path i that are adjacent to the Y nodes in V_i and hence $|U_i| \geq |V_i| + 1$. If $|V_i| = 0$, then $|U_i| \geq |V_i| + 1$, since in order for the source/sink to be released in ΔT_j, we must have $|U_i| \geq 1$ for all i. Thus, $|T_{j-1}| - |S_j| = \sum_{i=1}^{k} |V_i| - \sum_{i=1}^{k} |U_i| = \sum_{i=1}^{k}(|V_i| - |U_i|) \leq \sum_{i=1}^{k} -1 = -k$.

Hence, overhang $\geq k$. ▌

The cases where the flow actually uses up the whole graph will be rare. If the flow doesn't use all the nodes, then does the flow still give us some information? Yes.

Lemma 5.14 Given a unit-capacity graph and a single source node y_1 and a single sink node y_2. Let the maximum flow from source to sink be k. There may be several ways to achieve a k-flow. For each such k-flow, there will be some Y nodes that have no flow through them. If delay$(S) < k$, then for each possible k-flow from y_1 to y_2, S must release some Y node with no flow through it strictly before S releases y_1 or y_2.

Proof: Assume to the contrary that there is some schedule S with delay$(S) < k$ and there is some k-flow from y_1 to y_2 such that the source or the sink is released before a Y node with zero flow is released. Let the first source/sink released be in ΔT_j. The equations in Lemma 5.13 involving $|U_i|$ and $|V_i|$ still hold, and $|T_{j-1}| - |S_j| \leq -k$. Thus delay$(S)/geqk$. Contradiction. ▌

Using different choices for sources and sinks, this lemma gives us sets of conditions on

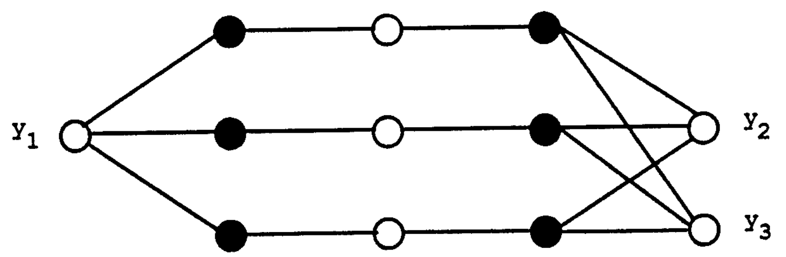


Figure 5.9: overhang = 3

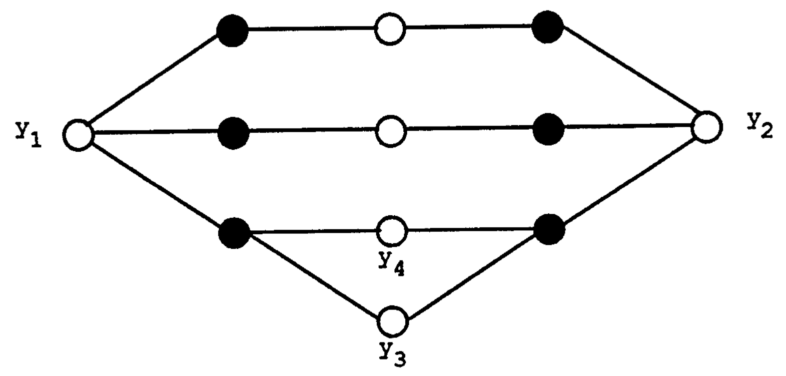


Figure 5.10: overhang = 2

allowable schedules. Along with other constraints, we can prove lower bounds on the delays of certain graphs.

Let's examine the example in Figure 5.9. There is a flow of 3 from y_1 to y_2 that uses all Y nodes except y_3. Lemma 5.14 tells us that y_3 must be released strictly before either y_1 or y_2 in any schedule with delay < 3. Similarly, the flow of 3 from y_1 to y_3 that uses all Y nodes except y_2 tells us that y_2 must be released strictly before either y_1 or y_3 in any schedule with delay < 3. Since these two conditions are contradictory, no delay < 3 schedule is possible, and overhang ≥ 3.

The example in Figure 5.10 is different. Here, there is a flow of 3 from y_1 to y_2 that uses all Y nodes except y_3. This means that y_3 must be released strictly before either y_1 or y_2 in any schedule with delay < 3. Similarly, there is a flow of 3 from y_1 to y_2 that uses all Y nodes except y_4 and this tells us that y_4 must be released strictly before either y_1 or y_2 in any schedule with delay < 3. These constraints present no contradiction, and there is indeed a schedule with a delay of 2.

One limitation of Lemmas 5.13 and 5.14 is that they can at best lower bound the

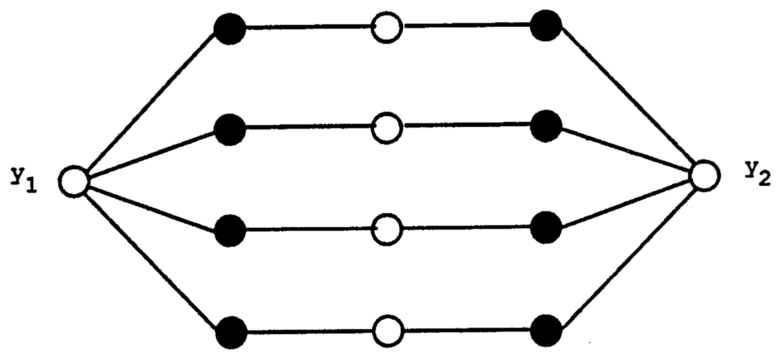


Figure 5.11: Flow of 4, before expanding y_1 and y_2

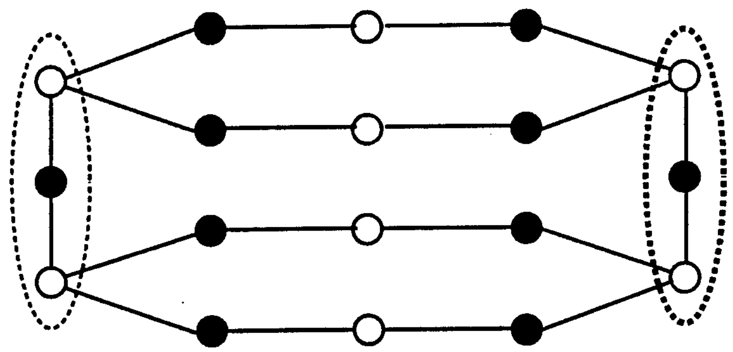


Figure 5.12: After expanding y_1 and y_2

overhang by the maximum degree of the graph. Clearly we should be able to do better: Suppose we start with the graph in Figure 5.11, which has a flow of 4, and then we apply the Expanding Transformation to it, yielding the equivalent graph in Figure 5.12, with a maximum degree 3. Now the flow can be at most 3. It would be nice if transforming the graph didn't diminish our ability to prove a lower bound. We will need a more powerful lemma. We need to use ***sets*** of sources and sinks in order to have a flow of 4.

We will also need to have ***ligament sets.*** A ligament set will be a set of X nodes that intuitively ties together the Y nodes of the sink (or source). For example, in Figure 5.13, $\{x_4\}$ is a valid ligament set for the sink node set $\{y_4, y_5\}$, and $\{x_1, x_2, x_3\}$ is a valid ligament set for the source node set $\{y_1, y_2, y_3\}$. For a source/sink node set Y', the ligament set $L(Y') \subseteq X$ must satisfy the following property:

In the graph induced by $(Y' \cup L(Y'))$, we must have $|\Gamma'(X')| \leq |X'|$ for every $X' \subseteq L(Y')$ that has $|X'| < |Y'| - 1$. That is, no subset X' of the ligament

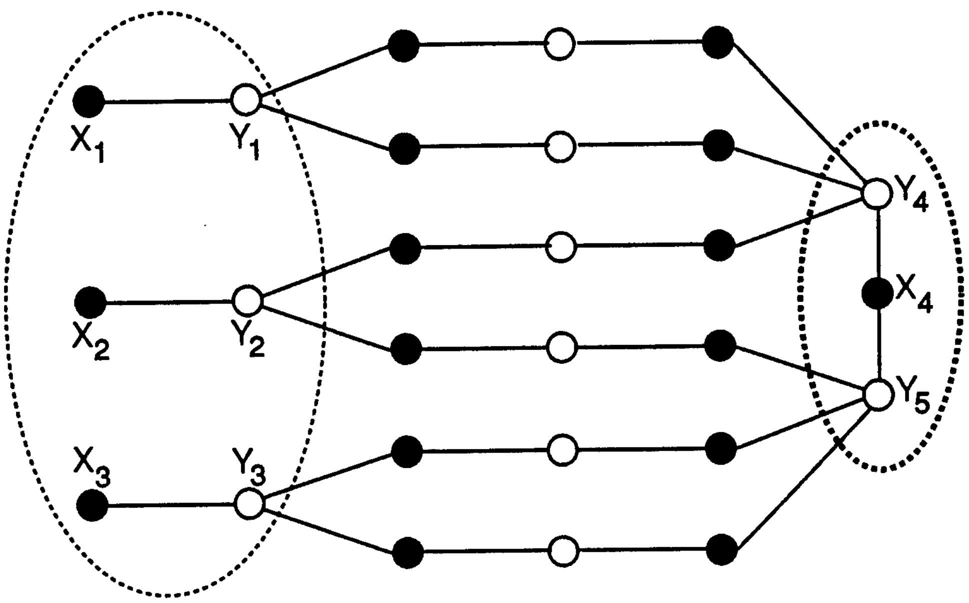


Figure 5.13: Example of ligament sets

nodes can release more than $|X'|$ nodes of the source/sink.

One consequence of this property is that $|L(Y')| \geq |Y'| - 1$, since otherwise $X' = L(Y')$ has $|X'| < |Y'| - 1$ and $|\Gamma'(X')| = |\Gamma'(L(Y'))| = |Y| > |X'| + 1 > |X'|$.

Note that the source and sinks sets don't need to be connected sets; for example, the source and sink sets shown in Figure 5.13 are allowed.

Lastly, we will not allow the flows to go through the ligament set. One way to visualize this is that $L(Y')$ and $L(Y)$ are removed from the graph before the flow is computed.

So what we have is this:

Lemma 5.15 We have a source node set Y_1 and a sink node set Y_2 with disjoint ligaments sets $L(Y_1)$ and $L(Y_2)$, respectively. Let the maximum flow in $G \setminus (L(Y_1) \cup L(Y_2))$ be k and assume that the set of k disjoint paths uses every node in the graph $G \setminus (L(Y_1) \cup L(Y_2))$. Then, the overhang of the graph G is $\geq k$.

Proof: The basic idea remains similar. There is no way to release either all the nodes in Y_1 or all the nodes in Y_2 without borrowing k. Hence there is no way to schedule the whole graph without borrowing k.

Assume without loss of generality that the schedule releases all of the nodes of the source before it releases all the nodes of the sink. Let the the last node of the source be released in ΔT_j. As before, let's look at S_j and T_{j-1}. Claim $\tilde{n}_{j-1} = |T_{j-1}| - |S_j| \leq -k$.

To see this claim, we partition each of the sets S_j and T_{j-1} into $k+2$ pieces. Since the graph has a flow of k between the source and sink, there are k vertex disjoint paths between y_1 and y_2. Number these paths from 1 to k. Let U_i be the nodes of S_j that are on path number i. Let V_i be the nodes of T_{j-1} that are on path number i. As before, for $1 \leq i \leq k$, $|U_i| \geq |V_i| + 1$.

We let U_{k+1} be the nodes of S_j that are in $L(Y_1)$. We let V_{k+1} be the nodes of T_{j-1} that are in Y_1. We let U_{k+2} be the nodes of S_j that are in $L(Y_2)$. We let V_{k+2} be the nodes of T_{j-1} that are in Y_2.

We claim that $|V_{k+1}| \leq |U_{k+1}|$. In verifying this, there are two cases. If $|U_{k+1}| \geq |Y_1| - 1$, then since $|V_{k+1}| < |Y_1|$, we see that $|V_{k+1}| \leq |U_{k+1}|$. If $|U_{k+1}| < |Y_1| - 1$, then the definition of ligament sets tells us that in the graph induced by $(Y_1 \cup L(Y_1))$, we have $|\Gamma'(U_{k+1})| \leq |U_{k+1}|$; since $V_{k+1} \subseteq \Gamma'(U_{k+1})$, then $|V_{k+1}| \leq |U_{k+1}|$. Similarly we can see that $|V_{k+2}| \leq |U_{k+2}|$.

Thus, $|T_{j-1}| - |S_j| = \sum_{i=1}^{k+2} |V_i| - \sum_{i=1}^{k+2} |U_i| = \sum_{i=1}^{k+2}(|V_i| - |U_i|) \leq \sum_{i=1}^{k} -1 = -k$.

Hence, overhang $\geq k$. ∎

Unfortunately, unlike with Lemma 5.13, there is no clear polynomial time algorithm to find the source and sink sets that yield the largest k.

As with the simpler source and sink sets, it is very useful to have a lemma that handles the case when the flows don't use all the nodes:

Lemma 5.16 We have a source node set Y_1 and a sink node set Y_2 with disjoint ligaments sets $L(Y_1)$ and $L(Y_2)$, respectively. Let the maximum flow in $G \setminus (L(Y_1) \cup L(Y_2))$ be k. If delay$(S) < k$, then for each possible k-flow from Y_1 to Y_2, S must release some Y node with no flow through it strictly before S releases either all the nodes of Y_1 or all the nodes of Y_2. ∎

For example, in Figure 5.14 the directed arcs show a flow of 7 from the sinks to the sources. Hence, for a schedule to achieve a delay less than 7, it must release y_1 or y_2 before it releases all the nodes of either the source or sink.

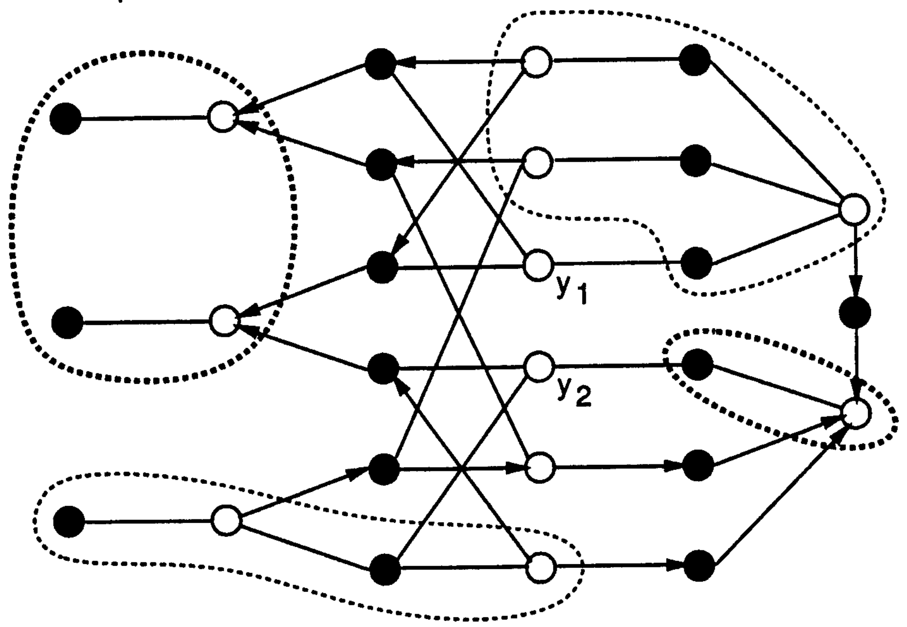


Figure 5.14: Flow of 7 that doesn't use y_1 and y_2

5.4 Minimal Graphs

Now that we have some lower bound results, we can prove that some sets of graphs are *minimal* graphs. That is, removing any edge from the graph reduces its overhang. One interesting reason to study minimal graphs is the possibility of finding a polynomial algorithm to verify them. If there were a polynomial algorithm that could take a graph and verify that it was a minimal graph of overhang k, then the PLT problem would be in *coNP* and hence in *coNP* $\cap$ *NP*.

5.4.1 Simple Minimals

One simple class of minimal graphs is suggested by Lemma 5.13. We take two Y nodes and string k disjoint paths between them as in Figure 5.15.

Before we prove that these are minimal, let's define a ***minimal ligament set***. In graph G, for a set Y, we will call a ligament set $L(Y)$ *minimal* if (1) $L(Y) \subseteq \Gamma(Y)$ and (2) after removing any edge between Y and $L(Y)$, the set $L(Y)$ no longer satisfies the ligament set property with respect to Y.

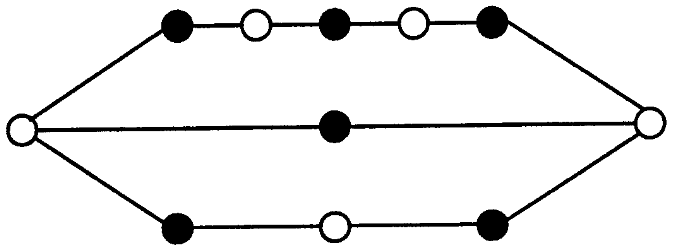


Figure 5.15: Minimal graph with overhang = 3

Lemma 5.17 Given a graph with a flow of k between disjoint source and sink sets such that (a) the ligament sets of the source and sink are minimal ligament sets and (b) every edge that is not in the ligament is used by the flow. Such a graph is a minimal graph with overhang k.

Proof: We already know that overhang $\geq k$. We will show that deleting any edge $e = (x, y)$ leaves overhang $= k - 1$. There are two cases.

If the deleted edge e is a flow edge: (Follow along in Figure 5.16.) In the graph $G \setminus e$, there is a unique path from y to a source/sink node. Schedule the X nodes along this path in order (x_1). Schedule all remaining X nodes adjacent to that source/sink node (x_2, x_3). Iterate through each remaining Y node that is in the same source/sink node set, scheduling all remaining X nodes adjacent to the Y node (x_4, x_5). Then repeat until done: for each Y node of degree 1, schedule its adjacent X node (x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}).

If the deleted edge e is between a ligament node and a source/sink node: (Follow along in Figure 5.17.) Removing the edge breaks the ligament set into two pieces. For the piece that contains y, iterate through each remaining Y node, scheduling all remaining X nodes adjacent to the Y node (x_1, x_2). Then as long as possible: for each Y node of degree 1, schedule its adjacent X node (x_3, x_4, x_5, x_6, x_7). If anything remains, it includes the other source/sink node set. Iterate through each remaining Y node that is in that source/sink node set, scheduling all remaining X nodes adjacent to the Y node (x_8, x_9, x_{10}). Then repeat until done: for each Y node of degree 1, schedule its adjacent X node (x_{11}, x_{12}, x_{13}).

∎

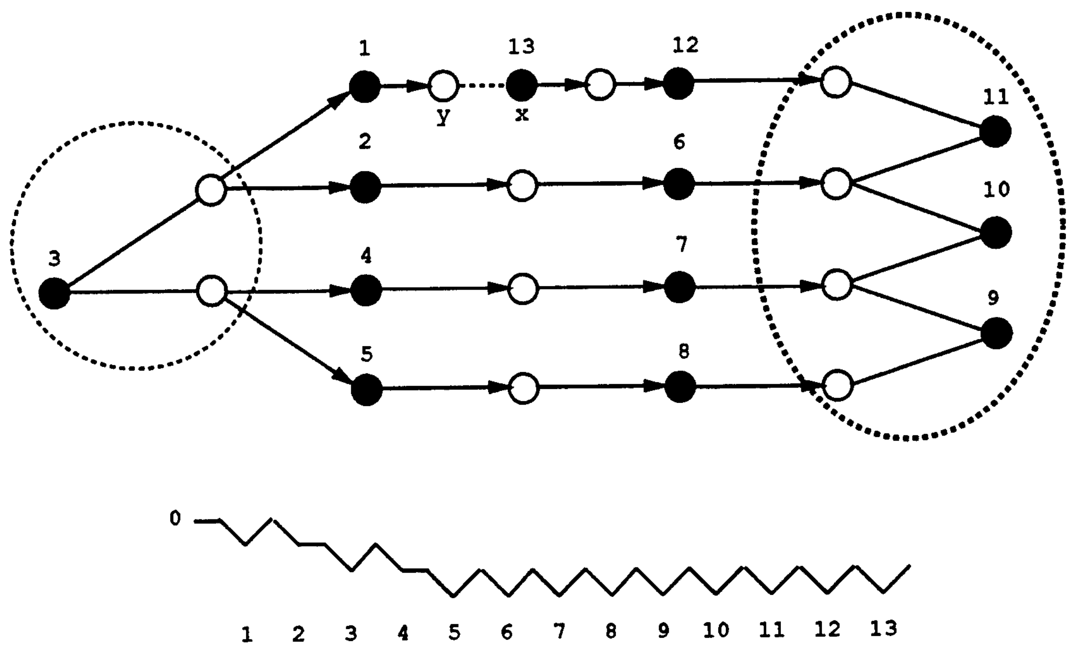


Figure 5.16: deleting a flow edge

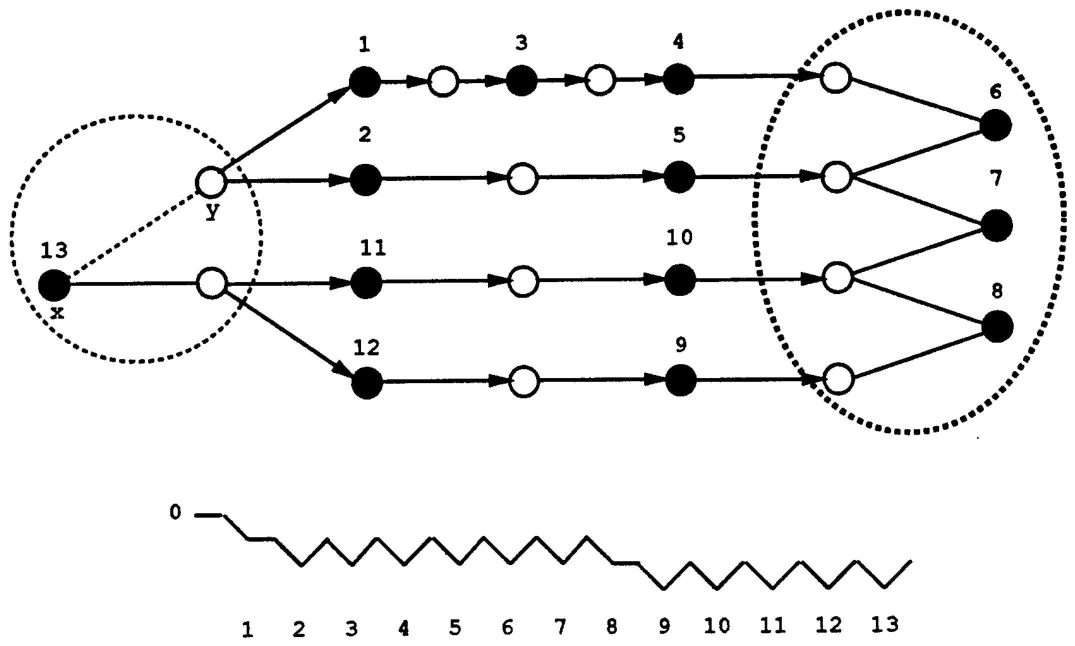


Figure 5.17: deleting a ligament edge

So besides proving that Figures 5.15 is minimal with overhang 3, this shows that Figure 5.13 is minimal with overhang 6. Note that while we already knew that these were the overhangs of the graphs, we did not know that the graphs were minimal, since the following lemma is false.

False Lemma 5.18 Let G' be a graph derived from G by applying the Collapsing Transformation. If G' is minimal for overhang k, then so is G. ▮

A simple counterexample can be seen in Figure 4.2. Both graphs have overhang 1. The graph on the right is minimal, but deleting the $x_1 \rightarrow y$ edge or the $x_2 \rightarrow y$ edge does not reduce the overhang of the graph on the left.

5.4.2 Disconnected Minimals

These simple minimals will always be connected graphs. It turns out that under the right conditions a set of unconnected minimals will be a minimal graph.

But, we must build up some tools, first.

Lemma 5.19 A connected minimal graph is either an **ascent** piece, a **descent** piece or a **plateau** piece.

Proof: Assume to the contrary that the graph has two or more such pieces. Since the graph is connected, there is an edge between two of the pieces. If this edge is removed, the pieces remain unchanged and their relative ordering is unchanged. That is, the well-ordered optimum schedule and hence the overhang remains unchanged. Thus the graph is not minimal. ▮

Lemma 5.20 For a connected minimal graph G that is an **ascent** with overhang $= k$, all MMBNPSs of G have borrowing $= -k$.

Proof: Assume to the contrary that the first MMBNPS piece, $S = [x_1, \ldots, x_z]$ is a (b, p) piece and that it has borrowing $b > -k$. So the situation is that shown on the left in Figure 5.18, where the initial MMBNPS S does not dip down the furthest. Hence, $G \setminus (S \cup \Gamma'(S))$ must have overhang $= k + p$.

Let's find a maximum matching in the graph induced by $(S \cup \Gamma'(S))$. It will be of size z. In G, other than these z matching edges, delete every other arc from nodes in the set

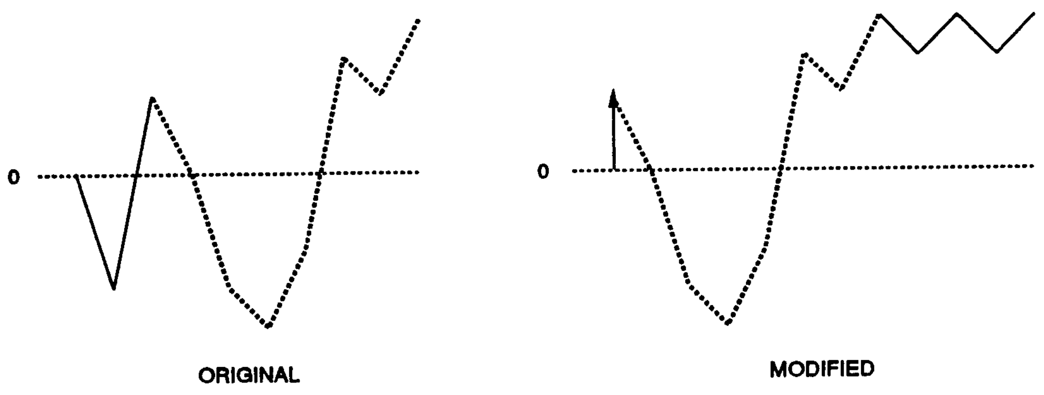


Figure 5.18: Charts for proof of Lemma 5.20

S. This will give us a modified graph G'. G' will now be $G \setminus (S \cup \Gamma'(S))$ plus $|\Gamma'(S)| - |S|$ isolated Y nodes and $|S|$ components each consisting of an X node, a Y node and an edge between them—$(-1, 0)$ pieces. The chart for this will now look like the chart on the right of Figure 5.18. G' still has overhang $= k$, and hence G could not have been minimal. ∎

Lemma 5.21 Let G be a graph which consists of many separate connected minimal subgraphs that are **ascent** subgraphs, and which each have overhang $= k$. Then G is minimal and has overhang $= k$.

Proof: From our merging algorithm and Lemma 5.20, we know that the overhang of the combined graph is k. If we delete any edge, then by scheduling the subgraph that contained that edge before the other subgraphs, we can find a schedule of overhang $= k - 1$. ∎

By Transposition:

Corollary 5.22 Let G be a graph which consists of m minimal graphs $(G_1, \ldots, G_m)$ which are **descent** subgraphs, and which each have overhang $= k$ and $|X_i| - |Y_i| = j$. Then G is minimal and has overhang $= m(k - j) + j$ and $|X| - |Y| = mj$. ∎

But these two flavors of minimal graphs don't mix:

Lemma 5.23 For a minimal graph G with overhang $= k$, there cannot be edges in both **ascent** and **descent** components.

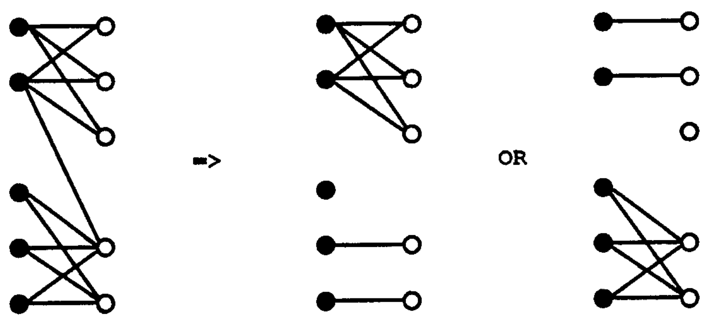


Figure 5.19: subgraphs that are minimal

Proof: Assume that the chart dips to a level of $-k$ during either the `plateau` or the `descent`. Then, similarly to 5.20, we delete all edge in the `ascent` except for a maximum matching within the `ascent`. In this modified graph, the matching edges from the `ascent` must now be in the `plateau`. The modified graph still has overhang $= k$.

Hence, if the chart of a minimal graph dips to $-k$ during either the `plateau` or `descent`, then there are no edges in the `ascent`. By transposition, if the chart of a minimal graph dips to $-k$ during either the `plateau` or `ascent`, then there are no edges in the `descent`. Since the must dip to $-k$ *somewhere*, then there either are no edges in the `ascent` or no edges in the `descent`. ▮

But, we can pepper either flavor of minimal graph with any number of isolated edges:

Lemma 5.24 Let G be a minimal graph which has overhang $= k$. Add to G two new nodes x and y and a single edge between them (x, y), yielding G'. Then G' is minimal and has overhang $= k$.

Proof: If the edge deleted is (x, y), then y is released immediately, and by scheduling x last, we will have reduced the overhang by 1. If the edge deleted is not (x, y), then schedule x and y last, and we will still have reduced the overhang by 1. ▮

So Figure 5.19 shows two different minimal subgraphs of the same graph.

Chapter 6

Games

6.1 A Game Theory Problem

The triangularization problem can also be framed as a two person game in the Conway sense [BCG82, Con76].

Consider the following game: There is a directed graph with Blue nodes and Red nodes. Blue and Red alternate turns. On a player's turn he must remove from the graph one node of his color that has indegree 0; all edges adjacent to the node are also deleted. The first player to have no legal moves on his turn is the winner.

Our triangularization problem PLT_1 corresponds to the special case of this game in which all of the directed arcs go from Blue nodes to Red nodes and Blue goes first. Can Blue win? What is his winning strategy? Note that in this special case, Red's choices are irrelevant to the outcome of the game.

Thus the question immediately arises: What use can we make of the theory of games that Conway et al develop? In Conway's theory he develops an algebra of position values. That is, each game position is assigned a value which has three properties:

1. ADD: The value of the sum of two games can be computed from the values of the two games.

2. AUGMENT: The value of a position can be computed from the values of its children positions.

3. WINNER: Given the value of a position, one can determine who would win the game if Blue started and who would win the game if Red started.

6.2 Misère Play

Unfortunately, his algebra is intended for *normal play* games. That is, games where the player who is stuck loses, in contrast to misère play where the player who moves last loses. The theory does not extend to misère play. In fact, no general value scheme for misère play is known — you need to essentially encode the entire position in your value.

Our game is a misère play game. Yet we can devise an algebra that satisfies all three properties and doesn't encode the entire problem in it. That is, it is very-many-to-1.

For the value of a position we will just use the fundamental chart of any optimal schedule of the corresponding graph.

There is an ADD procedure that is used in adding two positions: ADDing two independent games is the same thing that we earlier called merging two fundamental charts.

There is an AUGMENT procedure, which only needs to look at Blue's possible moves: How do we compute the value of a position given the values of its children? Let $v_1, \ldots v_k$ be the values of the children position for each of Blue's valid moves. We can easily compute $v_1', \ldots, v_k'$, the values of the initial position, assuming that move i is Blue's best move. Then we just need to pick out the best one of the v_i'. We already showed how to pick out the best chart from a set of charts when we were solving weighted chains in Section 3.4.

How do we determine the WINNER from the value?

If it is Blue's turn: If the fundamental chart dips below -1, then Red (Y) was stuck at some point, and Blue (X) loses. If the chart never dips below -1, and ends at -1, then Red is stuck and Blue loses. Otherwise, Blue will be stuck first, and so Blue wins.

If it is Red's turn: If the fundamental chart is ever below 1, then Red (Y) was stuck at some point, and Blue (X) loses. Otherwise, Blue will be stuck first, and so Blue wins.

So our value satisfies all three properties.

6.2.1 A Misère Hackenbush variant

Obviously our algebra can't be extended to cover arbitrary misère games. However, there is one form of misère Hackenbush that it can handle: misère childish red-blue hackenbush with stalks that have blue stems and red tips. That is, starting from the ground, the edges are blue. At some point they turn red and stay red, all the way to the tips.

6.3 Normal Play

When we play in a non-Misère fashion, we can make full use of the system of assigning values to games developed by Conway.

Definition 6.1 For a bipartite graph $G = (X, Y, E)$, a *Y-dominating set* is a set $X' \subseteq X$ such that for any $y \in Y$ there is some $x \in X'$ such that $(x, y) \in E$, i.e. $\Gamma(X') = Y$.

Lemma 6.1 If G is such that X is a minimal Y-dominating set for Y, then the value of G is 0.

Proof: If it is Y's turn to move, he has no legal moves and he loses. If it is X's turn to move, then each X node that he removes will release at least one Y node, and hence he will ultimately lose. A game in which whoever starts is the loser is a 0 game. ▮

Lemma 6.2 If G consists of an isolated Y node, then G has the value -1.

Proof: If it is Y's turn, his move must leave an empty graph. An empty graph is a special case of Lemma 6.1 and thus has value 0. If it is X's turn, he has no legal move. So the value of G is $\{\, | \, 0\} = -1$. ▮

Lemma 6.3 If G has a minimum Y-dominating set of size $d \geq 1$, then the value of the game is $|X| - d$.

Proof: We will use induction on $|X|$. The case when $|X| = 1$ and hence $d = 1$ is covered by Lemma 6.1.

The case when $|X| - d = 0$ is also covered by Lemma 6.1, so we only need to consider the general case where $|X| - d > 0$. Once again, since there exists a Y-dominating set, player Y has no legal moves. But X has three types of moves: moves that leave a minimum Y-dominating set of size d; moves that release a Y node, thus reducing the size of the minimum Y-dominating set to $d - 1$; and moves that don't do either.

Since $|X| - d > 0$, moves of the first type always exist; these moves are when we remove some X node that is not in all of the minimum Y-dominating sets of size d. They leave a graph G' with $|X'| = |X| - 1$ and $|X'| - d = |X| - d - 1$ and hence, by induction, with value $|X| - d - 1$.

Moves of the third type are when we remove a node that is in all of the minimum Y-dominating sets of size d, and yet it doesn't release a Y node. Hence, it leaves a graph G' with $d' > d$ and $|X'| - d' < |X| - d - 1$ and hence, by induction, with value $< |X| - d - 1$. This type of move is never desirable for X.

After a move of the second type, the graph G' has two parts. It has $j \geq 1$ isolated Y nodes and a piece with no isolated Y nodes, which we will call G''. It is the case that $d'' = d - 1$ and $|X''| = |X| - 1$. So, by induction on $|X|$, the value of G'' is $|X''| - d'' = |X| - d$. The value of G' is $|X| - d - j \leq |X| - d - 1$.

Since X can always leave a game of value $v = |X| - |d| - 1$ and never leave a game of larger value, the overall value of the game G is $\{v \mid \} = |X| - d$. ▌

So, we can find the value of a game iff we can find its minimum Y-dominating set. Unfortunately, this turns out to be NP-Complete.

Y-DOM-SET

INSTANCE: A bipartite graph $G = (X, Y, E)$ and a bound k.

QUESTION: Is there a set $X' \subset X$ such that $|X'| \leq k$ and $\Gamma(X') = Y$?

Lemma 6.4 Y-DOM-SET is NP-Complete — even if all elements of Y have degree ≤ 2.

Proof: Reduction from minimum vertex cover set. Given an instance of vertex cover set, G=(V,E) and a bound k. Is there a set $V' \subseteq V$ such that $|V'| \leq k$ and for every edge $e = (v_i, v_j) \in E$, either $v_i \in V'$ or $v_j \in V'$?

We convert this into an instance of Y-dom-set $G' = (X', Y', E')$ as follows: For every vertex $v_i \in V$ create a node $x_i \in X$. For every edge $e_i \in E$ create a node $y_i \in Y$. If vertex v_i is an endpoint of edge e_j, then put an edge in E' between x_i and y_j. Is there a Y-dominating set of size $\leq k$? ▌

Chapter 7

Conclusions

We have taken the triangularization problem, recast it in a number of different forms, and twisted it this way and that holding it up to a variety of light sources. What have we learned from this?

We have seen that the triangularization problem is very closely related to a number of scheduling problems that demand to be solved as computer engineers try to leave less of their available hardware idle. It is easy to believe that some of the tools we have honed in this thesis will prove useful in dealing with those scheduling problems.

We have seen a number of NP-complete problems that are very close to the triangularization problem. Perhaps someone else can close the gap and show that triangularization is NP-complete. Such a proof would probably require some very interesting methods.

We have developed a tool for segregating schedules. The first thing that this allowed us to do is break up schedules into a number of independent sub-schedules. Ultimately, this led us to well-ordered optimum schedules, fundamental charts, and nearly well-ordered schedules.

In fundamental charts, we have found a very simple description of a graph. It is a description that abstracts away all the unnecessary and confusing chaos of the wild crossing of edges from node to node. It is also a description that allows us to merge solutions for independent graphs in polynomial time, without reconsidering the interconnections of edges and nodes.

Besides being fascinating in their own right, these properties form the underpinnings of our polynomial algorithm to schedule weighted chains. We have also shown that graphs with all Y nodes of degree ≤ 2, graphs with all X nodes of degree ≤ 2, and graphs that are trees are all solvable in polynomial time. However, more interesting than any of these is the polynomial algorithm for solving PLATEAU pieces with overhang $\leq k$.

The next hardest problem to try to get a polynomial algorithm for is graph that are weighted trees. We have seen that both weighted chains and unweighted trees can be solved in polynomial time. Even a solution for weighted binary trees would be very interesting.

While our approximation results seem discouraging, it should be remembered that these dealt with worst-case behavior of approximation algorithms. Practical approximation algorithms should be pursued, but this can only be done effectively with a source of real instances.

Lastly, better lower bound techniques need to be pursued. For example, is there a family of graphs with all Y vertices of degree ≤ 3 whose overhang is $\Omega(n)$? (We have proved Lemma 4.12 that the overhang of such problems is at most about $n/4$.) Any progress here will aid our understanding of the triangularization problem.

Many aspects of this problem remain elusive. Something interesting *is* going on here.

Bibliography

[BCG82] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways, For Your Mathematical Plays.* Academic Press, 1982.

[CG72] E. G. Coffman and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[Con76] John H. Conway. *On Numbers and Games.* Academic Press, 1976.

[FERN84] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. *SIGPLAN Notices*, 19(6):37–47, June 1984.

[Fis79] Joseph A. Fisher. *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources.* PhD thesis, New York University, October 1979. U.S Department of Energy Tech Report COO-3077-161.

[Gab82] Harold N. Gabow. An almost-linear algorithm for two-processor scheduling. *JACM*, 29(3):766–780, July 1982.

[GGJK78] M. R. Garey, R. L. Graham, D. S. Johnson, and D. E. Knuth. Complexity results for bandwidth minimization. *SIAM J. Applied Math*, 34(3):477–495, May 1978.

[GM86] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16. ACM, 1986.

[Goy76] Deepak K. Goyal. Scheduling processor bound systems. Technical Report CS-76-036, Computer Science Department, Washington State University, Pullman, Washington, 1976.

[Goy77] Deepak K. Goyal. Scheduling processor bound systems. In *Proceedings of the Sicth Texas Conference on Computing Systems*, pages 7B21–7B30. UT, Austin, November 1977.

[Gro83] Thomas Gross. *Code Optimization of Pipeline Constraints*. PhD thesis, Stanford University, December 1983. Computer Systems Laboratory Tech Report 83-255.

[GS84] Eitan M. Gurari and Ivan Hal Sudborough. Improved dynamic programming algorithms for bandwidth minimization and the MinCut linear arrangement problem. *Journal of Algorithms*, 5:531–546, 1984.

[HG83] John Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[Lam87] Monica Sin-Ling Lam. *A Systolic Array Optimizing Compiler*. PhD thesis, Carnegie Mellon University, Computer Science Department, May 1987. Tech Report CMU-CS-87-187.

[LDSM80] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallett. Local microcode compaction techniques. *Computing Surveys*, 12(3):261–294, September 1980.

[LLMS87] Eugene Lawler, Jan Karel Lenstra, Charles Martel, and Barbara Simons. Pipeline scheduling: A survey. Technical Report RJ 5738 (57717), IBM, July 1987.

[LLRS85] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. Technical Report BS-R89xx, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1985.

[LVW84] Joseph Y-T. Leung, Oliver Vornberger, and James D. Witthoff. On some variants of the bandwidth minimization problem. *Siam J. Computing*, 13(3):650–667, August 1984.

[Sax80] James B. Saxe. Dynamic-programming algorithms for recognizing small-bandwidth graphs in polynomial time. *Siam J. Alg Disc Methods*, 1(4):363–369, December 1980.

[Too87] N. Toobaei. Some necessary conditions for a (0, 1)-matrix to be triangulizable. *AMS Notices*, page 7, January 1987.

[Tou84] Roy F. Touzeau. A Fortran compiler for the FPS-164 scientific computer. *SIGPLAN Notices*, 19(6):48–57, June 1984.

[Ull76] J. D. Ullman. The complexity of sequencing problems. In E. G. Coffman, Jr., editor, *Computer and Job-Shop Scheduling Theory*, pages 139–164. John WIley & Sons, 1976.

[Val79] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Veg82] Steven R. Vegdahl. *Local Code Generation and Compaction in Optimizing Microcode Compilers*. PhD thesis, Carnegie Mellon University, Computer Science Department, December 1982. Tech Report CMU-CS-82-153.